

6th International Workshop on Formal Methods for Blockchains

FMBC 2025, May 4, 2025, Hamilton, Canada

Edited by

Diego Marmosler

Meng Xu



Editors

Diego Marmsoler 

University of Exeter, UK
D.Marmsoler@exeter.ac.uk

Meng Xu 

University of Waterloo, Canada
meng.xu.cs@uwaterloo.ca

ACM Classification 2012

Security and privacy → Formal methods and theory of security; Security and privacy → Logic and verification; Theory of computation → Program verification; Software and its engineering → Formal software verification; Security and privacy → Distributed systems security; Computer systems organization → Peer-to-peer architectures

ISBN 978-3-95977-371-3

Published online and open access by

Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at <https://www.dagstuhl.de/dagpub/978-3-95977-371-3>.

Publication date

May, 2025

Bibliographic information published by the Deutsche Nationalbibliothek

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

License

This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0): <https://creativecommons.org/licenses/by/4.0/legalcode>.



In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/OASlcs.FMBC.2025.0

ISBN 978-3-95977-371-3

ISSN 1868-8969

<https://www.dagstuhl.de/oasics>

OASlcs – OpenAccess Series in Informatics

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

Editorial Board

- Daniel Cremers (TU München, Germany)
- Barbara Hammer (Universität Bielefeld, Germany)
- Marc Langheinrich (Università della Svizzera Italiana – Lugano, Switzerland)
- Dorothea Wagner (*Editor-in-Chief*, Karlsruher Institut für Technologie, Germany)

ISSN 1868-8969

<https://www.dagstuhl.de/oasics>

■ Contents

Preface	
<i>Diego Marmosoler and Meng Xu</i>	0:vii
Program Committee Chairs	
.....	0:ix
Steering Committee	
.....	0:xi
Program Committee	
.....	0:xiii
Supporting Reviewers	
.....	0:xv

Invited Talks

Is Formal Verification Practical?	
<i>Wolfgang Grieskamp</i>	1:1–1:2
Bringing the Power of Interactive Theorem Proving to Web3	
<i>Julian Sutherland</i>	2:1–2:1

Regular Papers

Formal Verification in Solidity and Move: Insights from a Comparative Analysis	
<i>Massimo Bartoletti, Silvia Crafa, and Enrico Lipparini</i>	3:1–3:18
ByteSpector: A Verifying Disassembler for EVM Bytecode	
<i>Franck Cassez</i>	4:1–4:15
Towards a Mechanization of Fraud Proof Games in Lean	
<i>Martín Ceresa and César Sánchez</i>	5:1–5:17
Validity, Liquidity, and Fidelity: Formal Verification for Smart Contracts in Cardano	
<i>Tudor Ferariu, Philip Wadler, and Orestis Melkonian</i>	6:1–6:21
A Readable and Computable Formalization of the Streamlet Consensus Protocol	
<i>Mauro Jaskelioff, Orestis Melkonian, and James Chapman</i>	7:1–7:18
Formal Verification of a Fail-Safe Cross-Chain Bridge	
<i>Filip Marić, Bernhard Scholz, and Pavle Subotić</i>	8:1–8:18
Verifying Smart Contract Transformations Using Bisimulations	
<i>Kegan McIlwaine and James Caldwell</i>	9:1–9:19
Program Logics for Ledgers	
<i>Orestis Melkonian, Wouter Swierstra, and James Chapman</i>	10:1–10:22
Formally Specifying Contract Optimizations with Bisimulations in Coq	
<i>Derek Sorensen</i>	11:1–11:13

6th International Workshop on Formal Methods for Blockchains (FMBC 2025).

Editors: Diego Marmosoler and Meng Xu



OpenAccess Series in Informatics

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

Tool Papers

Isabelle/Solidity: A Tool for the Verification of Solidity Smart Contracts <i>Asad Ahmed and Diego Marmosler</i>	12:1–12:9
A Benchmark Framework for Byzantine Fault Tolerance Testing Algorithms <i>João Miguel Louro Neto and Burcu Kulahcioglu Ozkan</i>	13:1–13:11
SCAR: Verification-Based Development of Smart Contracts <i>Jonas Schiffel and Bernhard Beckert</i>	14:1–14:13

■ Preface

This volume contains the proceedings of the 6th International Workshop on Formal Methods for Blockchains (FMBC 2025), to be held in Hamilton, Canada on May 04, 2025.

FMBC aims to bring together researchers and practitioners in the areas of formal methods and blockchain to promote a deeper understanding of how formal methods can be used for blockchain technology. Blockchain is a novel technology to store data in a decentralized way. Although the technology was originally invented to enable cryptocurrencies, it quickly found applications in several other domains. Since blockchains are often used to store financial transactions, bugs may result in huge economic losses and thus it is now of utmost importance to have strong guarantees of the behaviour of blockchain software. These guarantees can be brought by using formal methods. Indeed, blockchain software encompasses many topics of computer science where using formal methods techniques and tools is relevant: consensus algorithms to ensure the liveness and the security of the data on the chain, programming languages specifically designed to write smart contracts, cryptographic protocols, such as zero-knowledge proofs, used to ensure privacy, etc.

FMBC 2025 is the 6th International Workshop on Formal Methods for Blockchains, a series of workshops started in 2019. In past years, FMBC took place in Porto (2019, co-located with FM), online (2020 and 2021, co-located with CAV), Haifa (2022, co-located with CAV), and Luxembourg City (2024, co-located with ETAPS). This year FMBC attracted 18 submissions covering different areas of formal methods for blockchains. Each paper was reviewed by at least three reviewers and the Program Committee accepted 9 regular long papers and 3 tool papers.

FMBC 2025 would not have been possible without the deep investment and involvement of many supporters. We would like to express our gratitude to all the authors who submitted their work to the conference, the Steering Committee members who provided precious guidance and support, all the colleagues who served on the Program Committee, as well as the external reviewers, whose professional and efficient work during the review process helped us to produce a high-quality conference program. Particular thanks are given to the invited speakers, Julian Sutherland from Nethermind and Wolfgang Grieskamp from Aptos, for their willingness to talk about their research and share their perspective about formal methods for blockchains. The abstracts of the invited talks and a short bio of the speakers are included in this volume as well.

FMBC 2025 is co-located with ETAPS 2025, hosted and sponsored by McMaster University, Canada. Many thanks to all the local organizers and in particular to Alan Wasssyng and Angelo Gargantini, Workshop Chairs of ETAPS 2025, for their help and guidance. FMBC 2025 was financially supported by Aptos and Movement Labs.

April 2025

Diego Marmosoler
Meng Xu

APTOS

 **Movement** {LABS}

■ Program Committee Chairs

Diego Marmsoler
University of Exeter

Meng Xu
University of Waterloo



6th International Workshop on Formal Methods for Blockchains (FMBC 2025).
Editors: Diego Marmosler and Meng Xu



OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

■ Steering Committee

Bruno Bernardo

Nomadic Labs

Diego Marmsoler

University of Exeter

6th International Workshop on Formal Methods for Blockchains (FMBC 2025).
Editors: Diego Marmosler and Meng Xu



OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

■ Program Committee

Massimo Bartoletti
University of Cagliari

Bernhard Beckert
Karlsruhe Institute of Technology

Franck Cassez
Movement Labs

Denisa Diaconescu
University of Bucharest

Maurice Herlihy
Brown University

Sebastian Holler
Max Planck Institute for Security and
Privacy

Enrico Lipparini
University of Genoa

Fan Long
University of Toronto

Orestis Melkonian
Input Output (IOG)

Baoluo Meng
GE Aerospace Research

Burcu Kulahcioglu Ozkan
Delft University of Technology

Gordon Pace
University of Malta

Vincent Rahli
University of Birmingham

Sophie Rain
TU Wien

Augusto Sampaio
Federal university of Pernambuco

Derek Sorensen
Certora

Bas Spitters
Aarhus University

Meng Sun
Peking University

Mark Utting
The University of Queensland

Adele Veschetti
TU Darmstadt

Christoph Weidenbach
Max Planck Institute for Informatics

Teng Zhang
Aptos Labs

6th International Workshop on Formal Methods for Blockchains (FMBC 2025).
Editors: Diego Marmosler and Meng Xu



OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

■ Supporting Reviewers

Sarat Chandra Varanasi

Terru Stübinger

Is Formal Verification Practical?

Wolfgang Grieskamp ✉🏠

Aptos Labs, Palo Alto, CA, USA

Abstract

There is arguably no other domain in which the prospect of formal verification is as promising as in the area of smart contracts. Smart contracts are small- to medium-sized programs which are strongly isolated from external components. They handle high-value digital assets, and correctness is of utmost importance. Many large-impact bugs and exploits have been documented over the years. Bug bounty programs for contracts are common in the industry, offering awards in the millions of dollars, making the cost of bugs exorbitant, and the motivation for better solutions high. *Move* is a newer smart contract language used by multiple blockchains and has been designed at Meta from the ground up with formal verification in mind, featuring an integrated specification language and semantics well suited for formal reasoning. This opportunity resulted in the *Move Prover* [1], a tool that has been successfully used in the exhaustive formal verification of Diem at Meta, as well as for the frameworks of the Aptos protocol [2]. However, even though extensive work has been invested in the Move Prover to make it feasible for regular developers, most applications of the prover required specifically skilled engineers and tedious detail work to succeed, preventing it from going mainstream. In this talk, we explore the state-of-the-art of formal verification for Move. We identify the challenges that users face when trying to apply formal verification and point out future research directions to improve the expressiveness and usability of the Move Prover.

2012 ACM Subject Classification Software and its engineering → Formal software verification

Keywords and phrases Formal verification, smart contracts, Move

Digital Object Identifier 10.4230/OASICS.FMBC.2025.1

Category Invited Talk

Bio

Dr. Wolfgang Grieskamp is the head of the Move platform team at Aptos Labs, a startup incubated out of the previous work on blockchain technology at Meta between 2018 and 2021. At Aptos, Wolfgang helps operate the Aptos blockchain, a high-tps, low-latency layer 1 blockchain, which went to mainnet in 2022. He leads the work on the Move language, compiler, virtual machine, developer tooling, and formal verification. Before joining Aptos, Wolfgang was part of the Diem team (formerly Libra) at Meta, where he developed the Move Prover together with a team of fellow researchers. Before joining Meta, Wolfgang worked for a decade at Google on multiple projects in cloud computing and AI. Before Google, Wolfgang spent most of the 2000s at Microsoft Research, where he developed specification languages and related tools. Wolfgang obtained a PhD in 1999 from the Technical University of Berlin in the area of programming languages and systems. His publication record includes over 40 peer-reviewed articles at conferences and in journals.

References

- 1 David Dill, Wolfgang Grieskamp, Junkil Park, Shaz Qadeer, Meng Xu, and Emma Zhong. Fast and reliable formal verification of smart contracts with the move prover. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 183–200, Cham, 2022. Springer International Publishing.
- 2 Junkil Park, Teng Zhang, Wolfgang Grieskamp, Meng Xu, Gerardo Di Giacomo, Kundu Chen, Yi Lu, and Robert Chen. Securing Aptos Framework with Formal Verification. In Bruno



© Wolfgang Grieskamp;

licensed under Creative Commons License CC-BY 4.0

6th International Workshop on Formal Methods for Blockchains (FMBC 2025).

Editors: Diego Marmosoler and Meng Xu; Article No. 1; pp. 1:1–1:2

OpenAccess Series in Informatics



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

Bernardo and Diego Marmosler, editors, *5th International Workshop on Formal Methods for Blockchains (FMBC 2024)*, volume 118 of *Open Access Series in Informatics (OASICS)*, pages 9:1–9:16, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASICS.FMBC.2024.9.

Bringing the Power of Interactive Theorem Proving to Web3

Julian Sutherland ✉🏠

Nethermind, London, UK

Abstract

The security of the web3 ecosystem relies on the correctness of implementations of advanced mathematical formalisms, such as those underpinning various DeFi products or zk proof systems. The complexity of these formalisms, however, makes automated reasoning about said correctness challenging, if not intractable. In this talk, we give an overview of how some of these challenges can be tackled successfully in the world of interactive theorem proving. In particular, we focus on what it means to build an infrastructure for scalable reasoning about correctness of zk circuits, cryptographic algorithms, protocol models, and EVM/Yul-based smart contracts in the Lean proof assistant. Along the way, we give examples of how Nethermind was able to leverage such infrastructure in a number of real-world engagements.

2012 ACM Subject Classification Software and its engineering → Formal software verification

Keywords and phrases Formal verification, Interactive Theorem Proving, web3

Digital Object Identifier 10.4230/OASICS.FMBC.2025.2

Category Invited Talk

Bio

Julian completed his PhD in concurrent separation logics at Imperial College London, focusing on verifying safety, termination, and liveness properties of concurrent programs. He has been leading the Formal Verification team at Nethermind since its inception, for 3 years, during which time the team produced significant contributions towards the application of formal methods in web3, including the first formalisation of Yul in Lean, formalisation of RISC-Zero's Zirgen MLIR, verification tooling and an instrumentation of the Halo2 zkDSL to extract constraints into Lean, and the verification of zkSync's on-chain zk verifier.



© Julian Sutherland;

licensed under Creative Commons License CC-BY 4.0

6th International Workshop on Formal Methods for Blockchains (FMBC 2025).

Editors: Diego Marmosier and Meng Xu; Article No. 2; pp. 2:1–2:1

OpenAccess Series in Informatics



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

Formal Verification in Solidity and Move: Insights from a Comparative Analysis

Massimo Bartoletti  

University of Cagliari, Italy

Silvia Crafa  

University of Padova, Italy

Enrico Lipparini  

University of Cagliari, Italy

Abstract

Formal verification plays a crucial role in making smart contracts safer, being able to find bugs or to guarantee their absence, as well as checking whether the business logic is correctly implemented. For Solidity, even though there already exist several mature verification tools, the semantical quirks of the language can make verification quite hard in practice. Move, on the other hand, has been designed with security and verification in mind, and it has been accompanied since its early stages by a formal verification tool, the Move Prover. In this paper, we investigate through a comparative analysis: 1) how the different designs of the two contract languages impact verification, and 2) what is the state-of-the-art of verification tools for the two languages, and how do they compare on three paradigmatic use cases. Our investigation is supported by an open dataset of verification tasks performed in Certora and in the Aptos Move Prover.

2012 ACM Subject Classification Software and its engineering → Formal software verification

Keywords and phrases Smart contracts, Solidity, Move, Verification, Blockchain

Digital Object Identifier 10.4230/OASICS.FMBC.2025.3

Supplementary Material *Dataset:* <https://github.com/blockchain-unica/solidity-vs-move-verification>, archived at `swb:1:dir:8227d479ef035f889cd97557b058dcae2d2f9bd8`

Funding *Massimo Bartoletti:* Partially supported by project SERICS (PE00000014) and PRIN 2022 DeLiCE (F53D23009130001) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU.

Silvia Crafa: Supported by the National Recovery and Resilience Plan (NRRP) Project “Securing sSoftware Platforms - SOP”, CUP H73C22000890001.

Enrico Lipparini: Supported by project PRIN 2022 DeLiCE (F53D23009130001) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU.

1 Introduction

Due to the immutability of the code after deployment and the huge amount of economic assets managed, ensuring the correctness of smart contracts is a crucial task. Attacks exploiting code vulnerabilities and wrong implementations of the business logic are estimated to have caused over \$6 billion of losses [13], creating a huge demand for safer and verifiable code.

Solidity, the most adopted smart contract language, presents semantical quirks that make contract implementation quite error-prone, and that highly complicate the verification process. In order to address this issue, several bug-detection tools have been developed [34, 41], as well as some verification tools, that vary in scope, specification language, and level of abstraction. Most notably, SolCMC [1], shipped with the Solidity compiler, and the Certora Prover [15], developed for auditing.



© Massimo Bartoletti, Silvia Crafa, and Enrico Lipparini;
licensed under Creative Commons License CC-BY 4.0

6th International Workshop on Formal Methods for Blockchains (FMBC 2025).

Editors: Diego Marmosoler and Meng Xu; Article No. 3; pp. 3:1–3:18

OpenAccess Series in Informatics



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

Move is a more recent smart contract language, originally developed for the Diem/Libra blockchain and later adopted by Aptos, SUI and IOTA. Designed with verifiability in mind, Move has been accompanied by a formal verification tool [42] since its early development.

In this work, we investigate how differences in the design of Solidity and Move (in the Aptos dialect) affect verifiability. We base our study on a comparative analysis of a small set of paradigmatic use cases, each evaluated against a range of representative properties. These properties span from low-level aspects, such as function specifications and state invariants, to more high-level ones that characterize the business logic of the contract. For each property, we study the ground truth in Solidity and in Move, and we write, whenever possible, the corresponding formal specifications in the Certora Verification Language and in the Move Specification Language. We focus, in particular, on properties that exhibit discrepancies in ground truths, expressibility, or verifiability. The results of our analysis offer relevant insights about the following research questions:

- RQ1)** What is the impact of different features of Solidity and Move on verification?
- RQ2)** What is the state-of-the-art of verification tools for Solidity and Move, and which kind of properties are they currently able to verify?

As an additional contribution, we have developed a **public dataset**¹ (the first of this kind) that serves as a basis of an experimental – and extensible – comparison between the Certora and the Aptos Move specification languages and verifiers.

Structure. The paper starts in Section 2 with an overview of Solidity and Move, and their respective verification tools. Section 3 presents our methodology and discusses the choices of tools, use cases and properties. In Section 4 we present the results of our comparative analysis, addressing RQ1 and RQ2 in Section 4.1 and in Section 4.2, respectively. Finally, in Section 5 we summarize our findings, discuss limitations, and outline future work.

2 Background

In this section we overview the main features of the two languages and of the two verification tools considered in our experimental comparison. In particular, we focus on key design choices of the underlying intended blockchain’s model, since it has an impact on both the specification and the verification of contracts’ properties.

2.1 Contract languages: Solidity vs. Move

From the perspective of smart contract programming, a blockchain is best understood as an asset-exchange state machine, in which the state keeps track of the assets owned by each account, and every transaction contributes to a state transition, possibly creating new assets or exchanging assets among accounts. In Solidity, there are two kinds of accounts: externally owned accounts (EOAs) and contract accounts. The state of the asset-exchange machine can be seen as a map that associates each EOA with a balance of native assets owned by the account (e.g., ETH in Ethereum), and each contract account with a balance and a *storage*, which contains variables and data structures that define the contract state. Differently, in Move the state of the machine can be seen as a map from accounts to the assets owned by them. Assets (called *resources* in Move) are encoded by struct datatypes that enjoy linear

¹ <https://github.com/blockchain-unica/solidity-vs-move-verification>

■ **Listing 1** Simplified Solidity code for the Bank case study

```
contract Bank {
    mapping (address => uint) credits;
    function deposit() payable { credits[msg.sender] += msg.value; }
    function withdraw(uint amount) {
        credits[msg.sender] -= amount; payable(msg.sender).transfer(amount);
    }
}
```

■ **Listing 2** Simplified Move code for the Bank case study

```
module bank {
    struct Bank { credits : SimpleMap<address, Coin> } // resource definition
    fun init(account : &signer) {
        let bank = Bank { credits : simple_map::new() }; // create a resource
        move_to(account, bank); // now signer owns a bank
    }
    fun deposit(sender : &signer, owner : address, amount : u64) {
        let bank = borrow_global_mut<Bank>(owner); // borrow the resource
        let to_deposit = coin::withdraw(sender, amount); // get sender's coins
        let credit = map::borrow_mut(&mut bank.credits, address_of(sender));
        coin::merge(credit, to_deposit); // increase credit by merging coins
    }
    fun withdraw(sender : &signer, owner : address, amount : u64) { ... }
}
```

semantics, i.e., a static type system ensures that resources are never duplicated or lost. The main difference between Solidity and Move is the representation and accounting of assets. In particular, accounts in Solidity can only explicitly own native assets, while in Move they can own arbitrary resources. This has a relevant impact, since most real-world contracts involve the creation and exchange of *user-defined* assets, e.g. to represent utilities or market shares in DeFi protocols. Representing and handling user-defined assets in Solidity requires a suitable encoding in the smart contract, while in Move all assets are dealt uniformly.

To exemplify, consider the simplified Solidity code in Listing 1, which encodes a simple bank contract. Once deployed on the blockchain, the global store keeps track of the bank's *balance*, i.e. the amount of ETH associated to the contract account. The contract stores in a variable the *credits* (a *number that represents* ETHs) associated to each bank's client. When an account (*msg.sender*) invokes the function *deposit* sending a given amount (*msg.value*) of ETH, the effect is twofold: the ETHs are transferred to the contract's balance, and the sender's additional credit is registered. The *withdraw* function first decreases the number of credits and then transfers the amount of ETH to the sender. The corresponding Move code is shown in Listing 2. The code defines a *bank* module, which relies on two types of *resources*: the *Coins* provided by the underlying platform, and the user-defined *Bank* data structure. The contract is initialized by the function *init*: the signer of the transaction initializes a new, empty, *Bank* and registers its ownership in the global store. When an account invokes the function *deposit*, he takes the role of the *sender* (i.e., the transaction's signer), and the function code executes three steps: (i) the *Bank* resource is borrowed from the address of its *owner*, (ii) an *amount* of *Coins* are borrowed from those owned by the *sender*, and (iii) they are merged with those already registered in the *Bank's credits* map.

Despite its simplicity, this example already shows some differences between the two languages in terms of asset management. In Move, resources are first-class citizens, with properties such as linearity statically guaranteed by the type system. By contrast, in Solidity the user-defined assets must be carefully handled at the contract level (e.g., the logic of the `credits` map must correctly match the flow of ETH), which may be a source of critical bugs.

We will discuss other features and differences between Solidity and Move in Section 4, where they will be instrumental in addressing RQ1 about the impact of language design on smart contract verification.

2.2 Formal verification tools for Solidity and Move

For Solidity, there exist several bug detection tools [18] as well as several verification tools [34]. The two main verification tools are SolCMC [1], shipped with the Solidity compiler, and the Certora Prover [15]. Other verification tools, including SmartACE [37], SmartPulse [31], Solvent [7], VeriSolid [22], VerX [27], and Zeus [16], target various verification aspects, each tool having its own specification language, level of abstraction, and limitations. In this work, we focus on Certora (see Section 3), whose verification language (CVL) [9] features two ways of expressing contract properties: *invariants*, which represent conditions that must remain true across contract transitions, and *rules*, which are a flexible way to specify more general conditions on possible contract transitions. CVL rules can arbitrarily combine requirements on the contract state, calls to contract functions (possibly, leaving some call parameters partially specified), and assertions on the states reached upon these calls.

The Move language, since its early stages, has been tightly coupled and integrated with the Move Prover (MVP): they have been developed and maintained together, and the MVP is intended to be used routinely during smart contract development, likely to an advanced type checker. The Move Prover specification language (MSL) [4, 38] features different ways of expressing properties: function specifications (in terms of pre- and post- conditions), and invariants on functions, on struct datatypes, on global states, and on state transitions. Besides bug detection tools [30], we are only aware of another tool that addresses formal verification of Move contracts, VeriMove [21], built upon the Solidity counter-part VeriSolid.

As an example, consider the property “*after a successful deposit, the credits of the sender are increased exactly by the amount of tokens deposited*”. In CVL (Listing 3), the rule first specifies the call environment `e` (which includes the transaction parameters `msg.sender` and `msg.value`), and stores the sender’s credit before the call to `deposit()` in the variable `old_value`. Then, the rule calls `deposit()` with environment `e`, and checks whether the sender’s credits after the call have been increased by the amount sent. In MSL (Listing 4), the property is expressed as a function spec (i.e., a specification targeting a single function), in terms of pre and post conditions. The variables tagged with `post` refer to the values of the expressions *after* the call to `deposit`.

■ Listing 3 Specification of `bank/deposit-assets-credit` in CVL

```
rule deposit_assets_credit {
  env e; // environment variables of the call
  address addr_sender = e.msg.sender; // transaction sender
  mathint amount = e.msg.value; // amount of ETH tokens sent by sender to contract
  mathint old_value = currentContract.credits[addr_sender];
  deposit(e); // perform a successful call to deposit
  mathint new_value = currentContract.credits[addr_sender];
  assert new_value == old_value + amount; // verification condition
}
```

■ **Listing 4** Specification of `bank/deposit-assets-credit` in MSL (simplified)

```
spec bank_addr::bank {
  spec deposit {
    let addr_sender = signer::address_of(sender);
    let old_credits = global<Bank>(owner).credits;
    let old_value = simple_map::spec_get(old_credits, addr_sender).value;
    let post new_credits = global<Bank>(owner).credits;
    let post new_value = simple_map::spec_get(new_credits, addr_sender).value;
    ensures new_value == old_value + amount; // verification condition
  }
}
```

3 Methodology

We now detail the methodology we adopted for our comparative analysis, explaining the choices of the verification tools, use cases, and properties, and how we have built our dataset.

3.1 Verification tools

Given the variety of verification tools available, particularly for Solidity, doing an extensive comparison of all these tools lies beyond the scope of this work. We focus on the Certora Prover for Solidity and the Aptos MVP for Move. The choice of the Aptos MVP is straightforward, as it is, to the best of our knowledge, the only supported version of the Prover at the time of writing, which has furthermore been used to formally verify large Move libraries, including the entire Aptos smart contract layer [26]. We exclude VeriMove [21] as it only supports a strict subset of the language. For Solidity, the variety of available tools is broader. While no single tool strictly outperforms all others in every aspect, we choose the Certora Prover since it is the tool most used in real-world settings for the verification of complex properties. We will nonetheless explicitly mention other tools capable of addressing properties beyond the scope of the two selected tools, whenever applicable. In the following, we will refer to the two tools just as Certora and Move Prover (or MVP). We remark that Certora and MVP have been designed with different goals. In Move, specification and development go side-by-side. Certora, on the other hand, is more oriented to the ex-post analysis of contracts and is primarily used for auditing [11]: consequently, CVL is designed to support the verification of complex properties without requiring modifications to the contract code (e.g., updating ghost variables at given program points). Despite these differences, applying the state-of-the-art tools to a common benchmark is crucial to answer our research questions, namely which properties can be verified in the two languages at the time of writing (RQ2), and how the choice of the contract language affects the quality of the verification process (RQ1).

3.2 Use cases

In the selection of the verification use cases, we identify three paradigmatic smart contracts with increasing level of complexity and exhibiting a rich spectrum of features: a `bank` contract (already described in Section 2.1), a `vault` contract, and a `price-bet` contract.

The `vault` contract implements a security mechanism to prevent an adversary who has stolen the owner's private key from stealing their tokens. Upon creation, the owner specifies its private key, a recovery key, and a wait time. The contract has the following entry points:

- `receive(amount)`, which allows anyone to deposit tokens into the contract;

- `withdraw(receiver, amount)`, which allows the owner to issue a withdraw request, specifying the receiver and the desired amount;
- `finalize()`, which allows the owner to finalize the pending withdraw after the wait time has passed since the request;
- `cancel()`, which allows the owner of the recovery key to cancel the pending withdraw request during the wait time.

The `price-bet` contract implements a bet on a future exchange rate between two tokens. To create the contract, the owner specifies: itself as the contract owner; the initial pot, which is transferred from the owner to the contract; an oracle, i.e. a contract that is queried for the exchange rate between two given tokens; a deadline; a target exchange rate, which must be reached in order for the player to win the bet. The contract has the following entry points:

- `join()`, which allows a player to join the bet. This requires the player to deposit an amount of native cryptocurrency equal to the initial pot;
- `win()`, which allows the player to withdraw the whole contract balance if the oracle exchange rate is greater than the bet rate. This action is disabled after the deadline;
- `timeout()`, which can be called by anyone after the deadline, and transfers the whole contract balance to the owner.

We implement each use case in Solidity and in Aptos Move, ensuring that these implementations remain as close as possible. The verification of these use cases requires to deal with properties featuring several aspects, such as: key-value maps, access control, time constraints, contract-to-contract calls, and transaction-ordering dependencies.

3.3 Properties

For every use cases, we consider an extensive set of properties, ranging from low-level properties that only target single contract functions, to more high-level ones that characterize the global behaviour of the contract. Our choice of properties is based on breadth and diversity (in terms of language features involved, abstraction level, temporal logic structure). Overall, we end up with 66 properties. Aiming at generality, and potentially including also properties that cannot be expressed in the considered tools, we write properties in natural language. We then encode each property, whenever possible, as CVL and MSL specifications. Often, this translation involves adding suitable low-level technical assumptions, to make the specification aligned with the spirit of the corresponding natural language property. As an example, in Move, users may have a frozen coin store that prevents them from receiving tokens; in such a case, even if the natural language property does not mention such aspect, we consider adding such low-level technical assumptions as part of the translation process. Furthermore, coherently with most verification tools, we neglect transaction fees. We then manually annotate the expected truth value in Solidity and Move. Finally, we run the provers and take note of their output. We end up, for each use case, with a sheet consisting of four main columns for each property row: two columns for the ground truths, and two for the provers results. We enrich the table with additional columns containing notes on: the class of the property, the expected truth values, the formal specifications, and the provers outputs.

4 Comparison

Based on our dataset, we now present our comparative analysis. Building upon the analysis of each property, we elaborate our findings to construct an organized knowledge that extends beyond our choice of use cases. In particular, we focus on properties where discrepancies arise

between verification in Solidity and Move. These properties serve as illustrative examples for a broader discussion of the fundamental differences in the verification of the two languages. Our observations can be grouped as follows: properties whose ground truths disagree; properties that trivially hold in one language but not in the other; properties not expressible in one or both specification languages; properties expressible but not verifiable by one or both tools. These four cases are not necessarily independent of one another, but they help to better identify the primary causes of discrepancy. In the first two groups, the discrepancy specifically depends on the contract languages, while, in the latter two, it depends more on the specification language and prover functionalities. We accordingly organise this part into two subsection: Section 4.1 focuses on the impact of the contract languages, while Section 4.2 focuses on the impact of the specification languages and on the provers functionalities.

4.1 Impact of the contract language

Resource preservation. As observed in Section 2, Move enforces asset integrity by ensuring that assets cannot be duplicated but only *moved* between owners; by contrast, Solidity – except that for native tokens (ETH) – requires the management of assets to be implemented at a contract level. For example, in the [bank](#) use case, the `credits` are rendered in Move as a map from `address` to `Coin` (that *are* the actual assets), while in Solidity they are a map from `address` to `int`. This means that the Solidity code merely *tracks* the assets deposited by each user. However, implementation bugs can lead to a mismatch between the assets controlled by the contract and the overall amount of user credits, assigning more or fewer credits than they are entitled to. This significantly impacts the specification and verification of properties. First of all, in MSL, since credits *are* assets, such properties are implied by properties that concern assets. For example, in MSL the specification of the property

[bank/deposit-assets-credit](#): “*after a successful deposit of n tokens, the credits of the sender are increased by n* ”

is exactly a sub-specification of the property

[bank/deposit-assets-transfer](#): “*after a successful deposit of n tokens, n tokens pass from the control of the sender to the control of the contract.*”

In CVL, by contrast, these two properties are disjoint, and it is possible – in the presence of bugs related to the handling of credits – for the former to hold while the latter is violated. This shows that, to cover the same set of properties, Solidity requires a greater number of specifications than Move. Moreover, in Move, certain properties concerning credits trivially hold, while, in Solidity, they may be hardly verifiable, or even unexpressible. For example:

[bank/credits-leq-balance](#): “*the assets controlled by the contract are (at least) equal to the sum of all the credits*”

trivially holds in Move, where `credits` coincide with the deposited assets, but not in Solidity, where `credits` just represent the deposited assets. In general, verifying such kind of properties is quite challenging, as they require to reason about quantities depending on an unbounded number of users.

Access control and ownership. Most smart contracts implement access control mechanisms to ensure that certain actions can only be performed by certain users under certain conditions. A typical check is that some resources can only be updated by functions called by the contract owner. Move inherently supports this kind of check: it suffices that all the functions that update the resource borrow it through a signer address. This is because, in Move, a resource

can only be referenced through the address of its resource owner. This is a security pattern in Move to reduce the risks of access control errors [3]. In Solidity, instead, resource ownership is not a native notion, so it must be encoded by the contract logic. In particular, in order to implement the check above, the contract must first record the owner address in a variable, and each sensitive function must require that the transaction sender and the owner coincide. Forgetting even a single check can lead to vulnerabilities, as in the Parity Wallet hack, where the absence of such check in a function enabled the attacker to become the owner and steal all the contract funds [24]. In our dataset this difference in behaviour can be observed, e.g., for the property

vault/finalize-revert: *“a call to `finalize()` aborts if the sender is not the owner.”*

In CVL, we need to explicitly check that the address of the sender is equal to the `owner` field, while, in MSL, since the function directly accesses the `Vault` struct owned by the sender, and the `owner` is not determined by the value of a variable but by the address that owns the resource, then the property trivially holds, being enforced by the language. Another typical check is that some addresses used by the contract (e.g., its owner) do not change throughout the contract lifespan (e.g., `vault/owner-immutable`). In Solidity, it is possible to enforce that by declaring the addresses as `immutable`. In such a case, the property is directly enforced by the Solidity compiler, without having to resort to verification. Enforcing the same check in Move is less straightforward. A method is to record the concerned addresses as fields of some struct, and then verify with the MVP that these fields are invariant.

Assets transfer. Solidity and Move render assets and their transfers differently, leading to different techniques for expressing and verifying properties related to them. In Solidity, while there is a clear dichotomy in how the native asset (i.e., ETH) and user-defined assets (e.g., ERC20 tokens) are handled, in both cases transfers are rendered as contract calls. The outcome of a contract call depends on whether the callee is an externally owned account (EOA) or a contract account. When the callee is an EOA, the transfer is guaranteed to succeed, whereas for contract accounts the effect of the call depends on the implementation of the function handling the call. For instance, assets may be returned to the caller if the call reverts, or they may be forwarded (either in full or in part) to other accounts if the function is designed to do so and has enough gas. Therefore, properties about asset transfers should either discriminate between EOAs and contract accounts, or add assumptions about the implementation of the receiver function. However, the first choice is not always viable, as detecting whether an address is an EOA or a contract account (either at contract or specification level) is possible only in limited cases [25, 8]. The second choice is problematic as well, since if the assumptions are false then the property may be violated at runtime. Unlike Solidity, Move offers linguistic primitives for transferring ownership of resources, enabling a more disciplined modelling of asset transfers. This reduces the effort required to incorporate the necessary assumptions when encoding properties. In our dataset, we have observed this, e.g., in the property

vault/finalize-not-revert: *“a `finalize()` transaction sent by the contract owner, in state `REQ`, and after the wait time has passed, does not abort”*

which holds in Move but not in Solidity, since the transfer may fail when the receiver is a contract. Furthermore, also

vault/finalize-assets-transfer: *“after a successful `finalize()`, a given amount of assets pass from the contract to the receiver”*

holds in Move but not in Solidity since, if the receiver is a contract, the assets can immediately be transferred to another address through the fallback function.

Function dispatching. Solidity features a form of dynamic dispatching, in that the compiler does not always know, for a contract-to-contract call, the code that will be executed in the callee. This poses significant challenges to verification. Indeed, to avoid unsoundness, verification tools must assume that contract-to-contract calls can execute arbitrary code, which easily leads to false negatives. In order to address the issue, Certora allows users to specify a set of possible implementations of the callee, and verify the caller against each of them [12]. This technique can require considerable effort, and does not resolve the underlying unsoundness issue. Move, on the other hand, features static dispatching, i.e. the compiler (and, consequently, the verifier) know exactly the code that will be executed in the callee. In particular, Move does not support inheritance nor any form of method redefinition. We have observed the impact of these different dispatching designs, e.g., in

price-bet/win-revert: “a `win` transaction aborts if the oracle exchange rate is smaller than the bet exchange rate.”

In Certora, verifying the property requires the user to explicitly instruct the verifier to resolve the call with a given oracle implementation: leaving that unspecified would make verification fail. In practice, many Solidity contracts are written in a way that makes it impossible to predict the actual implementations of the callees (e.g., Solidity contracts using ERC20-compatible tokens usually define only their interface).

Other features. *Immutability.* In Solidity, the `immutable` keyword allows to enforce that certain variables cannot change value throughout the whole lifespan of the contract, making certain properties (e.g., the above-mentioned `vault/owner-immutable`) enforced by the Solidity compiler. In Aptos Move, since an equivalent modifier does not seem to be available, such properties have to be explicitly verified with a prover.²

Self-destruct. In Solidity, contracts can receive native tokens at any time through the `self-destruct` method. This requires additional precautions during implementation to prevent funds from getting locked in the contract. For example, our Solidity implementation of the `bank` use case allows users to withdraw only the funds corresponding to their credits (i.e., funds that have been previously deposited). In contrast, funds received via `self-destruct` cannot be withdrawn from the contract and remain locked. This is not the case in Move, as no equivalent of the `self-destruct` method exists. For example, the property

bank/no-frozen-assets: “if the contract controls some assets, then it is always possible to transfer them to some user”

holds in Move, but not in Solidity, since the contract only allows creditor to withdraw the assets they have deposited, but does not provide any function to transfer funds received via `self-destruct`, resulting in funds getting stuck in the contract.

Necessary technical assumptions. As discussed in Section 3, the translation of properties written in natural language to formal specification often requires the addition of low-level technical assumptions. Here, we report the cases that we have observed in our experiments.

Accepting incoming transfers. As observed in the “Assets transfer” paragraph, properties concerning the transfer of assets may need further assumptions on the receiver. In Move, the only technical assumption we had to add in our dataset is that the `CoinStore` of the

² Note that, in SUI Move, it is possible to define *frozen objects* (i.e. objects that cannot be modified nor moved). It does not seem possible to define *frozen fields* of an object, though.

receiving address is not `frozen`. In Solidity, one sufficient condition that can be used when the receiver equals to the transaction sender is that the sender is an EOA. Although this could be encoded in CVL by requiring that `e.msg.sender==e.tx.origin`, the Certora prover does not use this additional assumption, leading to a false negative. This is the case, e.g., of:

`bank/withdraw-assets-transfer`: “after a successful `withdraw(amount)`, exactly `amount` units of `T` pass from the control of the contract to that of the sender”

Other conditions, such as ensuring that the receiver does not fail or does not perform further calls, do not appear to be expressible in CVL.

Coin-to-FungibleAsset. Aptos has recently introduced a “*Fungible Asset*” (FA) standard [2] that extends the *Coin* standard, enabling automatic migration from Coin to FA by default. This automatic migration can make certain properties concerning the transfer of Coins violated, since Coins are not preserved (but migrated to FA). This is the case, e.g., of:

`bank/deposit-revert`: “a transaction `deposit(amount)` aborts if `amount` is greater than the `T` balance of the transaction sender.”

In order to verify such properties, it is necessary to disable the automatic migration.

Sender is not the contract. In Solidity, it is possible that a contract calls itself. In certain cases, it may be necessary to assume that this is not the case, as, otherwise, certain properties might either not hold, or be unverifiable in practice. For example, `bank/deposit-assets-transfer` specifies that, after a successful deposit of n tokens, the balance of the sender is decreased by n . While this property is true without further assumptions (since the specific `Bank` contract cannot call itself), in Certora the verification will fail without adding the assumption that the sender is not the contract. This is because verification tools usually over-approximate the set of possible executions, thus considering also the impossible case in which the contract calls itself.

4.2 Impact of the specification language and prover functionalities

We now consider different classes of properties and discuss how (and whether) they can be expressed in the two specification languages. The organization in classes has not to be intended as a formal taxonomy, rather as a schematic way to present our findings.

Function specs. We denote by “function spec” properties that specifically target a given function. We divide these properties into “success conditions”, which characterize the conditions under which a function aborts or not, and “post-conditions”, which express properties regarding the state after the call, assuming that the call has not aborted. The Move Prover has an ad-hoc specification format for function specs. In Certora, function specs can be expressed as rules that explicitly mention the function being called, and using `requires` statements for pre-conditions, the expression `lastReverted` for checking abort conditions, and the statement `assert` for post-conditions. Listing 3 and Listing 4 presented in Section 2.2 are examples of function specs in CVL and MSL, respectively. Both tools perform well over properties of this kind in our dataset.

State invariants. We denote by “state invariants” properties of the form “for every reachable state s , it holds that $P(s)$ ”, where $P(s)$ is a property that only mentions variables in the state s . In Move, state invariants can be proved in two ways: either using a *struct invariant* spec, in case an invariant only deals with a single structure (e.g., in any state, the vault state is `IDLE` or `REQ`, i.e. `vault/state-idle-req-inter`), or, otherwise, using a *global invariant*

spec (e.g., the owner and the recovery keys are distinct, i.e. `vault/keys-distinct`). In Certora, there is a common way to write invariants. Both tools perform well over properties of this kind on our dataset.

■ **Listing 5** Specification of `vault/state-idle-req-inter` in CVL

```
invariant state_idle_req_inter()
currentContract.state == Vault.States.IDLE || currentContract.state == Vault.States.REQ;
```

■ **Listing 6** Specification of `vault/state-idle-req-inter` in MSL as struct invariant

```
spec vault_addr::vault { spec Vault { invariant (state == IDLE) || (state == REQ); } }
```

Single-transition invariants. We denote by “single-transition invariants” properties of the form “*for every reachable state s , and for every transaction T , either T aborts, or it holds that $P(s, \text{next}(s, T), T)$* ”, where $\text{next}(s, T)$ is the state after a successful execution of T in s . Note that function specs are a special case where the called function is fixed. Certora is quite flexible for the verification of such properties, and allows to express arbitrary (quantifier-free) conditions on the parameters of T . In the Move Prover, there are two different ways to express single-transition invariants, both of which are less general than Certora rules. The first way is to use *global invariant updates*. This construct, however, does not allow to make explicit mention of the parameters of the transaction T , restricting expressible properties to those of the form $P(s, \text{next}(s, T))$, where T remains implicitly universally quantified. The second way is to use a *schema* of function specs (that is, syntactic sugar to group together a set of function specs with a common body). Writing a single-transition invariant this way, however, requires to write an instance of the schema for each method, making the MSL spec significantly more verbose than in CVL. As an example, consider the property:

`bank/assets-dec-onlyif-deposit`: “*if the assets of a user A are decreased after a transaction, then that transaction must be a `deposit()` where A is the sender*”

In CVL, it is possible to succinctly express such property as follows:

■ **Listing 7** Specification of `bank/assets-dec-onlyif-deposit` in CVL

```
rule assets_dec_onlyif_deposit {
  env e; method f; calldataarg args; address a;
  require e.msg.sender != currentContract && a != currentContract;

  mathint old_a_balance = nativeBalances[a];
  f(e, args); // non-reverting call to an arbitrary function f of the Bank contract
  mathint new_a_balance = nativeBalances[a];

  assert new_a_bal < old_a_bal => // if the balance has decreased...
    (f.selector == sig:deposit().selector && e.msg.sender == a); // ...then f=deposit
}
```

In MSL, it is only be possible to specify the contrapositive, i.e. that, for every transaction that is not a `deposit()`, or for which A is not the sender, then the assets of A are not decreased. This, however, requires to write a spec for each function except `deposit()`, and one further function spec for the `deposit()`, restricted to the case of A not being the sender.

■ **Listing 8** Specification of `bank/assets-dec-onlyif-deposit` in MSL

```
spec bank_addr::bank {
  spec withdraw {
    let a = signer::address_of(sender);
    let old_a_bal = global<coin::CoinStore<AptosCoin>>(a).coin.value;
    let post_new_a_bal = global<coin::CoinStore<AptosCoin>>(a).coin.value;
    requires !features::spec_is_enabled(features::COIN_TO_FUNGIBLE_ASSET_MIGRATION);
    ensures new_a_bal >= old_a_bal;
  }
  spec deposit {
    let a = signer::address_of(sender);
    ensures forall b: address where b!=a : // b is not the sender
      global<coin::CoinStore<AptosCoin>>(b).coin.value
        >= old(global<coin::CoinStore<AptosCoin>>(b).coin.value);
  }
}
```

Note that, in the case `bank` had a greater number of functions, the size of the MSL specification would grow proportionally, while the CVL spec size would remain constant.

Multiple transition invariants. We denote by “multiple-transition invariants” properties of the form “for every reachable state s , and for every sequence of transactions $\vec{T} = T_1 \dots T_n$, either one transaction aborts, or $P(s, \text{next}(s, \vec{T}[1:1]), \dots, \text{next}(s, \vec{T}[1:n]), T_1 \dots T_n)$ holds”, where $\text{next}(s, \vec{T}[1:i])$ denotes the state after the successful execution of T_1, \dots, T_i . In CVL, it is possible to express such specifications analogously to single-transition invariants, by subsequent function calls in the same rule. In MSL, this kind of specifications does not seem to be expressible. For example, consider the property:

`vault/finalize-or-cancel-twice-revert`: “a `finalize()` or a `cancel()` transaction aborts if performed immediately after another `finalize()` or `cancel()` transaction.”

This is not expressible in MSL, while Certora can verify the following CVL spec:

■ **Listing 9** Specification of `vault/finalize-or-cancel-twice-revert` in CVL

```
rule finalize_or_cancel_twice_revert {
  env e1, e2; bool b1, b2; // environments and selectors for transactions tx1, tx2
  if (b1) { finalize(e1); } else { cancel(e1); } // tx1 performs finalize or cancel
  if (b2) { finalize@withrevert(e2); } else { cancel@withrevert(e2); } // same for tx2
  assert lastReverted; // checks that the 2nd tx is always reverted
}
```

Metamorphic properties. These are properties that involve multiple finite sequences of transactions [14]. A typical class of metamorphic property are *additivity properties*: e.g.:

`bank/deposit-additivity`: “two successful `deposit()` of n_1 and n_2 units of token T performed by the same sender are equivalent to a single `deposit()` of $n_1 + n_2$ units of T ”

In CVL, it is possible to express some metamorphic properties through the use of `storage` types, which allow to record the contract storage at different points of execution and to later compare them (see, e.g. Listing 10). This feature is not present in MSL, so metamorphic properties do not seem expressible.

■ **Listing 10** Specification of `bank/deposit-additivity` in CVL (simplified)

```
using Bank as c;
rule withdraw_additivity {
  env e1, e2, e3; // environments for transactions tx1,tx2,tx3
  uint v1, v2, v3; // values sent along with transactions tx1,tx2,tx3
  storage initial = lastStorage; // save the current storage in variable initial

  require e1.msg.sender == e2.msg.sender; // the senders of tx1,tx2 must be equal
  require v1+v2 <= currentContract.opLimit;
  withdraw(e1,v1); withdraw(e2,v2); // perform tx1,tx2 in sequence
  storage s12 = lastStorage; // saves the current storage in variable s12

  require e3.msg.sender == e1.msg.sender; // the sender of tx3 is the same as tx1,tx2
  require v3 == v1+v2; // the amount of tx3 must be the sum of the amounts in tx1,tx2
  withdraw(e3,v3) at initial;
  storage s3 = lastStorage; // saves the current storage in variable s3

  assert s12[c] == s3[c]; // checks that tx1;tx2 have the same effect of tx3
}
```

Other properties. Some classes of properties do not seem expressible in any of the two tools. Without claiming exhaustivity, we now briefly discuss some of the classes we have encountered, with particular attention to those that seem addressable by other tools.

Liveness. Liveness properties have the form “*eventually a state that satisfies certain conditions is reached*”. In `price-bet`, a desirable liveness property is

`price-bet/eventually-balance-zero`: “*eventually the contract balance goes to 0*”

Note that this property is closely related to, but more abstract than, the property

`price-bet/timeout-not-revert`: “*a transaction `timeout()` [which transfers the assets controlled by the contract to the owner] does not revert if the deadline has passed*”

Tools able to handle such kind of properties, usually under the assumption of fairness conditions (in the example, that the `timeout()` function is called at least once after the deadline), are VeriSolid [19], VeriMove [21], and SmartPulse [31].

Liquidity/Enabledness. Liquidity [7] or Enabledness [29] properties are of the form “*in every reachable state, certain users are always able to fire a (fixed) number of transactions to reach a desirable state*”. In `bank`, an example of such properties is

`bank/no-frozen-credits`: “*if the credits are strictly positive, it is possible to reduce them*”

Note that this kind of property never mentions the function that should be called nor its parameters, as they are existentially quantified and determining them (as a function of the current state) is a task of the tool. A tool that addresses such kind of properties is Solvent [7].

CTL fragment: The specification language of VeriSolid (and, consequently, of VeriMove) covers an expressive fragment of Computational Tree Logic (CTL). Such expressivity comes at the expenses of soundness, as the verification process relies on a certain level of abstraction. Examples of CTL specifications include the Liveness seen before, as well as properties of the form “*P₁ cannot happen after P₂*”, or “*If P₁ happens, then P₂ can only happen after P₃ happens*”. These properties cannot be expressed in CVL, since it is not possible to talk about unbounded sequences of method calls, but only about sequences of states of finite length. E.g., a property not expressible in CVL but in the CTL fragment supported by VeriSolid is:

`vault/finalize-after-withdraw-not-revert`: “*after a successful `withdraw()`, if no `cancel()` or `finalize()` have been called successfully, then `finalize()` does not abort*”

All these properties have a higher level of abstraction than those discussed in the previous paragraphs. Although some of these properties, in certain cases, can be reformulated in terms of more concrete properties that imply them, doing so requires a more advanced knowledge of the low-level aspects, and reduces their generality. It has been observed that properties that abstract the system have a better return-on-investment than low-level properties [38].

Orthogonal features of properties. We finally address specific features of properties that can appear in all previous classes, hence for which a separate discussion is needed.

Inter vs. Intra function invariants Invariants can be of two kinds: those that must be preserved across function calls (*inter-function* invariants) and those that must be preserved within the execution of a function (*intra-function* invariants). In the latter, the notion of *reachable state* is extended to intermediate states. In some cases, intra-function invariants give stronger security guarantees. For example, consider the invariant

`vault/keys-invariant-inter`: “the receiver key cannot be changed after initialization”

Requiring this invariant to only hold inter-function is not enough, as it does not capture attacks where an adversary (i) changes the `receiver` key within `finalize()` before the transfer, (ii) sends the contract tokens to her address, and (iii) restores the key to the original value before the end of the function. It is necessary to require the invariant to hold also intra-function (`vault/keys-invariant-intra`). In Certora, verification of intra-function invariants is possible through ghost variables and hooks [10]. In Move, on the contrary, verification of intra-function invariants is, in general, not possible. The MVP can check that an invariant holds *globally*, i.e. every time the global state is updated [4], but this cannot capture every change that occurs during the execution of a function. In the example attack mentioned above, the MVP is not able to detect that the `receiver` key is changed within the execution of the `finalize()`.

Nested quantifiers. Several interesting properties require the nesting of quantifiers. In Certora, quantifier nesting is limited to *exists-forall* fragments, whereas *forall-exists* fragments are disallowed. This makes not expressible in CVL properties such as:

`bank/exists-at-least-one-credit-change`: “after a successful transaction, the credits of at least one account have changed”

Although MSL allows arbitrary combinations of quantifiers, in practice the verification of such properties can be problematic, as the underlying SMT solvers often struggle with quantifiers. In our experiments, we managed to successfully verify the previous property, but got an inconsistent result in the case of `bank/exists-unique-asset-change`. This inconsistency may be caused by the version of the underlying SMT solver used.

Gas. As discussed in “Assets transfer” in Section 4.1, the truth of certain properties may depend on the amount of gas available to the involved functions. For instance, in Solidity the ground truths of `bank/withdraw-assets-transfer` and `vault/finalize-assets-transfer` differ because of the functions used in the respective contracts to transfer ETH from the contract to another address: in the implementation of `bank`, we are using `transfer`, which do not carry enough gas to perform further calls, while in `vault` we are using `call`, which instead transfers all the gas to the callee. Certora however over-approximates the amount of gas available, so it gives a false negative for `bank/withdraw-assets-transfer`.

5 Conclusions

The empirical analysis of our study validates the folklore knowledge that Move is better suited for verification than Solidity. In particular, Move’s resource-orientation facilitates the verification of properties concerning, e.g., resource preservation, ownership, and transferring of assets. The only weak spot we have observed in (Aptos) Move is the lack of a construct to enforce the immutability of contract variables – a feature that instead is present in Solidity. We have noted that, in order to properly specify certain properties and determine their truth, some low-level aspects of the underlying contract layers must be taken into account. While this could be discouraging for smart contract developers unfamiliar with these low-level details, it can also serve as an incentive to deepen their understanding on these aspects, ultimately leading to more secure smart contract implementations.

Concerning verification tools, we have observed that the Certora Prover can express a broader set of properties than the Move Prover, e.g., transition invariants involving multiple transactions, metamorphic properties, and intra-function invariants. We believe that all the functionalities needed to verify such properties could be smoothly added to the Move Prover, as well. We have also noted that there are several relevant classes of properties that are out of the scope of both tools (e.g., liveness, liquidity/enabledness, and, more generally, other complex temporal properties concerning the business logic of the contract). We have observed that some of these properties can be addressed by other tools, although their current maturity level remains below that of the Certora and Move provers.

We have contributed with an open dataset of smart contract implementations and verification tasks performed in the two tools (the first of this kind), that we envision will further encourage research on formal verification of Solidity and Move.

Limitations. Although our empirical analysis is based on a set of 66 verification tasks covering a broad range of properties, we expect that extending our dataset would highlight additional differences between verification in Solidity and Move. Moreover, it could reveal some further kinds of properties that would be desirable to verify on real-world smart contracts but currently fall beyond the scope of existing verification tools. This could be the case, e.g., of economic properties of DeFi protocols, whose verification currently requires either using weaker analysis techniques than formal verification (e.g., property-based testing [20], statistical model checking [6]), taint analysis [35, 17], type systems [39, 40], attack synthesis [36] or abstracting from actual contract code [33, 32, 5, 23, 28].

References

- 1 Leonardo Alt, Martin Blicha, Antti E. J. Hyvärinen, and Natasha Sharygina. Solcmc: Solidity compiler’s model checker. In *Computer Aided Verification (CAV)*, volume 13371 of *LNCS*, pages 325–338. Springer, 2022. doi:10.1007/978-3-031-13185-1_16.
- 2 Aptos. Aptos fungible asset (FA) standard. <https://aptos.dev/en/build/smart-contracts/fungible-asset>, 2025.
- 3 Aptos. Move security guidelines. <https://aptos.dev/en/build/smart-contracts/move-security-guidelines>, 2025.
- 4 Aptos. Move specification language. <https://aptos.dev/en/build/smart-contracts/prover/spec-lang>, 2025.
- 5 Kushal Babel, Philip Daian, Mahimna Kelkar, and Ari Juels. Clockwork finance: Automated analysis of economic security in smart contracts. In *IEEE Symposium on Security and Privacy (SP)*, pages 2499–2516. IEEE Computer Society, 2023. doi:10.1109/SP46215.2023.10179346.

- 6 Massimo Bartoletti, James Hsin-yu Chiang, Tommi A. Junttila, Alberto Lluch-Lafuente, Massimiliano Mirelli, and Andrea Vandin. Formal analysis of Lending Pools in Decentralized Finance. In *Int. Symp. on Leveraging Applications of Formal Methods (ISoLA)*, volume 13703 of *LNCS*, pages 335–355. Springer, 2022. doi:10.1007/978-3-031-19759-8_21.
- 7 Massimo Bartoletti, Angelo Ferrando, Enrico Lipparini, and Vadim Malvone. Solvent: Liquidity verification of smart contracts. In *Integrated Formal Methods (iFM)*, pages 256–266. Springer-Verlag, 2024. doi:10.1007/978-3-031-76554-4_14.
- 8 Massimo Bartoletti, Fabio Fioravanti, Giulia Matricardi, Roberto Pettinau, and Franco Sainas. Towards benchmarking of Solidity verification tools. In *International Workshop on Formal Methods for Blockchains (FMBC)*, volume 118 of *OASICS*, pages 6:1–6:15, 2024. doi:10.4230/OASICS.FMBC.2024.6.
- 9 Certora. The Certora Verification Language. <https://docs.certora.com/en/latest/docs/cvl/index.html>, 2025.
- 10 Certora. Hooks. <https://docs.certora.com/en/latest/docs/cvl/hooks.html>, 2025.
- 11 Certora. Reports. <https://www.certora.com/reports>, 2025.
- 12 Certora. Working with multiple contracts. <https://docs.certora.com/en/latest/docs/user-guide/multicontract/index.html>, 2025.
- 13 Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Ben Livshits. Smart contract and DeFi security: Insights from tool evaluations and practitioner surveys. In *International Conference on Software Engineering (ICSE)*, pages 60:1–60:13. ACM, 2024. doi:10.1145/3597503.3623302.
- 14 Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Comput. Surv.*, 51(1):4:1–4:27, 2018. doi:10.1145/3143561.
- 15 Daniel Jackson, Chandrakana Nandi, and Mooly Sagiv. Certora technology white paper. <https://docs.certora.com/en/latest/docs/whitepaper/index.html>, 2022.
- 16 Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: analyzing safety of smart contracts. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2018. URL: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_09-1_Kalra_paper.pdf.
- 17 Queping Kong, Jiachi Chen, Yanlin Wang, Zigui Jiang, and Zibin Zheng. DeFiTainter: Detecting price manipulation vulnerabilities in DeFi protocols. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1144–1156, 2023. doi:10.1145/3597926.3598124.
- 18 Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, and Heung-No Lee. Ethereum smart contract analysis tools: A systematic review. *IEEE Access*, 10:57037–57062, 2022. doi:10.1109/ACCESS.2022.3169902.
- 19 Anastasia Mavridou, Aron Laszka, Emmanouela Stachtari, and Abhishek Dubey. VeriSolid: Correct-by-design smart contracts for Ethereum. In *Financial Cryptography and Data Security*, pages 446–465. Springer, 2019. doi:10.1007/978-3-030-32101-7_27.
- 20 Mikkel Milo, Eske Hoy Nielsen, Danil Annenkov, and Bas Spitters. Finding smart contract vulnerabilities with concert’s property-based testing framework. In *International Workshop on Formal Methods for Blockchains (FMBC)*, volume 105 of *OASICS*, pages 2:1–2:13, 2022. doi:10.4230/OASICS.FMBC.2022.2.
- 21 Keerthi Nelaturu, Eric Keilty, and Andreas Veneris. Natural language-based model-checking framework for Move smart contracts. In *Software Defined Systems (SDS)*, pages 89–94, 2023. doi:10.1109/SDS59856.2023.10328964.
- 22 Keerthi Nelaturu, Anastasia Mavridou, Emmanouela Stachtari, Andreas G. Veneris, and Aron Laszka. Correct-by-design interacting smart contracts and a systematic approach for verifying ERC20 and ERC721 contracts with VeriSolid. *IEEE Trans. Dependable Secur. Comput.*, 20(4):3110–3127, 2023. doi:10.1109/TDSC.2022.3200840.

- 23 Eske Hoy Nielsen, Danil Annenkov, and Bas Spitters. Formalising decentralised exchanges in Coq. In *ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pages 290–302. ACM, 2023. doi:10.1145/3573105.3575685.
- 24 OpenZeppelin. The Parity Wallet hack explained. <https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7>, 2017.
- 25 OpenZeppelin. Utilities / address. <https://docs.openzeppelin.com/contracts/4.x/api/utils#Address>, 2024.
- 26 Junkil Park, Teng Zhang, Wolfgang Grieskamp, Meng Xu, Gerardo Di Giacomo, Kundu Chen, Yi Lu, and Robert Chen. Securing Aptos framework with formal verification. In *International Workshop on Formal Methods for Blockchains (FMBC)*, volume 118 of *OASICS*, pages 9:1–9:16, 2024. doi:10.4230/OASICS.FMBC.2024.9.
- 27 Anton Permenev, Dimitar K. Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin T. Vechev. VerX: Safety verification of smart contracts. In *IEEE Symposium on Security and Privacy*, pages 1661–1677. IEEE, 2020. doi:10.1109/SP40000.2020.00024.
- 28 Daniele Pusceddu and Massimo Bartoletti. Formalizing Automated Market Makers in the Lean 4 Theorem Prover. In *International Workshop on Formal Methods for Blockchains (FMBC)*, volume 118 of *OASICS*, pages 5:1–5:13, 2024. doi:10.4230/OASICS.FMBC.2024.5.
- 29 Jonas Schiffl and Bernhard Beckert. A practical notion of liveness in smart contract applications. In *International Workshop on Formal Methods for Blockchains (FMBC)*, volume 118 of *OASICS*, pages 8:1–8:13, 2024. doi:10.4230/OASICS.FMBC.2024.8.
- 30 Shuwei Song, Jiachi Chen, Ting Chen, Xiapu Luo, Teng Li, Wenwu Yang, Leqing Wang, Weijie Zhang, Feng Luo, Zheyuan He, Yi Lu, and Pan Li. Empirical study of Move smart contract security: Introducing MoveScan for enhanced analysis. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 1682–1694. ACM, 2024. doi:10.1145/3650212.3680391.
- 31 Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu K. Lahiri, and Isil Dillig. Smart-Pulse: Automated checking of temporal properties in smart contracts. In *IEEE Symposium on Security and Privacy (SP)*, pages 555–571. IEEE, 2021. doi:10.1109/SP40001.2021.00085.
- 32 Xinyuan Sun, Shaokai Lin, Vilhelm Sjöberg, and Jay Jie. How to exploit a DeFi project. In *Workshop on Trusted Smart Contracts*, volume 12676 of *LNCS*, pages 162–167. Springer, 2021. doi:10.1007/978-3-662-63958-0_14.
- 33 Palina Tolmach, Yi Li, Shang-Wei Lin, and Yang Liu. Formal analysis of composable DeFi protocols. In *Workshop on Trusted Smart Contracts*, volume 12676 of *LNCS*, pages 149–161. Springer, 2021. doi:10.1007/978-3-662-63958-0_13.
- 34 Palina Tolmach, Yi Li, Shangwei Lin, Yang Liu, and Zengxiang Li. A survey of smart contract formal specification and verification. *ACM Comput. Surv.*, 54(7):148:1–148:38, 2022. doi:10.1145/3464421.
- 35 Shuai Wang, Chengyu Zhang, and Zhendong Su. Detecting nondeterministic payment bugs in Ethereum smart contracts. *Proc. ACM Program. Lang.*, 3(OOPSLA):189:1–189:29, 2019. doi:10.1145/3360615.
- 36 Hongbo Wen, Hanzhi Liu, Jiaxin Song, Yanju Chen, Wenbo Guo, and Yu Feng. FORAY: towards effective attack synthesis against deep logical vulnerabilities in defi protocols. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1001–1015. ACM, 2024. doi:10.1145/3658644.3690293.
- 37 Scott Wesley, Maria Christakis, Jorge A. Navas, Richard J. Treffer, Valentin Wüstholtz, and Arie Gurfinkel. Verifying Solidity smart contracts via communication abstraction in SmartACE. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 13182 of *LNCS*, pages 425–449. Springer, 2022. doi:10.1007/978-3-030-94583-1_21.
- 38 Meng Xu. Research report: Not all Move specifications are created equal : A case study on the formally verified Diem Payment Network. In *Workshop on Language-Theoretic Security (LangSec)*, pages 200–214. IEEE, 2024. doi:10.1109/SPW63631.2024.00024.

- 39 Siqui Yao, Haobin Ni, Andrew C. Myers, and Ethan Cecchetti. SCIF: A language for compositional smart contract security. *CoRR*, abs/2407.01204, 2024. doi:10.48550/arXiv.2407.01204.
- 40 Brian Zhang. Towards finding accounting errors in smart contracts. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 138:1–138:13. ACM, 2024. doi:10.1145/3597503.3639128.
- 41 Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. Demystifying exploitable bugs in smart contracts. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 615–627. IEEE, 2023. doi:10.1109/ICSE48619.2023.00061.
- 42 Jingyi Emma Zhong, Kevin Cheang, Shaz Qadeer, Wolfgang Grieskamp, Sam Blackshear, Junkil Park, Yoni Zohar, Clark W. Barrett, and David L. Dill. The Move Prover. In *Computer Aided Verification (CAV)*, volume 12224 of *LNCS*, pages 137–150. Springer, 2020. doi:10.1007/978-3-030-53288-8_7.

ByteSpector: A Verifying Disassembler for EVM Bytecode

Franck Cassez 

Movement Labs, San Francisco, CA, USA

Abstract

We present **ByteSpector**, a tool for constructing and verifying control flow graphs (CFGs) from Ethereum Virtual Machine (EVM) bytecode. CFGs play a crucial role in analyzing smart contract behavior, but resolving dynamic jumps and ensuring CFG correctness remain significant challenges. **ByteSpector** addresses these challenges by generating formally verified CFGs, i.e., all target jumps have been resolved correctly, which can serve as a foundation for further contract verification.

ByteSpector introduces several key innovations. First, **ByteSpector** features an efficient algorithm for resolving dynamic jumps that uses a combination of abstract interpretation and semantics reasoning. Second **ByteSpector** can automatically generate *proof objects* from EVM bytecode. Proof objects are **Dafny** programs that encode the semantics of the bytecode, and can be used to prove that computed CFGs over-approximate the contracts execution paths. Third, **ByteSpector** is written in **Dafny** and is guaranteed to be free of common runtime errors like array-out-of-bounds, division-by-zero etc. Moreover, the code and libraries can be automatically translated into multiple languages (e.g., C#, Python, Java, JavaScript), making them reusable in broader verification frameworks.

By generating **Dafny** proof objects (and verified CFGs), **ByteSpector** provides a robust foundation for bytecode-level analysis, enabling formal verification of smart contracts beyond high-level source code analysis.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases EVM bytecode, deductive verification, Control Flow Graph

Digital Object Identifier 10.4230/OASICS.FMBC.2025.4

1 Introduction

The Ethereum Virtual Machine (EVM) is the execution layer of Ethereum, running smart contracts, programs that power decentralized applications (dApps) and manage billions in assets, particularly in DeFi. Smart contracts are typically written in high-level languages like Solidity¹ and Vyper,² then compiled into EVM bytecode for execution. Smart contract high-level code undergoes testing and audits before deployment. However, since the EVM executes bytecode, not high-level (Solidity or Vyper source) code, verifying bytecode is crucial for several reasons:

Compiler bugs. Compilers may introduce errors, producing incorrect bytecode. Certora identified several Solidity compiler bugs, including: in 2022, a memory optimisation issue mistakenly removed operations affecting computation [12]; in 2021, a dynamic array bug caused potential memory corruption [7]. Fortunately, the previous bugs were discovered and fixed before being exploited. However, in 2023, a bug [11] in the Vyper compiler was exploited resulting in USD26 Million stolen.

EVM-specific constraints. The EVM has some specific features that are not present in source code, such as *gas* and a bounded execution *stack*. For instance, if we want

¹ <https://soliditylang.org>

² <https://vyperlang.org>



© Franck Cassez;

licensed under Creative Commons License CC-BY 4.0

6th International Workshop on Formal Methods for Blockchains (FMBC 2025).

Editors: Diego Marmosoler and Meng Xu; Article No. 4; pp. 4:1–4:15

OpenAccess Series in Informatics



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

to guarantee the absence of *stack-overflow/underflow exception*, or determine the gas requirements to avoid an *out-of-gas exception*, we need to reason about the bytecode. These safety properties can only be verified at the bytecode level.

The problem we address is enabling formal reasoning about EVM bytecode. To achieve this, we have developed **ByteSpector**, a tool that extracts **Dafny** proof objects from EVM bytecode. These proof objects can be analyzed using a **Dafny** formal semantics of the EVM, leveraging **Dafny**'s verification capabilities to reason about bytecode behavior. **ByteSpector** is not intended to replace existing tools for verifying high-level smart contract code. It does not verify that the compiler correctly translates high-level code to bytecode, but rather provides a way to reason about the bytecode itself.

Our contribution

Our contribution is a tool, **ByteSpector**, that can automatically construct *proof objects* from EVM bytecode. Proof objects are **Dafny** (a verification-friendly programming language) programs that encode the semantics of the bytecode, can be instrumented with specifications (e.g., absence of stack underflow) and reasoned about. **ByteSpector** is itself written in **Dafny** which has several advantages: the code is free of runtime errors (arithmetic, array-out-of-bounds, division-by-zero, etc.), and always terminates. Moreover, the **Dafny** code can be automatically translated to several target languages (e.g., C#, Python, Java, JavaScript) using the **Dafny** backends and re-used in other projects.

2 Overview

In this section we introduce the workflow of **ByteSpector** and a simple example to highlight the main features of the tool. Figure 1 summarises the main stages: we start with the EVM bytecode of a program, disassemble it into segments of instructions, build a control flow graph (CFG) of the program, and use the CFG to generate a **Dafny** proof object that can be reasoned about using the **Dafny-EVM**, a formal semantics [5] of the EVM written in **Dafny**.



■ **Figure 1** Disassembling steps. The EVM bytecode is split into segments. A CFG of the bytecode is built where each node is associated with a segment of code. The CFG is encoded as a **Dafny** program that can be instrumented and mechanically verified.

An EVM program is a finite sequence of bytes stored in a contract's code region starting at address `0x00`. This bytecode can be decoded into readable EVM instructions and organised into *segments* that are *non-branching instruction sequences* except possibly for the last instruction e.g., `JUMP`. For example, the bytecode³ `0x6005600d565b600b600d565b005b56` corresponds to the disassembled **TwoCalls** program in Listing 1.

To analyze execution flow, we construct a *Control Flow Graph* (CFG), where nodes represent bytecode segments. Figure 2a shows the CFG for **TwoCalls**, where Segment 3 executes twice, and the `JUMP` at line 18 has multiple possible targets depending on context. **ByteSpector** resolves these *dynamic jumps* and determines all possible execution paths.

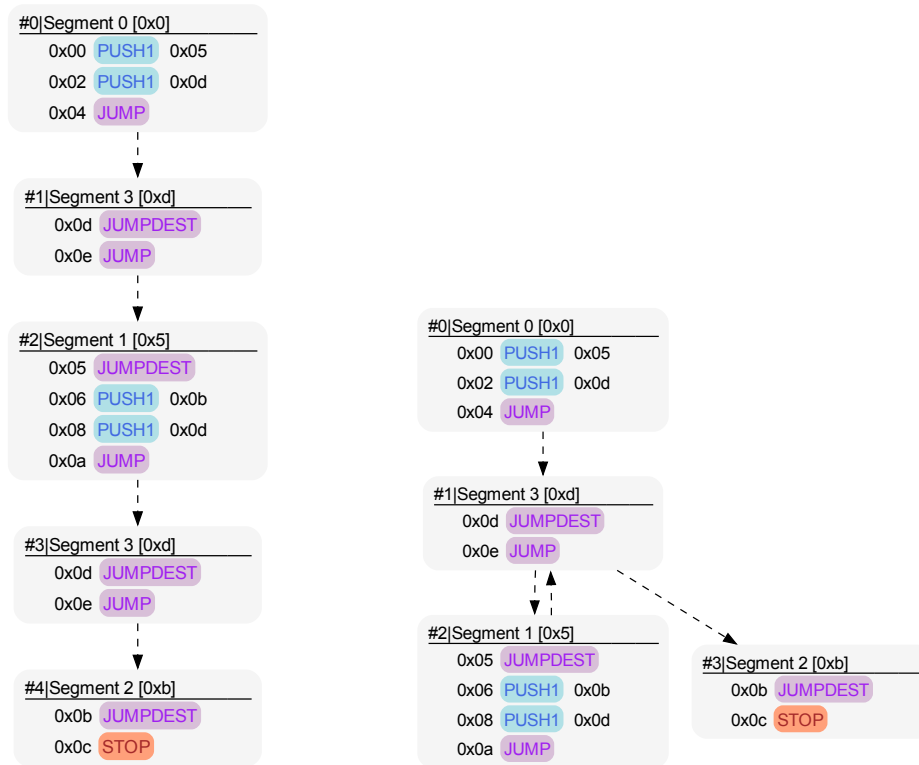
³ The prefix `0x` indicates hexadecimal encoding.

■ **Listing 1** The TwoCalls program

```

1 // Address: 0x00_01_02_03_04_05_06_07_08_09_0a_0b_0c_0d_0e
2 // Bytecode: 0x60_05_60_0d_56_5b_60_0b_60_0d_56_5b_00_5b_56
3 // PC: Instruction args* // description and stack content after execution
4 // Segment 0
5 0x00: PUSH1 0x05 // Push 1 byte [0x05]
6 0x02: PUSH1 0x0d // Push 1 byte [0x0d, 0x05]
7 0x04: JUMP       // Jump to top of stack 0x0d [0x05] (call1)
8 // Segment 1
9 0x05: JUMPDEST   // [] empty
10 0x06: PUSH1 0x0b // Push 1 byte 0x0b [0x0b]
11 0x08: PUSH1 0x0d // Push 1 byte 0x0d [0x0d, 0x0b]
12 0x0a: JUMP      // Jump to top of stack 0x0d (call2) [0x0b]
13 // Segment 2
14 0x0b: JUMPDEST   // []
15 0x0c: STOP
16 // Segment 3 -- Does do not anything except to simulate a function call/return
17 0x0d: JUMPDEST   // call1 stack is [0x05] | call2 stack is [0x0b]
18 0x0e: JUMP      // Jump (call1) to 0x05 [] | Jump (call2) to 0x0b []

```



(a) A precise CFG with 5 nodes.

(b) A coarse CFG with 4 nodes.

■ **Figure 2** Two CFGs for the TwoCalls program. The nodes of the graphs are identified by a unique number (top left of a node) #n. The initial node is #0. Each node is mapped to a segment of the bytecode of TwoCalls and the start address of the segment is shown in the node e.g., [0x5]. Each segment ends in a jump instruction and the target of the jump is shown by the dashed arrow. The coarse CFG (b) contains a cycle and many infeasible paths, whereas (a) captures the control flow more precisely.

■ Listing 2 Section of the Dafny proof object for program TwoCalls

```

28  /** Node 2. Segment Id for this node is: 1. Starting at 0x5. Type is: JUMP Segment */
29  function ExecuteFromCFGNode_s2(s0: EVMSState, gas: nat): (s': EVMSState)
30    requires s0.pc == 0x5 as nat
31    requires s0.Operands() >= 0
32    decreases gas
33  { if gas == 0 then s0
34    else
35      var s1 := JumpDest(s0);
36      var s2 := Push1(s1, 0x0b);
37      var s3 := Push1(s2, 0x0d);
38      var s4 := Jump(s3);
39      // JUMP to the target at Peek(0)
40      ExecuteFromCFGNode_s3(s4, gas - 1)
41    }
42  /** Node 3. Segment Id for this node is: 3. Starting at 0xd. Type is: JUMP Segment */
43  function ExecuteFromCFGNode_s3(s0: EVMSState, gas: nat): (s': EVMSState)
44    requires s0.pc == 0xd as nat
45    requires s0.Operands() >= 1 && s0.stack[0] == 0xb
46    decreases gas
47  { if gas == 0 then s0
48    else
49      var s1 := JumpDest(s0);
50      var s2 := Jump(s1);
51      // JUMP to the target at Peek(0)
52      ExecuteFromCFGNode_s4(s2, gas - 1)
53  }

```

The CFG is encoded as a Dafny *proof object*, a Dafny program defining bytecode semantics for each node and transitions between them. Listing 2 shows the ByteSpector-generated code for nodes #2 and #3. The Dafny code uses Dafny-EVM functions like `Jumpdest`, `Push1`, `Jump` that provide the semantics of the EVM instructions. Preconditions (lines 30–31, 44–45) restrict segment entry points, while function calls (lines 40, 52) define CFG edges. If Dafny successfully verifies the proof object, the CFG is *sound*⁴ and safely *over-approximates* the bytecode’s behavior, which is essential for proving safety properties.

3 From EVM bytecode to segments of instructions

In this section we introduce the main features of the EVM, its semantics, and how the bytecode can be partitioned into *segments*.

3.1 The Ethereum virtual machine

The EVM is a *stack* machine with a bounded stack for computations. *Memory* stores temporary data during execution, while *global storage* retains permanent contract state on the blockchain. Every computation consumes *gas*, a measure of computational cost, and results in one of the following outcomes:

- a *successful termination*, the effect of the computation is stored in the global storage.
- an *abort* (exception) e.g., caused by stack overflow, jumps to invalid targets, invalid number of arguments to an instruction, etc.
- an *out-of-gas exception* if the gas budget is exhausted before the end of the execution.

⁴ Soundness is formally defined in the next sections.

3.2 EVM bytecode

The EVM supports around 150 instructions to perform arithmetic or bitwise operations (e.g., `ADD`, `SUB`, `AND`, `OR`), *stack* operations (e.g., `PUSH0`, `PUSH1`, `POP`, `SWAP1`, `DUP1`), *memory store/load* operations e.g., `MSTORE`, `MLOAD`, *global storage* operations e.g., `SSTORE`, `SLOAD`, unconditional (resp. conditional) *jumps*, `JUMP` (resp. `JUMPI`), as well as blockchain specific operations (e.g., `BALANCE`, `ADDRESS`, `GASLIMIT`). A few instructions, such as `PUSH`es, have parameters and are encoded on a variable number of bytes. For example `PUSH2` pushes two bytes on the stack and expects two bytes as arguments e.g., `PUSH2 0x3056`. We consider only well-formed EVM bytecode with valid opcodes and correct argument counts. Such bytecode can be decoded into an instruction sequence, as shown in Listing 1 (lines 5–18).

3.3 Semantics of the EVM

The EVM semantics, as per the Yellow Paper [13] define the *state transition* function of the EVM, that maps an EVM state s to a new state s' . A state includes bytecode, program counter $s.pc$ (instruction in the program to be executed next), the stack content $s.stack$, the memory content $s.memory$, gas left for the rest of the computation $s.gas$, global storage, and several other fields that we omit here. We let $|s.stack|$ be the size of the stack, and $s.stack[i..]$ be the sequence $s.stack$ without the first i elements. Following the Dafny-EVM [5] semantics, we separate instruction semantics from gas consumption (ignored for now). For instance, if $s.pc$ points to a `POP` instruction, the next state $\text{Next}(s)$ is defined by:⁵

$$\text{POP} : \text{Next}(s) = \begin{cases} \text{if } |s.stack| < 1 \text{ then } s_{\perp}, (stack \text{ is empty}) \\ \text{else } s' \text{ where } s'.pc = s.pc + 1 \text{ and } s'.stack = s.stack[1..]. \end{cases}$$

If the stack is empty, the execution aborts with an *error state* s_{\perp} . Otherwise, the program counter is incremented by one and the top element of the stack is removed.

The *jump* instructions `JUMP` and `JUMPI` (unconditional/conditional jump) are *branching* instructions that may transfer the control location ($s.pc$) to an address that is not the next instruction in the code. However, a nice feature of the EVM is that the target of a jump instruction must be a *legal* address, i.e., the target address of the jump must be a `JUMPDEST` instruction. For instance, `TwoCalls`, Listing 1, has 3 legal target addresses: `0x05`, `0x0b`, and `0x0d`. As a result, given a program p , we can compute the set $\text{ValidJumps}(p)$ of legal target addresses in p by identifying the `JUMPDEST` instructions. If the instruction at $s.pc$ is `JUMP`, $\text{Next}(s)$ is defined by:

$$\text{JUMP} : \text{Next}(s) = \begin{cases} \text{if } |s.stack| < 1 \text{ or } s.stack[0] \notin \text{ValidJumps}(p) \text{ then } s_{\perp} \\ \text{else } s' \text{ where } s'.pc = s.stack[0] \text{ and } s'.stack = s.stack[1..]. \end{cases}$$

For given bytecode p and input i , the EVM semantics uniquely define $\text{run}(p, i)$, a finite sequence of EVM states (termination is ensured by the gas limit). The error state s_{\perp} has no successors. The set of all possible executions of p is: $\text{Runs}(p) = \{\text{run}(p, i) \mid i \in \mathcal{I}\}$ where \mathcal{I} is the finite set of inputs (the calldata has a maximal size in the EVM).

⁵ For simplicity, we only define the PC and stack updates.

3.4 Code segments

A segment is a sequence of non-branching consecutive instructions ending at:

- branching instructions `JUMP`, `JUMPI`, or
- terminal instructions `STOP`, `REVERT`, `RETURN`, or
- any instruction before a `JUMPDEST`.

This ensures that *i*) jump targets start segments, preventing jumps into the middle of a segment and *ii*) there is a unique (linear) execution flow within each segment. Segments are shown in Listing 1 and the CFG of `TwoCalls` in Figure 2a. We extend `Next` to segments to compute the state `s : Next(s)` after executing all instructions in a segment `s`, returning an error state if execution fails before completion.

4 Control flow graphs for EVM bytecode

ByteSpector builds an intermediate artefact, a *control flow graph* (CFG) for the EVM bytecode, before extracting the proof object from the CFG. A program's Control Flow Graph (CFG) represents how the program counter `s.pc` updates during *program execution*. In this section we define CFGs and their expected properties and show how to build a CFG for EVM bytecode.

4.1 Control flow graph

A CFG g is a *directed graph* with *nodes* and *edges* and a designated *initial node*, $\text{Init}(g)$. Finite *paths* in the graph can be constructed by starting at $\text{Init}(g)$ and following the (directed) edges of the CFG. We let $\text{Paths}(g)$ be the set of finite paths in g from $\text{Init}(g)$.

A graph g is a CFG for program p if the nodes of g are labelled with segments of bytecode of p or an *error node* to represent the error state s_\perp . Examples of CFGs for the program `TwoCalls` are shown in Figure 2, page 3. The nodes of the graph are assigned a unique id ($\#k$, top left of node) and are *labelled*⁶ with *segments* (0–3).

A path π uniquely defines a *sequence* of control locations, $\text{Execs}(\pi)$, obtained by concatenating the control locations of the segments on π . The set of (finite) sequences of control locations defined by a CFG g is: $\text{Execs}(g) = \{\text{Execs}(\pi) \mid \pi \in \text{Paths}(g)\}$. For instance, the path $\pi_2 = \#0 \rightarrow \#1 \rightarrow \#2 \rightarrow \#3$ in the CFG of Figure 2a defines the sequence of segments $[0, 3, 1, 3]$ and the sequence of control locations $[0x00, 0x02, 0x04, 0x0d, 0x0e, 0x05, 0x06, 0x08, 0x0a, 0x00d, 0x0e]$ which is a feasible execution in `TwoCalls`. In the CFG Figure 2b, the path $\pi_1 = \#0 \rightarrow \#1 \rightarrow \#2 \rightarrow \#1 \rightarrow \#2 \rightarrow \#1 \rightarrow \#3$ defines the sequence of segments $[0, 3, 1, 3, 1, 3, 2]$ which in turn defines the sequence of control locations obtained by expanding and concatenating the locations in each segment. Note that π_1 is not *feasible* in the actual program `TwoCalls` as they can be only two executions of Segment 3.

An important and desirable property of a CFG is that it over-approximates the set of executions of the program. Given an execution $e \in \text{Runs}(p)$, which is a sequence s_0, s_1, \dots, s_n of EVM states (Section 3.3), we define (the projection) $e.pc = s_0.pc, s_1.pc, \dots, s_n.pc$ of corresponding sequence of control locations⁷ (program counters), and we define the set of pc-runs of p as $\text{PCRuns}(p) = \{e.pc \mid e \in \text{Runs}(p)\}$. Given a program p and a CFG g for p :

► **Definition 1.** A CFG g is *sound* for program p if $\text{PCRuns}(p) \subseteq \text{Execs}(g)$.

⁶ Two nodes may be labelled with the same segment, the node/segment mapping is not necessarily 1-1.

⁷ The error state is mapped to $s_\perp.pc = -1$.

This definition does not uniquely define a sound CFG for a program.⁸ Both CFGs in Figure 2 are sound. The CFG in Figure 2a is more precise than the one in Figure 2b. The CFG in Figure 2b is an over-approximation of the CFG in Figure 2a and it contains a cycle that is not present in the CFG in Figure 2a, and not even feasible in program `TwoCalls`. The previous observation allows us to define a simple algorithm to build a CFG for EVM bytecode: from each segment ending in a `JUMP/JUMPI`, we create edges to each segment starting with a `JUMPDEST` in the code. This over-approximates the set of sequences of control locations and produces a sound CFG but this over-approximation may be too coarse to be of any use (like the CFG in Figure 2b). Our objective is to compute sound CFGs that are as precise as possible.

4.2 Abstract semantics of the EVM

To compute precise CFGs, we use *abstract semantics* (abstract interpretation) that track the program counter $s.pc$ and an *abstract stack* representation. While restrictive, this approach is sound and effective (see Section 6). This works well for two reasons:

1. valid jump targets ($\text{ValidJumps}(\cdot)$) are known at compile-time.
2. jump targets for `JUMP/JUMPI` are usually stored on the stack because: *i*) computing targets with arithmetic is rare due to gas costs and it is error-prone and *ii*) stack storage is cheaper than memory, so compilers optimise for gas by storing jump targets on the stack. Since we do not track memory, CFG construction might fail if target addresses are stored there. However, experiments (Section 6) confirm this rarely occurs in practice.

An (EVM) *abstract state* s of type **EState** is a pair $(s.pc, s.stack)$: $s.pc$ is the program counter, and $s.stack$ is an *abstract stack*, with elements either $v \in \text{ValidJumps}(p)$ or \perp which is the *unknown/arbitrary value*. The abstract representation of the stack captures the size of the stack, and the target addresses of `JUMP/JUMPI` instructions.

The *abstract semantics* of EVM instructions (Figure 3) is given by the function⁹ $\text{Next}^\alpha : \mathbf{EState} \rightarrow 2^{\mathbf{EState}}$. The abstract semantics may contain more than one successor states for some instructions like `JUMPI` that have two possible successors: one for the case when the condition is true and one for the case when the condition is false. Other instructions are deterministic and have a single successor. What is captured in the semantics in Figure 3 is that the result of an addition operation “add and pop the two values at the top of the stack, and push the result” is abstracted as \perp . It is not likely to be a target jump. For some instructions like `POP`, Next^α is identical to Next . The abstract semantics¹⁰ of `PUSHk` instructions track the concrete values that are pushed on the stack, but only if they are in $\text{ValidJumps}(p)$. For jump instructions, the abstract semantics check if the target address is a valid jump target and if the stack is not empty. This implies that if $s.stack[0] == \perp$, the target of the jump is not known, we end up in an error (abstract) state s_\perp .

4.3 CFG computation

Given a segment $[s]$, and an abstract state s such that the start address of $[s]$ is $s.pc$, we let $[s] : \text{Next}^\alpha(s)$ be the abstract states obtained after executing the segment $[s]$ from s .

⁸ There may be an infinite number of sound CFGs for some programs. A CFG is a finite automaton and defines a regular language. The set of executions of a program may not be a regular language and sometimes can be over-approximated by an infinite number of more and more precise regular languages.

⁹ The complete definition of the Next^α function can be found in the code base here.

¹⁰ The PC advances by $k + 1$ as `PUSHk` instructions have k bytes as arguments.

$$\begin{aligned}
\text{ADD} : \text{Next}^\alpha(s) &= \text{if } |s.\text{stack}| < 2 \text{ then } \{s_\perp\} \text{ else } \{(pc + 1, [\perp] + s.\text{stack}[2..])\} \\
\text{PUSHk } v : \text{Next}^\alpha(s) &= \begin{cases} \text{if } v \in \text{ValidJumps}(p) \text{ then } \{(s.pc + k + 1, [v] + s.\text{stack})\} \\ \text{else } \{(s.pc + k + 1, [\perp] + s.\text{stack})\} \end{cases} \\
\text{JUMP} : \text{Next}^\alpha(s) &= \begin{cases} \text{if } |s.\text{stack}| > 0 \wedge s.\text{stack}[0] \in \text{ValidJumps}(p) \text{ then} \\ \quad \{(s.\text{stack}[0], s.\text{stack}[1..])\} \\ \text{else } \{s_\perp\} \end{cases} \\
\text{JUMPI} : \text{Next}^\alpha(s) &= \begin{cases} \text{if } |s.\text{stack}| < 2 \text{ then } \{s_\perp\} \\ \text{else if } s.\text{stack}[0] \in \text{ValidJumps}(p) \text{ then} \\ \quad \{(s.\text{stack}[0], s.\text{stack}[2..]), (s.pc + 1, s.\text{stack}[2..])\} \\ \text{else } \{s_\perp, (s.pc + 1, s.\text{stack}[2..])\}. \end{cases}
\end{aligned}$$

■ **Figure 3** Abstract semantics of some EVM instructions.

To compute the CFG of program p , we perform a standard Depth First Search (DFS) traversal of the graph defined by Next^α starting from the initial abstract state $(0x00, [])$. It is not guaranteed that the DFS will terminate, as the stack may grow indefinitely.

To ensure termination, we limit the DFS to a given maximum depth (e.g., 100). If the maximum depth is reached (indicated by the result of the DFS), we can try a larger bound and iterate until the CFG fits within the bound. We write $\text{CFG}(p)$ for the CFG of program p as computed by our DFS algorithm on abstract states.

The DFS algorithm completes the exploration of a branch when it encounters a previously visited node with the same abstract state. However, in some cases, this may lead to non-termination regardless of the maximum depth limit, as the *abstract stack* may grow arbitrarily large. In the next section we provide a sufficient condition to test whether two abstract states are *equivalent* and can be safely merged.

4.4 Loop detection

The bytecode in Listing 3 is a simple program that contains an unbounded loop. The CFG for this program is depicted in Figure 4. After n execution of the **JUMP** instruction, the abstract stack content at PC location $0x02$ is $[0x02, \perp, \perp, \dots, \perp]$ with n unknown values.

The DFS algorithm introduced previously will not terminate for any maximum depth limit. However, we can see that the part of the stack that grows arbitrarily large is made of \perp elements and this has no impact on the target of the **JUMP** instruction, which remains constant and equal to $0x02$. For instance when we run the DFS algorithm we explore the path $\pi' = (0x00, []) \rightarrow (0x02, [0x02]) \rightarrow (0x02, [0x02, \perp])$. If we execute Segment 1 from the state $(0x02, [0x02, \perp])$, we end up in a state where the top of the stack is still $0x02$. The state $(0x02, [0x02, \perp])$ can be merged with the state $(0x02, [0x02])$ because every future execution after $(0x02, [0x02, \perp])$ is covered by executions starting from $(0x02, [0x02])$. To identify such configurations, we need to determine that every execution of Segment 1 results in the **JUMP** at its end always targeting $0x02$.

We can provide a *sufficient condition* to test whether this is the case:

1. Compute the *weakest precondition* that ensures that at the end of Segment 1 the value of the PC is $0x02$;
2. check whether this weakest precondition is *invariant* when executing Segment 1.

Assume we want to *prove* that after executing the `JUMP` instruction at 0x06 (line 11, Figure 4), the program counter is 0x02. As a `JUMP` instruction updates the program counter to the value at the *top* element of the stack, the *least* we can require is that, before we execute the `JUMP` instruction, the top of the stack contains 0x02. Any other choice would fail to result in `pc == 0x02` after the `JUMP`. If execution begins in a state s where the top of the stack holds 0x02, represented by the predicate $s.stack[0] == 0x02$, then executing a `JUMP` instruction guarantees that the next instruction to execute is at location 0x02.

To check the previous condition, we can compute the *weakest precondition* that ensures that executing a `JUMP` instruction leads to the postcondition `pc==0x02`. Weakest preconditions are standard concepts in formal verification and we illustrate the computation of weakest preconditions with a simple example. We let $Wpre(i, P)$ denote the weakest precondition (a predicate) for instruction i to ensure that P (a post condition) holds after the execution of the instruction. For instance, for the `JUMP` instruction, we have:

$$Wpre(JUMP, pc == 0x02) = s.stack[0] == 0x02.$$

Computing weakest preconditions for *sequences* of instructions amounts to propagating the weakest preconditions backwards. For a sequence of instructions $\sigma + [a]$, and a predicate P , $Wpre(\sigma + [a], P) = Wpre(\sigma, Wpre(a, P))$ with $Wpre([], P) = P$. If we perform this computation for all the instructions in Segment 1, we obtain condition C_1 :

$$C_1 = Wpre(JUMPDEST PUSH0 SWAP1 DUP1 JUMP, pc == 0x02) = s.stack[0] == 0x02.$$

To decide whether we can add a loop edge from node #1 to node #1 in the CFG of the program, we are going to check whether C_1 is *preserved* by Segment 1. If this is the case we have an *invariant* and a proof that any future execution of Segment 1 will end up in a state that satisfies C_1 . To check whether C_1 is preserved by Segment 1, we can compute the *postcondition* of C_1 after executing Segment 1. If it *implies* C_1 then it is an invariant. We have implemented this check in the DFS algorithm, and we can successfully build a CFG for programs with unbounded stacks similar to Listing 3.

Listing 3 Loop1 with an unbounded stack.

```

1 PC Instruction // abstract stack
2
3 // Segment 0
4 0x00: PUSH1 0x02 // [0x02]
5
6 // Segment 1
7 0x02: JUMPDEST // [0x02]
8 0x03: PUSH0 // [?, 0x02]
9 0x04: SWAP1 // [0x02, ?]
10 0x05: DUP1 // [0x02, 0x02, ?]
11 0x06: JUMP // [0x02, ?] ...

```



Figure 4 The CFG of Loop1.

4.5 Minimisation

Finally, we apply a minimisation algorithm after the DFS to collapse nodes that are *language-equivalent*. Minimising the CFG amounts to minimising an automaton and can be done with standard techniques to *partition* the set of nodes in *equivalence classes*. We have

implemented a minimisation algorithm (a mixture of Moore and Hopcroft algorithms) that can generate a minimised version of the CFG. In the experiments the minimisation stage provides marginal benefits. It minimises very short subgraphs of the CFG, mostly nodes that contain segments `STOP` or `REVERT` instructions.

5 Soundness test for CFGs

We formally verify that ByteSpector’s CFGs are sound, ensuring they can be reliably used for code analysis, auditing, and inspection. Since the CFG algorithm relies on abstract semantics, errors could arise from opcode interpretation or other parts of the code. The CFG computation itself is not certified, it is not inherently proven to always produce sound CFGs. To address this, we perform soundness checks, making ByteSpector a certifying CFG generator, as per the definition in [9].

5.1 Soundness proof

Given a program p , a node n in $\text{CFG}(p)$ corresponds to an abstract state $n = (n.pc, n.stack)$. The component $n.stack$ is a list of either concrete values (target of jumps) or \perp . As a result we can view $n.stack$ as a *finite set of constraints* $n.\phi$, of the form $n.stack[i] = v$, with $v \in \text{ValidJumps}(p)$ and write a node/abstract state in the form $(n.pc, n.\phi)$. The constraints $n.\phi$ enforces that some elements of the stack have some specific values.

To prove that $\text{CFG}(p)$ is sound, we need to show that it *simulates* the executions of the program p . To do so we prove the following:

Condition 1 The initial `EVMState` state s_0 of the execution of p in the EVM satisfies the constraints of the initial node $\text{Init}(g)$, i.e., $s_0.pc = \text{Init}(g).pc$ and $s_0.stack$ satisfies the constraints $\text{Init}(g).\phi$.

Condition 2 For any node $n = (n.pc, n.\phi)$ in $\text{CFG}(p)$, and any concrete `EVMState` state $s = (s.pc, s.stack, \dots)$ that satisfies the constraints of n i.e., $s.pc = n.pc$ and $s.stack$ satisfies the constraints $n.\phi$, if $s' = \text{Next}(s)$ then there exists a (direct) successor node $n' = (n'.pc, n'.\phi)$ of n in $\text{CFG}(p)$ such that $s'.pc = n'.pc$ and $s'.stack$ satisfies $n'.\phi$.

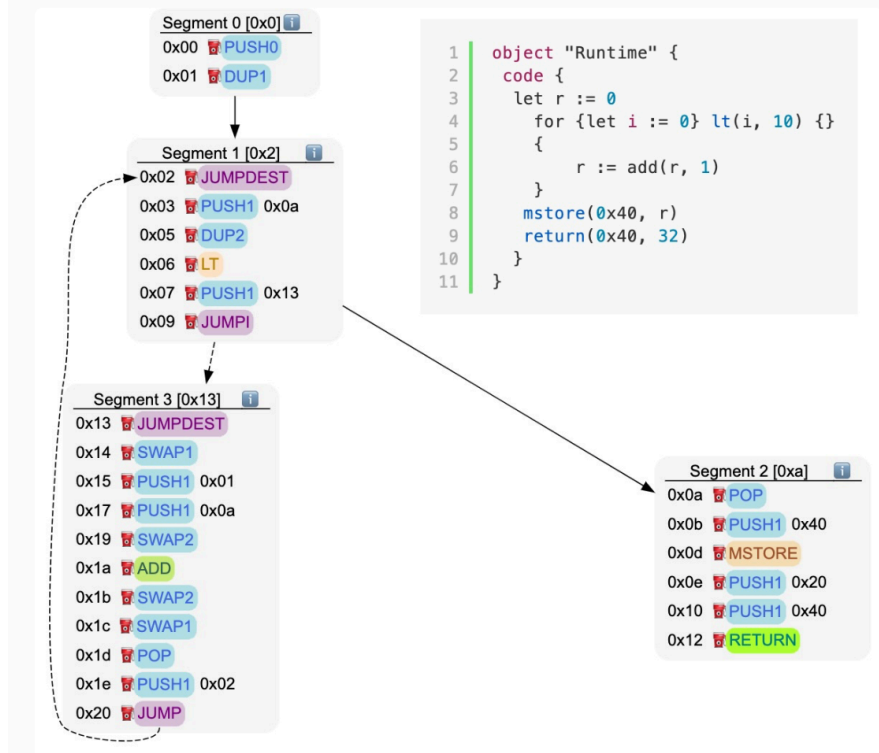
Proving these two properties for all nodes of the CFG ensures (by induction) that the CFG is sound. Condition 1 relating the initial state of the EVM and the initial node of the CFG is straightforward to prove: the initial state of the EVM starts at $pc = 0$ and an empty stack, and the initial node of the CFG is associated with the segment of code starting at $pc = 0$ and an empty stack. In the Section 5.3 we explain how to prove the second condition.

5.2 Proof objects for EVM bytecode

To prove the soundness of a CFG, we generate a Dafny proof object that encodes the bytecode semantics and node transitions.

The Dafny-EVM library provides functions like `PUSH0`, `DUP`, and `POP` to represent EVM instructions as state transformations, and `EVMState` that models the EVM state. Listing 4 shows the proof object for `SimpleLoop`, and Figure 5 shows the CFG for `SimpleLoop`. The semantics of Segment 0 is implemented in `ExecuteFromCFGNode_s0` lines 2-11. This function maps an EVM state,¹¹ `s0: EVMState` to a state `s': EVMState` obtained after executing the segment.

¹¹The `gas` parameter is explained later.



■ **Figure 5** A CFG for program `SimpleLoop` (top right). The CFG contains a cycle and is an over-approximation of the set of executions of `SimpleLoop` which is limited to 10 iterations of the loop. The bytecode for `SimpleLoop` is not shown but rather the source intermediate representation in Yul code that is compiled to the bytecode with the Solidity compiler.

Each function has a *precondition* (`requires`) ensuring the *pc* in the initial state `s0` matches the segment start address. The CFG of `SimpleLoop` indicates that the successor of node 0 is node 1, so the line 10 is a call to the execution of the next node/segment, `ExecuteFromCFGNode_s1`. When we verify the proof object, Dafny checks that for all input states `s0` that satisfy the precondition of `ExecuteFromCFGNode_s0`, the precondition of `ExecuteFromCFGNode_s1` is satisfied at the callsite in line 10. If Dafny successfully verifies `ExecuteFromCFGNode_s0` the condition 2 is verified for node 0. The encoding of the other nodes and edges follow the same pattern.

Dafny requires proof of termination. For CFGs with loops, we use a `gas` parameter that strictly decreases with each transition in the CFG, proving that all paths terminate. This effectively verifies soundness for every path of arbitrary length, not actual gas usage.

Note that with our encoding and Dafny-EVM semantics, we prove more than soundness. In the Dafny-EVM, each operation e.g., `Pop(s0)` at line 38, may have some preconditions: e.g., the stack must not be empty, otherwise the `POP` operation results in a error state (stack underflow). In our encoding, we ensure that the stack preconditions of each Dafny-EVM function (`Pop`, `PushN`, `Swap`, ...) in the code are satisfied. To do so, we add some constraints, for example, for `ExecuteFromCFGNode_s1` to execute without error, we require that the stack on entry has more than two elements `requires s0.Operands() >= 2`.

To summarise, given a CFG, we can build a Dafny program, a proof object, that encodes the semantics of the bytecode in each node, and the transitions between nodes. If the proof object verifies, we have a proof that the CFG is sound (and no stack underflow exceptions¹² occurs.) This provides a soundness test for CFGs generated by `ByteSpector`.

¹²The absence of stack overflows can be guaranteed by adding a requirement on the capacity of the stack in the initial state.

■ Listing 4 Proof object for program SimpleLoop

```

1  /** Node 0. Segment Id is:0. Starting at 0x0. Type is: CONT Segment */
2  function ExecuteFromCFGNode_s0(s0: EVMState, gas: nat): (s': EVMState)
3  requires s0.pc == 0x0 as nat // PC requirement for this node.
4  requires s0.Operands() >= 0 // Stack requirements for this node.
5  decreases gas {
6    if gas == 0 then s0
7  else
8    var s1 := Push0(s0); // Push 0 onto the stack
9    var s2 := Dup(s1, 1); // Dup1: Duplicate the top of the stack
10   ExecuteFromCFGNode_s1(s2, gas - 1) // Go to the next instruction at pc + 1
11 }
12 /** Node 1. Segment Id is:1. Starting at 0x2. Type is: JUMPI Segment */
13 function ExecuteFromCFGNode_s1(s0: EVMState, gas: nat): (s': EVMState)
14 requires s0.pc == 0x2 as nat // PC requirement for this node.
15 requires s0.Operands() >= 2 // Stack requirements for this node.
16 decreases gas {
17   if gas == 0 then s0
18   else
19     var s1 := JumpDest(s0);
20     var s2 := PushN(s1, 1, 0x0a);
21     var s3 := Dup(s2, 2);
22     var s4 := Lt(s3);
23     var s5 := PushN(s4, 1, 0x13);
24     var s6 := JumpI(s5);
25     if s5.stack[1] > 0 then // This is a JUMPI segment
26       ExecuteFromCFGNode_s3(s6, gas - 1)
27     else
28       ExecuteFromCFGNode_s2(s6, gas - 1)
29 }
30 /** Node 2. Segment Id is:2. Starting at 0xa. Type is: RETURN Segment */
31 function ExecuteFromCFGNode_s2(s0: EVMState, gas: nat): (s': EVMState)
32 requires s0.pc == 0xa as nat // PC requirement for this node.
33 requires s0.Operands() >= 2 // Stack requirements for this node.
34 decreases gas {
35   if gas == 0 then s0
36   else
37     var s1 := Pop(s0);
38     var s2 := PushN(s1, 1, 0x40);
39     var s3 := MStore(s2);
40     var s4 := PushN(s3, 1, 0x20);
41     var s5 := PushN(s4, 1, 0x40);
42     var s6 := Return(s5);
43     s6 // Segment is terminal
44 }
45 /** Node 3. Segment Id is:3. Starting at 0x13. Type is: JUMP Segment */
46 function ExecuteFromCFGNode_s3(s0: EVMState, gas: nat): (s': EVMState)
47 requires s0.Operands() >= 2 // Stack requirements for this node.
48 decreases gas {
49   if gas == 0 then s0
50   else
51     var s1 := JumpDest(s0);
52     var s2 := Swap(s1, 1);
53     var s3 := PushN(s2, 1, 0x01);
54     var s4 := PushN(s3, 1, 0x0a);
55     var s5 := Swap(s4, 2);
56     var s6 := Add(s5);
57     var s7 := Swap(s6, 2);
58     var s8 := Swap(s7, 1);
59     var s9 := Pop(s8);
60     var s10 := PushN(s9, 1, 0x02);
61     var s11 := Jump(s10);
62     ExecuteFromCFGNode_s1(s11, gas - 1) // JUMP to the target at Peek(0)
63 }

```

5.3 Efficient soundness test

The technique we have described previously has some limitations:

- for large segments of code, Dafny may be unable to verify some functions of the proof object due to the complexity of the code. Indeed, the Dafny-EVM semantics is quite complex for some opcodes (and the state has several components) and the verification of the proof object may require a lot of resources, and time out.
- some opcodes correspond to calls to other contracts e.g., `CALL`, `DELEGATECALL`, `STATICCALL`. In the CFG generation algorithm, the effects of these calls are abstracted, only taking into account the effect on the stack. The Dafny-EVM semantics of these instructions is more complex, involving the memory, storage, and the return value of the call.

The solution to these problems is to use an *abstract version* of the Dafny-EVM semantics. The abstract version is restricted to tracking the (abstract) stack and the program counter, and does include the memory, storage, etc.

For calls to other contracts, we encode the visible effect of the call on the stack: from the caller point of view, a `CALL` instruction pops 7 arguments from the stack and pushes one return value (that could be an error code) if it does not abort. As a result the semantics of call instructions can be abstracted as a simple stack operation as shown in Listing 5.

■ **Listing 5** Abstract semantics of calls

```

1 function Call(s: EState): (s': EState)
2   // CALL needs seven arguments
3   requires s.Operands() >= 7
4   {
5     // CALL pops 7 arguments and if it does not abort, returns a result on top of the stack
6     EState(s.pc + 1, [0] + s.stack[7..])
7   }
```

To preserve soundness, we can prove that the abstract semantics *simulate* the Dafny-EVM semantics, and the proof can be written in Dafny. The proof for the `POP` instruction is specified in Listing 6.

■ **Listing 6** Simulation proof for abstract semantics

```

1 // A Dafny lemma to prove that abstract Pop simulates Bytecode.Pop
2 lemma SimulationCorrectPop(s: EState, st: EVMState)
3   requires s.ABSTRACTS(st) // if s abstracts st
4   // if executing Pop from st leads to a non error state Bytecode.Pop(st) (type EXECUTING?)
5   ensures Bytecode.Pop(st).EXECUTING? ==>
6     // then 1. abstract Pop can be executed from s (preconditions of Pop are satisfied)
7     Pop.requires(s)
8     // and 2. executing abstract semantics Pop from s leads to a state Pop(s)
9     // that abstracts Bytecode.Pop(st)
10    && Pop(s).ABSTRACTS(Bytecode.Pop(st))
11  { // Thanks Dafny
12  }
```

6 Evaluation

ByteSpector is implemented in Dafny [8] and available at <https://github.com/franck44/evm-dis>. In our experiments, ByteSpector can successfully construct provably sound CFGs for 978 out of 1048 contracts (93.3% success rate).

6.1 Implementation

ByteSpector consists of 6 modules, 36 files, 6000 lines of code, and 2200 lines of comments. The implementation is formally verified, ensuring no division-by-zero, out-of-bounds errors, or non-terminating functions. We use pre/postconditions and datatype constraints to enforce functional correctness. For example, the segment types in Dafny (`JUMP`Seg, `STOP`Seg) are constrained to end with the corresponding EVM instructions (`JUMP`, `STOP`) and this property is statically verified on the Dafny code. In our experimental evaluation, we have used the Java backend to generate a Java version of ByteSpector.

6.2 Experiments

For experimental evaluation we use a dataset from the project EVMLiSA [2], available at <https://github.com/lisa-analyzer/evm-lisa>.

The initial dataset `less_than_3000_opcode.txt` has 1704 EVM bytecode contracts' addresses. The contracts are published on Ethereum, live, and their bytecodes can be retrieved using the `etherscan.io` API. Each contract has less than 3000 opcodes. In our experiments we excluded the contracts containing the recently introduced instructions `RJUMP`, `RJUMPI`, `RJUMPV` and set the maximum depth to 100 for the CFG generation algorithm. These instructions do not present major difficulties and are already partially supported in our code base, but we have not yet fully integrated them in the CFG generation algorithm. Overall there are 1048/1704 contracts that 1) do not contain `RJUMP`'s instructions and 2) the maximum depth is not reached during the generation of the CFG. With our technique we can generate and verify the CFG for almost all the contracts. A few of them contain jump targets that are stored in memory (probably using an old version of the Vyper compiler).

7 Related work and conclusion

Decompiling low-level code and building CFGs has a long history. We refer the reader to [4] for an history of decompilation. More specifically, for the decompilation of EVM bytecode, most of the tools use static analysis techniques (and value sets) or symbolic execution and SMT-solvers to generate CFGs. Tools like Oyente [3], EthIR [1], Mythril [10] and EtherSolve [6] use symbolic execution and/or SMT-solvers to compute CFGs. Some of these tools are not maintained anymore and it is not possible to compare our results with them.

EVMLiSA [2] is a tool with a large set of benchmarks. The technique used in EVMLiSA relies on stack abstraction too, but the abstraction is more complicated than the one we presented in this paper: it tracks sets of stacks.

Our contribution goes beyond the computation of CFGs and outperforms the previous tools in several aspects: *i*) the CFGs are certified (formally verified); *ii*) the proof objects can be instrumented with specifications to capture more complicated functional requirements and *iii*) the source code (<https://github.com/franck44/evm-dis>) is written in Dafny, verified, and artefacts available in several target languages (Java, Python, C#) and can be integrated in existing code bases.

References


- 1 Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. EthIR: A Framework for High-Level Analysis of Ethereum Bytecode. In Shuvendu K. Lahiri and Chao Wang, editors, *Automated Technology for Verification and Analysis*, pages 513–520, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-030-01090-4_30.

- 2 Vincenzo Arceri, Saverio Mattia Merenda, Greta Dolcetti, Luca Negrini, Luca Olivieri, and Enea Zaffanella. Towards a sound construction of EVM bytecode control-flow graphs. In Luca Di Stefano, editor, *Proceedings of the 26th ACM International Workshop on Formal Techniques for Java-like Programs, FTfJP 2024, Vienna, Austria, 20 September 2024*, pages 11–16. ACM, 2024. doi:10.1145/3678721.3686227.
- 3 Syed Badruddoja, Ram Dantu, Yanyan He, Kritagya Upadhayay, and Mark Thompson. Making smart contracts smarter. In *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–3, 2021. doi:10.1109/ICBC51069.2021.9461148.
- 4 Zion Leonahenahe Basque. 30 years of decompilation and the unsolved structuring problem: Parts 1 & 2. Blog Post. Accessed 2024-10-11. URL: <https://mahaloz.re/dec-history-pt1>.
- 5 Franck Cassez, Joanne Fuller, Milad K. Ghale, David J. Pearce, and Horacio Mijail Anton Quiles. Formal and executable semantics of the ethereum virtual machine in dafny. In Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker, editors, *Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings*, volume 14000 of *Lecture Notes in Computer Science*, pages 571–583. Springer, 2023. doi:10.1007/978-3-031-27481-7_32.
- 6 Filippo Contro, Marco Crosara, Mariano Ceccato, and Mila Dalla Preda. Ethersolve: Computing an accurate control-flow graph from ethereum bytecode. In *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021*, pages 127–137. IEEE, 2021. doi:10.1109/ICPC52881.2021.00021.
- 7 Uri Kirstein. Bug Disclosure – Solidity Code Generation Bug Can Cause Memory Corruption. Medium Post by Certora, 2021. Accessed 2024-10-11. URL: <https://medium.com/certora/bug-disclosure-solidity-code-generation-bug-can-cause-memory-corruption-bf65468d2b34>.
- 8 K. Rustan M. Leino. Accessible software verification with Dafny. *IEEE Softw.*, 34(6):94–97, 2017. doi:10.1109/MS.2017.4121212.
- 9 Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, 2009. doi:10.1007/s10817-009-9155-4.
- 10 Bernhard Mueller. Smashing smart contracts, 2018. 9th HITB Security Conference. URL: <https://tinyurl.com/y827tk72>.
- 11 Natachi Nnamaka. The Vyper Compiler Saga: Unraveling the Reentrancy Bug that Shook DeFi. Medium Post by Rektify AI, 2023. Accessed 2024-10-10. URL: <https://medium.com/rektify-ai/the-vyper-compiler-saga-unraveling-the-reentrancy-bug-that-shook-defi-86ade6c54265>.
- 12 John Toman. Overly Optimistic Optimizer – Certora Bug Disclosure. Medium Post by Certora, 2022. Accessed 2024-10-10. URL: <https://medium.com/certora/overly-optimistic-optimizer-certora-bug-disclosure-2101e3f7994d>.
- 13 David Wood. Ethereum: a secure decentralised generalised transaction ledger, 2022. Berlin version d77a387 - 2022-04-26. URL: <https://ethereum.github.io/yellowpaper/paper.pdf>.

Towards a Mechanization of Fraud Proof Games in Lean

Martín Ceresa 

IMDEA Software Institute, Madrid, Spain

César Sánchez 

IMDEA Software Institute, Madrid, Spain

Abstract

Arbitration games from Referee Delegation of Computations are central to layer two optimistic rollup architectures (L2), one of the most prominent mechanisms for scaling blockchains. L2 blockchains operate on the principle that computations are valid unless proven otherwise. Challenging incorrect computations requires users to construct fraud proofs through a dispute resolution process that involves two opposing players. Fraud proofs are objects that establish when proposed computations are invalid, and they are so computationally small and cheap that can be checked by the underlying trusted blockchain. Arbitration games, this challenging process, involve one player posing strategic questions and another player revealing details about computations.

Arbitration games start from the posting of Disputable Assertions (DAs), DAs contain partial information about computations including their result. Since there is no trust between players, hashes are posted as compact witnesses of knowledge. One player provides information decomposing hashes while the other decides which “path” to take navigating the computation trace. When a path is exhausted, all the required information to compute the result from the data provided following the path has been revealed and the path can be proven to be faulty or correct.

We explore in this paper the formalization of arbitration games in Lean4, introducing the first machine-checkable strategies that honest players can play guaranteeing success. These strategies ensure: on one side, the successful debunking of dishonest computations via the construction of fraud proofs, while in the other, the successful navigation of the challenge process through correct answers. In short, these are the winning strategies that honest players (on both sides) can follow. We explore in this paper formal abstractions to capture disputable assertions, arbitration games on finite binary trees asserting data-availability and membership, game transformations, and then discuss how to work towards a general formal framework for referee delegation of computations.

2012 ACM Subject Classification Software and its engineering → Formal methods; Software and its engineering → Correctness; Software and its engineering → Software libraries and repositories; Theory of computation → Interactive proof systems; Theory of computation → Program reasoning; Theory of computation → Program constructs

Keywords and phrases blockchain, formal methods, layer-2, optimistic rollups, arbitration games

Digital Object Identifier 10.4230/OASICS.FMBC.2025.5

Supplementary Material *Software*: <https://gitlab.software.imdea.org/martin.ceresa/leanfpgames> [13], archived at `swb:1:dir:761ba38f606c1b4a0a9e202e6518d092d51ff381`

Funding Partially funded by DECO Project (PID2022-138072OB-I00) – funded by MCIN/AEI/10.13039/501100011033 and by the ESF+ – and by grant from Nomadic Labs and the Tezos Foundation.

Acknowledgements Thanks to Margarita Capretto for her insightful ideas and help.



© Martín Ceresa and César Sánchez;

licensed under Creative Commons License CC-BY 4.0

6th International Workshop on Formal Methods for Blockchains (FMBC 2025).

Editors: Diego Marmosoler and Meng Xu; Article No. 5; pp. 5:1–5:17

OpenAccess Series in Informatics



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

1 Introduction

Blockchain¹ technology is the first massively adopted decentralized third-party trusted mechanism to perform money exchanges [25]. The second wave of blockchains introduced computational mechanisms, so users were not only capable of performing money transactions but also they could perform computations [9]. Users now can upload small programs, called smart contracts (contracts), whose execution governs the interaction between users, including the transfer of tokens and cryptocurrency. Enabling agents to describe and invoke computations opened a whole new horizon in this field, for the good or bad.

The good part is the fulfilling of a long due dream of having a technology to run trusted third-party code [34, 37]. The contract code describes with precision what is going to be executed, and users can trust and reproduce what the resulting effects of executing transactions are [29]. This spawned a new challenge of formal program verification [33, 28, 2, 26, 6, 15, 5, 3, 31, 32, 17, 4, 23, 11].

The bad part is that this new technology came with a whole new attack surface on contracts. Programs now share the same “memory-space”, all agents can invoke contracts and contracts can invoke any other contract, which can potentially produce unexpected transfer of cryptocurrency, or the break of interaction protocol designed between contracts by developers [14].

The massive adoption of contract based blockchains came with another cost. Resources are limited and blockchains are decentralized, so there is limit in the number of transactions per unit of time that can be included in the blockchain, making computation expensive. Expensive computations come in two flavours: expensive execution and space costs. To solve these two problems, several solutions were proposed: scaling the blockchain itself (with techniques like sharding [36] or faster consensus protocols [19, 27, 16]), or devising mechanisms to compute offchain (outside the blockchain), minimizing the blockchain interactions [19].

Rollups are offchain mechanisms to provide a solution for blockchain scalability. These offchain solutions are potentially dangerous since the only trusted computation is what executes in the blockchain using contracts (onchain). Therefore, to provide the same guarantees as the underlying blockchain, L2 introduces new concepts, and they come (broadly speaking) in two flavours.

Zero-knowledge Rollups. Zero-Knowledge (ZK) proofs are mechanisms to provide proof of correctness of computations that can be verified onchain by a smart contract [35]. Ideally, committing zk-proofs and checking their correctness is less expensive than actually running the computations they verify. In addition, the system must guarantee that no faulty proofs can be produced, and that all correct computations can be proven (even if crafting their proof is expensive). Therefore, instead of running contracts, zk-proofs are generated and provided to be checked onchain, and if the proof passes the verification process, the L2-blockchain evolves. This approach is implemented by several solutions² but a scalable solution based on ZK-proofs is still under research [22].

Optimistic Rollups. Computations are assumed correct unless proven otherwise [21, 10]. Instead of verifying zk-proofs, optimistic rollup schemes employ an arbitration mechanism that guarantees a single honest agent (with sufficient resources) is capable of detecting and

¹ We refer to distributed ledgers with crypto-currencies capabilities as *blockchain*.

² Readers can check the state of L2 ecosystems at <https://l2beat.com/scaling/summary>

reverting dishonest computations as well as defending correct computations. The correctness of the system is then predicated on a single agent observing the evolution of the system. The scalability predicates on the deterrent of being caught lying preventing dishonest players from posting dishonest computations. Instead of running checks or other mechanisms on transactions, optimistic rollups only keep track of the proposed transactions and block effects produced by executing them. There is a period window where these proposed effects can be challenged. When proposals are challenged, the party proposing the next state and the challenging party play an arbitration game to decide whether the proposal is (in)correct. The proposal is discarded if the challenging party wins the arbitration game or stays alive waiting for the period to end otherwise. Losing parties lose a stake that they must place and winning parties win a reward, which create incentives to participate honestly in the ecosystem and to deter dishonest behavior. Proposals surviving the challenging time period become permanent and the L2-blockchain evolves. That is, the correctness of an L2 optimistic rollup lays on the combination of (1) a correctness criteria that states the ability of a single honest to effectively dispute dishonest allegations (either remove faulty computations or defend honest computation claims), and (2) the economic incentives for agents to follow honest behavior. We focus in this paper only on the computational part of these schemes.

The correct evolution of L2 optimistic rollup blockchains rely on the actions of honest agents. Honest players can locally compute and propose the next blockchain state executing sequentially transaction requests. A single honest player is capable of challenging faulty proposals and winning arbitration games. In this article, we focus on the study of the correctness of arbitration games formally. It is crucial that an honest proposer can propose the next block knowing that it can always defend the proposals. Also, a single honest challenger must be able to challenge a dishonest proposal knowing they can always win, preventing the blockchain from progressing dishonestly.

Previous works capture computations as interactive games between players formally using *interaction trees* [20, 38]. Moreover, there is a big effort in characterizing general computations and proving properties about programs and protocols [7, 8]. We do not follow such a general approach here but a much more concrete one, because of the nature of our model of computation, where authenticated data structures are at the core of our approach. We took a more pragmatic path and trying to maximize the use of two simple building blocks: Merkle tree formation and membership games. An important characteristic of our approach is that our games are finite and players have bounded time to play (as in chess clocks).

The main artifact of our work is a Lean4 library that models the concepts in arbitration games, including DAs, players and strategies. We provide proofs showing the correctness of the strategies for honest players. Moreover, since Lean4 generates executable programs, we exercise our strategies and have them interactively play the arbitration game.

The readers can find the code of the library and all proofs in a git repository <https://gitlab.software.imdea.org/martin.ceresa/leanfpgames> where we created a tag pointing at the current state of the library “FMBC.Final”. The repository is not yet complete but will be publicly released as a full library in the near future.

2 Preliminaries

In this section, we briefly explain the relevant notions to this article of Optimistic Rollups. Arbitrum Optimistic Rollups [21] are the first implementation of L2 optimistic rollups, implementing Referee Delegation of Computations [10], serving as the leading example in the L2 ecosystem attacking blockchain scalability issues. The main idea is to perform as

much computation as possible outside the blockchain while keeping the same guarantees. Optimistic Rollups propose to optimistically execute transactions, i.e. transaction executions are assumed correct unless someone challenges the correctness of the computation³.

The evolution of the L2-blockchain goes as follows: one agent proposes a *disputable assertion* (DA) asserting a fact, e.g. what the result of executing the effect of a transaction is, and if the DA survives a fixed period of time, it becomes a committed fact. We can see how this is appealing for blockchains, no expensive computations are done if DAs survive. The brilliant idea comes when DAs are challenged.

When a DA is challenged, the agent proposing the DA, the *proposer*, and the agent challenging, the *challenger*, engage in a turn-based two player game. In this game, called arbitration game, the two players compete against each other to build a witness. Witnesses prove that one of the players is wrong and the other is right. When witnesses prove proposers wrong, they are called fraud-proof and are employed to debunk claims. When witnesses prove challengers wrong, they do not prove proposers right, and depending on the claim and arbitration game, they can provide partial correctness.

Participation in the L2 blockchain evolution is rewarded and misbehaviour penalized and, in arbitration games, it is strictly enforced through penalties. Players stating facts (DAs) place stakes on them, and in the case they cannot support their claims, they lose their stakes. Players challenging DAs place stakes on their (challenging) claim, which they can lose if they cannot debunk the supposedly false claim they are challenging. The entire game is played through the underlying blockchain and the losing party loses their stakes and the winning party is rewarded. We focus on the computation mechanics and not in the monetary analysis of arbitration games.

One may ask why would players lie if they are going to lose money. In this article, we provide a way to prove that lying players are going to get caught by playing the game⁴. We complete our argument with that rational agents will not play to lose money⁵, and thus, arbitration games are never played⁶. In other words, arbitration games are deterrent mechanisms never to be played but necessary to guarantee the correct evolution of the blockchain.

Optimistic Model of Computation for Blockchains

In L2 Optimistic Rollups, the goal is to compute the next state of the blockchain, i.e. the result of applying the effects of executing transactions in a given block. In Refereed Delegation of Computation (RDoC) [10], the authors decompose the execution of transactions as small steps in Turing machines, while in Arbitrum [21], they use small steps in the ethereum virtual machine described using WASM. Both approaches map trust of the whole computation to a single computation step run in a trusted computation device, a blockchain.

RDoC and Arbitrum use hashes to represent compact witnesses. Assuming there is a collision resistant hash function \mathcal{H} , one can use Merkle Trees, a compact representation of trees into single hashes. Merkle trees are the main example of authenticated data structures, one can locally verify element e belongs to a tree with a path of hashes leading to e without knowing the entire data behind the Merkle tree.

³ In this article, we do not address questions as Sibyl attacks preventing agents from challenging malicious transactions by blocking their access to the blockchain.

⁴ Assuming players have enough money and can interact with the blockchain to play the game.

⁵ Assuming a close economic world. In open economic worlds, agents may lose at some small close markets while winning somewhere else and actually have bigger net earnings.

⁶ To the authors knowledge, there is no evidence of arbitration games ever played in the whole Arbitrum blockchain history nor in similar L2 blockchains like Optimism.

Transactions as checkable sequences of operations. Transactions in blockchain are stored as traces of executed operations, representing all the small steps required to go from the previous state to the current state of the blockchain. When these operations are expanded, we see their name and arguments. Therefore, we can hash each basic operation along with its arguments to create a compact witness, a hash which is the sole result from hashing such operation (because we are using a collision resistant hash function). Now, we can form a balanced tree with the sequence of hashes and form a Merkle tree with them. This hash is a witness of the whole computation generated by the original transaction.

The main idea of L2 optimistic rollups is to propose DAs asserting the result of the computation to be as compact as possible. Therefore, we have DAs as sequences of hashes plus their Merkle tree representing the skeleton of the computation tree. Other players can challenge DAs by requesting hashes until leaves are found. When a leaf is found, the proposing party needs to reveal the raw data used to compute leaf hashes, i.e. basic operations and their arguments. Therefore, the last basic step has to be run by the trusted computation device. One game is that whenever challengers request hashes, we split the computation in half bisecting the trace, and ask the DA proposer to provide the corresponding hash. This game is called bisection game and they are a subclass of a more general notion of game called arbitration games.

In theory, i.e. in the work of RDoC [10], computations are represented as the steps taken by Turing machines. Therefore, having a trusted single Turing machine step interpreter is enough to simulate one step and provide trusted execution to the whole ecosystem. In practice, i.e. in the work of Arbitrum [21], computations are represented as small steps taken by the ethereum virtual machine (EVM). Therefore, in Arbitrum, they instrumented the EVM and adapted it to single step executions using WASM and the bisection game is performed over their machine steps, the Arbitrum Virtual Machine (AVM). As result of executing transactions, we have a list of low-level verifiable atomic operations taking the current state of the blockchain to the next one.

In this work, we abstract these small steps and only focus in the main operations over Merkle trees. This work is part of a bigger research enterprise where we believe there are other ways to trust from small to big computations, we show our first step as an example in Section 5 and a small discussion in Section 6.

Agents observing the blockchain know everything. All information is public, and thus, all agents know what transactions are being executed and can compute their resulting effects. In particular, agents can compute all intermediate steps and the resulting Merkle tree of all computations. Therefore, if an agent is lying other agents know about it and can engage in an arbitration game.

In the protocol just described, two things can go wrong:

- The Merkle tree hash is not the hash resulting from hashing the trace.
- There is something wrong with the data (elements) in the trace.

Therefore, there are two basic building blocks in this protocol.

When posting DAs, the proposing agents are committed to provide (if required) information derived from the witness. Witnesses are Merkle trees, so the hashes provided must hash the parent hash in each step. In Arbitrum, when agents propose the next step in the L2 blockchain placing a DA, they publish a compression of the trace plus the resulting hash. When there is something wrong with the trace (or the trace itself), opposing agents challenge such DA. In both challenges the idea is the same, having a top hash, the agent defending the DA (usually the one proposing it) decomposes the top hash into other hashes, and thus we have a way to link previous proof witnesses into the new ones. The challenger party decides which path to take in case hashes are wrong, repeating the process.

The difference between the two possible dishonest moves is whether or not the challenger party accepts the top hash to be correct. If the top hash is incorrect, the hash does not follow from the data proposed, a *data-availability* arbitration game is played ⁷(see Section 3.2). If the top hash follows from the data but the data is incorrect, an *membership* arbitration game is played (see Section 4). In the case where the membership arbitration game is played, we also need a trusted validation function so we can test its validity once an element is proved to be part of the data provided. Other properties can be defined using the membership arbitration game, for example, to show that an element appears twice in a block, it suffices to show that it appears at two different places.

3 A Generic Fraud Proof Game Formalization

In this section, we present an abstract formalization of fraud proof games that encompasses the arbitration games (and RDoC) presented in Section 2. Then, we instantiate this formalization to other games proposed in [12], which are games used over correct encodings of batches of transactions in Layer 2 blockchains. These games include, for example, specific games to decide whether an element belongs to a Merkle tree (see Section 4).

3.1 DAs

Protocols begin when a proposing agent asserts the result of a given computation. We abstract away some details and encode the representation of a computation as a tree, similarly to what algebras with a single binary operation [18] or suspended algebraic effects [30]. DAs can be interpreted as data and the resulting of consuming such data into a resulting value.

```
structure TraceTree (α β γ : Type) where
  mk :: (data : BinaryTree α β) (res : γ)
```

where `BinaryTree` are binary trees with leaves of type α and nodes having information of type β . We also implicitly (and optimistically) assume the following property for DAs:

```
def implicit_assumption (comp : ComputationTree α β γ) (leaf_interpretation : α → γ)
  (node_interpretation : β → γ → γ → γ) : Prop
:= fold leaf_interpretation node_interpretation comp.data = comp.res
```

The goal of L2 optimistic rollups is to avoid as much computation as possible, and thus, the `implicit_assumption` is never *executed* but it has to be guaranteed by the system. Then, faulty DAs are guaranteed to be discovered, and fault-proofs generated.

Hashes – Authenticated Data Structures

In fraud-proofs verifiable blame can be assigned to players. The way this is done in RDoC is by creating computations using authenticated data structure (i.e. Merkle trees). When players post DAs, they are committing to a strategy, which states that following the computation the claimed result is obtained. The computation is encoded as a tree with no information, the skeleton of a computation, and the result is encoded as the hash that results from hashing the tree itself. The proposing player can be asked to expose the data in the computation and incrementally validate the data provided (by hashing it) against the submitted hash. In Lean, this means that our trace tree is of the form:

```
abbrev DAs (H : Type) := TraceTree Unit Unit H
```

⁷ When the data is not public, this game can be employed as a very expensive data retrieval mechanism.

Using the Lean class system, we can have a function hashing elements supporting a binary operation combining them:

```
class Hash (α H : Type) where mhash : α -> H

class HashMagma (H : Type) where comb : H -> H -> H
```

However, when a proposing players propose data behind a hash, we assume that they cannot provide a different element that collides with the original element, that is, that hashes have no collisions. Technically, we need to propagate the non-colliding condition to the binary operator as follows.

```
class CollResistant (α H : Type) [op : Hash α H] where
  noCollisions : forall (a b : α), a ≠ b -> op.mhash a ≠ op.mhash b

class SLawFulHash (H : Type) [m : HashMagma H] where
  neqLeft : forall (a1 a2 b1 b2 : H), a1 ≠ a2 -> m.comb a1 b1 ≠ m.comb a2 b2
  neqRight : forall (a1 a2 b1 b2 : H), b1 ≠ b2 -> m.comb a1 b1 ≠ m.comb a2 b2
```

This way of presenting the computational part through classes `Hash` and `HashMagma` and assumptions through classes `CollResistant` and `SLawFulHash` is very useful when having computations on one side and proof on the other. When defining games, functions and executing strategies, we work with their computational counterpart. When proving theorems about such functions and games, we need to also include our theoretical assumptions.

3.2 Generic Arbitration Games

Arbitration games are turn-based two-player games over DAs. One player reveals information, in this case hashes, while the other chooses which path to follow to continued the exploration of the trace tree proposed.

We define the following general game over binary trees abstracting types away:

```
inductive ChooserMoves where | Now | ContLeft | ContRight
def treeCompArbGame {α α' β γ : Type}
  -- Game Mechanics
  (leafCondition : α -> α' -> γ -> Winner)
  (midCondition : β -> γ -> γ -> γ -> Winner)
  -- Public Information
  (da : ComputationTree α β γ)
  -- Players
  (revealer : BinaryTree (Option α') (Option (γ × γ)))
  (chooser : BinaryTree Unit ((γ × γ × γ) -> Option ChooserMoves))
  : Winner := match da.data, revealer with ...
```

Here, `Winner` is just a two element type to say which player has won, `TraceTree` is the DA defined before, and `ChooserMoves` describes choosers actions either challenge current assertion or chooses what path to take, left or right in binary trees. Players can choose not to play, modeled using Lean `Option` type.

The function `treeCompArbGame` pattern matches the arena in the DA, the player `revealer` and `chooser`, and feeds the chooser function with the information provided by the revealer. Depending on the result of the chooser, the game continues creating a new DA with the information provided by the revealer or the condition `midCondition` is triggered and one player wins⁸. Each pattern matching involving players represent a player interaction with the blockchain. A player that failing to fulfill their part loses the game.

⁸ Full implementation in file `GenericTree.lean`: <https://gitlab.software.imdea.org/martin.ceresa/leanfpgames/-/blob/master/FraudProof/Games/GenericTree.lean>.

Because our arena is a tree and due the nature of Merkle trees, there are two ways to be fraudulent in this scheme. One is to provide faulty information, for example wrong small step in a transactions execution. The other is to give a faulty tree where some elements do not hash to their parents, that is, the computation itself is faulty. The first is what we call `leafCondition`, which is a condition on the leaves of the arena. The second is an intermediary condition, `midCondition`, that is, a condition over the nodes. When challengers engage in arbitration games, they try to find which nodes or leaves violate these conditions. Once we define our two conditions, we define a game.

Valid Merkle Tree Game. Instantiating the above game, we have the following game:

```
def cond_hash_elem {H α : Type} [BEq H] [h : Hash α H]
  (leaf: H) (rev : α) (res : H)
  : Bool := h.mhash rev == res && leaf == res

def cond_hash { H : Type } [BEq H] [mag : HashMagma H] (res l r : H)
  : Bool := mag.comb l r == res

abbrev valid_Merkle_tree := treeCompArbGame cond_hash_elem cond_hash
```

Once the committing hash has been established as well-constructed, agents can play a different (more efficient) game challenging the validity of the claim by pinpointing an invalid element. Depending on the context, we have different validity test, e.g. no duplicated operations or small-step validity. The game then is reduced to show that there is an invalid element in the data proposed by proving that the invalid element belongs to the current DA Merkle tree. Since paths in Merkle trees can be seen as trace trees, we can play this game using `treeCompArbGame`. In Section 4, we define an alternative game for membership which is logarithmic in the length of the path.

3.3 Winning Strategies

We focus now on proving that honest players can always win. Depending on their role, players have different winning conditions. Players proposing DAs have the *optimistic advantage*: they are right unless proven otherwise. Honest players proposing DAs know the data they used to create, move first and, if required, defend their claim against all possible challengers, honest or otherwise. When it comes to challenging players, we can build winning strategies against dishonest proposers that submitted a faulty DA. Because of the optimistic advantage, honest challenging players – knowing the data behind the DA – only challenge when they detect there is an invalid DA.

From the definition of `treeCombArbGame`, we get that revealer players (in L2, the ones proposing DAs) can be challenged at any moment, and thus they need to win all possible challenges to make the DA consolidate. Therefore, revealer players, for a given DA, need to win all possible conditions (leaf and node conditions).

Challenger players follow the same reasoning, but in this case, they only challenge when they know they are going to win. In our games, challenger players act as choosers, choosing which path to take. This means they need to know the missing data in the computation tree before playing.

The following definition states that a player strategy follows a given DA and that leaf and node conditions are met:

```
def tree_comp_winning_conditions {α α' β γ : Type}
  -- Game Mechanics
  (leafCondition : α -> α' -> γ -> Prop)
```

```

(midCondition :  $\beta \rightarrow \gamma \rightarrow \gamma \rightarrow \gamma \rightarrow \text{Prop}$ )
-- Public Information
(da : ComputationTree  $\alpha \beta \gamma$ )
(player : BinaryTree (Option  $\alpha'$ ) (Option ( $\gamma \times \gamma$ )))
: Prop :=
match da.data , player with
| .leaf a' , .leaf (.some a) => leafCondition a' a da.res
| .node b' gl gr , .node (.some b) pl pr =>
  midCondition b' da.res b.1 b.2
  ^ tree_comp_winning_conditions leafCondition midCondition < gl , b.1 > pl
  ^ tree_comp_winning_conditions leafCondition midCondition < gr , b.2 > pr
| _ , _ => False

```

For revealer players, if they know the data and computed the final result properly – that is, `tree_comp_winning_conditions` is true – they win against all chooser players. In L2 terms, revealers can defend their claim against all possible dishonest agents⁹.

```

theorem winning_prop_hashes {H  $\alpha$  : Type}
[DecidableEq H]
[Hash  $\alpha$  H] [HashMagma H]
-- Public Information
(da : ComputationTree H Unit H)
-- Players
(revealer : BinaryTree (Option  $\alpha$ ) (Option (H  $\times$  H)))
(good_revealer : revealer_winning_condition
  cond_hash_elem (fun _ => cond_hash) da revealer)
: forall (chooser : BinaryTree Unit ((H  $\times$  H  $\times$  H)  $\rightarrow$  Option ChooserMoves)),
  valid_merkle_tree da revealer chooser = Player.Proposer
:= winning_proposer_wins _ _ da revealer good_revealer

```

When it comes to challengers, first we need to generate the strategy and then prove a similar theorem but working over the assumption that the challenger knows the data and that the computation leads to a different hash.

```

theorem winning_gen_chooser {H  $\alpha$  : Type}
[hash : Hash  $\alpha$  H] [HashMagma H] [DecidableEq H]
-- Public Information
(pub_data : BinaryTree H Unit)
-- Players
(revealer : BinaryTree (Option  $\alpha$ ) (Option (H  $\times$  H)))(rev_res : H)
(chooser : BinaryTree (Option  $\alpha$ ) (Option (H  $\times$  H)))(ch_res : H)
(good_chooser : winning_condition_player cond_hash_elem cond_hash
  (const id) < pub_data , ch_res > chooser)
(hneq :  $\neg$  rev_res = ch_res)
: valid_merkle_tree < pub_data , rev_res >
  revealer (chooser.map (fun _ => ())) gen_chooser_opt)
= Player.Chooser := by ...

```

The above proof needs to be sure that when the revealer provides the data (as a hash) the corresponding element is publicly known and no other element can be produced with the same hash. In this presentation, we are using `DecidableEq H` hiding this fact¹⁰. The same goes for intermediary steps, the chooser player needs to have some guarantee when choosing paths, because otherwise the revealer may produce elements hashing to the same hash, invalidating the challenge.

⁹ Honest challengers will not challenge honest DAs.

¹⁰ See file `DataStructures/Hash.lean` at <https://gitlab.software.imdea.org/martin.ceresa/leanfpgames/-/raw/master/FraudProof/DataStructures/Hash.lean>.

4 Membership Games for Merkle Trees

Specific fraud proof games have also been employed [12] to guarantee that Layer 2 sequencers propose valid batches of transactions. These fraud proof games do not verse about the outcome of generic computations (as in RDoC) but, they correspond instead to concrete programs that evaluate certain aspects of data-types, in particular of batches of transactions encoded as Merkle trees. The most fundamental building block for such games is a membership game that allows to prove that a given element is in the batch proposed. In turn, membership games can be used by a challenger to show that the batch is illegal because it contains repeated elements (providing proofs of the same element in two different positions). We detail membership games in the rest of this section.

The definition of a claim is similar to the DAs in the previous sections, but now the DA has the form of a Merkle tree path (instead of a trace tree), and additional indication of the source and destination hashes.

```
inductive Direction where | Left : Direction | Right : Direction
-- Sequence of length |n| indicating Left or Right
abbrev Path (n : Nat) := Sequence n Direction

structure ElemInMTree (H : Type) (n : Nat) where
  path : Path n
  src  : H
  dst  : H
```

The above claim encodes the idea that if hash `src` is a the root of Merkle tree, there is a path `path` of length `n` from `src` to `dst`. There are two (equivalent¹¹) games we can play: a path from `elem` to `dst` and a path from `dst` to `elem`. For simplicity, we focus on paths starting from the element up to the root. The implicit property is the proof of an element belonging to a Merkle tree, which is the sequence of intermediate hashes lead to the root.

In this game, the arena is a list where one player reveals the missing data while the other either chooses to challenge the current step or continues up on the path. The missing data is (1) the next hash in the path from the current element and (2) the hash used to compute it— in the case of Merkle trees, this encodes a Merkle subtree. However, instead of defining a new game, we can use our previous definitions. We map the arena (a path) to a trace tree, and to map the players strategies, the usual way to map a sequence into a tree with one deep child and the other child being a leaf. We use the arena guiding their strategies indicating which child belongs to the path and which one is an unexplored subtree. We map the move `Continue` depending on the side `Direction` dictated by the path to `Left` or `Right`. Conditions check (if required) that hashes match, i.e. if the proposer player gave a subpath whose last element is the next hash in the path to the root.

```
inductive ChooserSmp : Type where | Now | Continue
def elem_in_tree_forward_gentree {H : Type}
  [BEq H] [mag : HashMagma H] {n : Nat} (da : ElemInMTree H n)
  (proposer : Sequence n (Option (H × H)))
  (chooser : Sequence n (H × H × H → Option ChooserSmp))
  := treeCompArbGame leaf_condition_range mid_condition_range_one_step_forward
    {data := skl_to_tree da.data, res := da.res}
    (build_proposer' da.res.1 da.data proposer)
    (build_chooser' da.data chooser)
```

¹¹We prove them in Lean as there is a transformation to go and come back from both games resulting in the same player winning.

Logarithmic FraudProof

We now introduce an alternative more efficient membership game, which requires a logarithmic number of moves on the length of the path. To prove both games equivalent, we define a transformation of the arena and prove that corresponding players that know the data win one game if and only if they win the other game. This game mimics the bisection game used in Arbitrum.

The linear and logarithmic games are different from the point-of-view of the challenger. When building the fraud-proof, we are verifying the existence of a path from a leaf to the root. In the linear game, we ask to the revealer to reveal each element along the path verifying that it is correct by checking that hashes match. In the logarithmic game, we ask for the hash of the element in the middle of the path, effectively bisecting the path in two, and then, choosing which half to challenge next. The main difference is that to choose whether to challenge the upper or lower sub-paths, the challenger needs to know the path upfront. On the other hand, in the linear games, the challenger does not need to know the path and can run the check and challenge until hashes do not match. Honest challengers playing the linear game only need to know that the path is invalid. In fact, the path can be provided fully by the proposer and checked in one shot of computation (requiring to check a linear number of hashes in the size of path).

In the logarithmic games, the revealer – instead of decomposing the parent hash into two children hashes – given two hashes (corresponding to the extremes of the path), proposes the hash in the middle of the path.

First, we transform the arena. From a sequence of `Direction` of length 2^n for some n , we build a tree having as leaves the sequence (in the same order) and no information at the nodes.

```
def built_up_arena {n : Nat} : Sequence (2^n) Direction -> BinaryTree Direction Unit
:= gen_info_perfect_tree (seq_constant ())
```

Then, we transform the strategy of the revealer in a similar way. From the missing data, we can compute all intermediary hashes along the path (spine hashes) and the auxiliary hashes (representing Merkle subtrees in the original computation tree). We take all spine hashes but the last (the Merkle tree root hash) and place them at the nodes and subtree hashes at the leaves.

```
def forward_proposer_to_tree {H : Type} {n : Nat}
  (prop : Sequence (2^n) (H × H)) : BinaryTree H H
:= gen_info_perfect_tree
  ( Fin.init -- Drop last hash (top hash [forward])
    $ sequence_coerce (by have pg := @pow_gt_zero n; omega)
    $ seqMap (fun p => p.fst) prop) -- Spine hashes
  ( seqMap (fun p => p.snd) prop) -- leaves matching subtrees
```

Finally, we show that the above transformations map linear revealer winning players into winning logarithmic revealer players.

```
theorem proposer_winning_mod_forward {H : Type} {lgn : Nat}
  [DecidableEq H] [HashMagma H] (da : ElemInTree (2^lgn) H)
  (proposer : Sequence (2^lgn) (H × H))
  (wProp : elem_in_revealer_winning_condition_forward
    da (seqMap (.Next) proposer))
  (chooser : BinaryTree Unit (Range H -> H -> Option ChooserMoves))
  : spl_game ({data := built_up_arena da.data , res := da.mtree})
    ( BinaryTree.map .some .some $ forward_proposer_to_tree proposer)
    chooser = Player.Proposer := by ...
```

Where game `spl_game` is essentially the same as `treeCompArbGame` but instead of disclosing a pair of hashes from a hash, the revealer is ask to provide a hash in the middle of two hashes plus there is no node conditions (if triggered the proposer player wins the game.) In this game, both players have to play until a leaf is reached, since there is no way to know that intermediary steps are correct. Intuitively, we are not following small verifiable steps, but jumping around in the trace tree. To build fraud-proofs is enough, since we only need one witness to show that the computation is invalid.

When it comes to the challenger, we do something similar to what we did before. The main difference is the winning condition. We cannot transform choosers as defined in the previous games. We used functions since they have to handle all possible hashes revealed by the other player. Therefore, we define choosers knowing the data and generate their strategies.

```

theorem range_choser_wins {H : Type}
  [BEq H] [LawfulBEq H] [HashMagma H] [hash_props : SLawFulHash H]
  -- DA elements
  (comp_skeleton : BinaryTree SkElem Unit)
  (input_rev input_ch : H)(output : H)
  -- Players says that path starts at different places
  (hneq : ¬ input_rev = input_ch)
  -- Players
  (revealer : BinaryTree (Option H) (Option H))
  (chooser : BinaryTree H H)
  -- Chooser computation is fold plus invariants.
  (chooser_wise : knowing comp_skeleton chooser input_ch output)
  : spl_game { data:= comp_skeleton , res := (input_rev , output) }
    reveler (gen_to_fun_chooser (BinaryTree.map .some .some chooser))
    = Player.Chooser := by ...

```

The above theorem proves that honest choosers win, but does not say anything about how long games are. In the case the arena is a binary complete tree, finding the fraud-proof is logarithmic in the path length. Predicate `knowing` states that the data the chooser has faithfully describes a path from hash `input_ch` to hash `output`, similar to `winning_condition_player`. What it is missing is to connect the winning chooser players in the linear game with the above logarithmic game, we leave that to future work.

5 Example: A Simple Protocol

Recent work [12] introduces arbitration games to guarantee properties of batches of transactions proposed by sequencers. These arbitration games now correspond to the execution of concrete specific algorithms known a-priori and not (as in RDoC) to games where one must reason about arbitrary traces of computation from a universal machine.

In [12], a block b is proven to be valid if and only if (1) all transactions in b are valid (which can be check locally by a function `valid`), (2) there are no duplicates transactions in b , (3) no transaction appears in a previous accepted block [12, Section 3.2 (*certified legal batch tag*)]. The definition characterizes the notion of validity completely, and thus, we can also detect invalid blocks by detecting when (at least) one of the above conditions does not hold playing specific games. To see the application of our approach, we focus on the first two properties. The third one can be modeled by encoding the history of accepted blocks as a large Merkle tree or multiple signed Merkle trees, and we leave it as future work.

In our Lean library, we define a structure `Valid_DA` mapping the definition of a valid block as the first two properties plus the correctness of the Merkle tree.

```

structure Valid_DA {α H : Type} [DecidableEq α] [Hash α H] [HashMagma H]
  (data : BinaryTree α) (mk : H) (P : α → Bool) where
  -- Merkle tree is correct.
  MkTree : data.hash_BTree = mk
  -- All Elements are valid.
  ValidElems : data.fold P Bool.and = true
  -- There are no duplicates.
  NoDup : List.Nodup data.toList

```

We then model the interaction between the two players by their actions. Proposing players generate the DA from a sequence of values (as a tree) and also provides all their strategies before hand. Because of the two kinds of games that can be played there is one data-availability strategy and one strategy for each possible paths. Outside of this model, strategies are played interactively, but we do not have reactive components in our model.

```

structure P1_Actions (α H : Type) : Type
where
  da : BinaryTree α Unit × H
  dac_str : BinaryTree (Option α) (Option (H × H))
  gen_elem_str : {n : Nat} → Path n → (Sequence n (Option (H × H))) × Option α

```

The player choosing and challenging dishonest claims have one action per invalid property of the DA. In this example, and because of simplicity, we show the linear games.

```

inductive P2_Actions (α H : Type) : Type where
  -- Player 2 challenging the Merkle tree formation
  | DAC (str : BinaryTree Unit ((H × H × H) → Option ChooserMoves))
  -- Player 2 accepts the Markle is well form but there is an invalid element
  | Invalid {n : Nat} (p : α) (path : Path n)
    (str : Sequence n ((H × H × H) → Option ChooserSmp))
  -- Player 2 accepts the Markle is well form and all elements are valid
  -- but there is a repeated element
  | Duplicate (n m : Nat) -- There are two paths
    (path_p : Path n) (path_q : Path m)
    -- Strategies to force proposer to show elements.
    (str_p : Sequence n ((H × H × H) → Option ChooserSmp))
    (str_q : Sequence m ((H × H × H) → Option ChooserSmp))
  -- Player 2 accepts the DA proposed
  | Ok

```

Now we have all the pieces to define the protocol. The protocol is simply an intermediary mechanism invoking games when required and indicates when a proposal should be accepted or no. When implemented in the real-world, this is implemented in a smart contract governing the computational aspects of the system. Here we show a fragment of the protocol when the chooser challenges the creation of the Merkle tree¹².

```

def linear_l2_protocol {α H : Type} [BEq α] [BEq H] [o : Hash α H] [HashMagma H]
  (val_fun : α → Bool) (playerOne : P1_Actions α H)
  (playerTwo : (BTree α × H) → P2_Actions α H) : Bool
:= match playerTwo playerOne.da with | .DAC ch_str =>
  -- Challenging Sequencer (Merkle tree is not correct)
  match data_challenge_game
    < playerOne.da.fst.map o.mhash , playerOne.da.snd >
    playerOne.dac_str ch_str with
  | .Proposer => true
  | .Chooser => false
  ...

```

Finally, we prove only valid blocks survive the protocol in presence of honest choosers.

¹²The rest of the protocol can be found in the file “L2.lean” at <https://gitlab.software.imdea.org/martin.ceresa/leanfpgames/-/blob/master/FraudProof/L2.lean>

```

theorem honest_choser_valid {α H}
  [BEq H] [LawfulBEq H] [DecidableEq α]
  [o : Hash α H] [m : HashMagma H] [InjectiveHash α H] [InjectiveMagma H]
  (P : α → Bool) (p1 : P1_Actions α H)
  : linear_l2_protocol P p1 (honest_choser_val_fun) ↔ valid_da p1.da P

```

where `valid_da` states that all properties are valid, i.e. building a `Valid_DA`.

6 Conclusions and Future Work

Arbitration games in L2 Optimist Rollups systems are deterrents to prevent fraudulent blocks to consolidate. Such systems rely on the argument that if malicious agents lie, they are caught and penalized. They also rely on agents knowing the public data.

We defined the concepts of DAs, games, players, winning strategies and transformations between games in Lean4. Moreover, we proved that honest players knowing the data, have winning strategies to defend honest claims and to challenge dishonest claims. This is (to the best of our knowledge) the first work to mechanize and prove winning strategies for honest players in the computational model of L2 optimistic rollups. We explored simple notions of equivalence between games: same winning players and mapping winning strategies to winning strategies. Finally, we defined a simple Layer2 protocol and proved it correct.

As future work, we propose the following paths.

Polish the library. All proofs not provided due to the limited space are proved. The library is a proof-of-concept, so the first step is to refactor it to get a cleaner code base. Additionally, with the intuition gained, we want to properly define games borrowing formal concepts from Combinatorial Game Theory and Operational Game Semantics [8].

Generalization. The main idea of DAs is to hide data and computation and to use Merkle trees to build (verified) blaming chains, fraud proofs. In this work, we focused on formalizing these concepts on trees, but we plan on explore different authenticated-data structures [24]. Another generalization is to have *container* data-types as the arena, computations as *folds* and DAs as predicates over these computations[1].

Game Description and Interaction Language. RDoC performs arbitration over the trace of traces of computation from arbitrary programs. However, we can play arbitration games over higher abstractions or programs fixed a-priori. If we are able to decompose validity of bigger DAs into smaller ones, we may be able to play specific games over different algorithms more efficiently. Once players accept the hash to be a Merkle tree, they can engage into specific games. Game `elem_in_tree_is_invalid(path, hash)` challenges the agent posting the DA that element in `path` is invalid. Game `elem_in_tree_is_twice(path_1, path_2, hash)` challenges the agent posting the DA that element in `path_1` is the same as the one in `path_2`, and thus, the block is invalid for repeating elements. To describe all these different situations, we would like to have a nice game language, probably a subset of the *Game Description Language (GDL)*. Ideally, we want to verify the basic components of these games and derive proofs to the more general games.

References

- 1 Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005. Applied Semantics: Selected Topics. doi:10.1016/j.tcs.2005.06.002.

- 2 Wolfgang Ahrendt and Richard Bubel. Functional verification of smart contracts via strong data integrity. In *Proc. of ISO/IEC JTC1 SC22 WG2 N15478*, number 12478 in LNCS, pages 9–24. Springer, 2020. doi:10.1007/978-3-030-61467-6_2.
- 3 Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. ConCert: a smart contract certification framework in Coq. In *Proc. of the 9th ACM SIGPLAN Int'l Conf. on Certified Programs and Proofs (CPP'20)*, pages 215–218. ACM, 2020. doi:10.1145/3372885.3373829.
- 4 Shaun Azzopardi, Joshua Ellul, and Gordon J. Pace. Monitoring smart contracts: ContractLarva and open challenges beyond. In *Proc. of the 18th International Conference on Runtime Verification (RV'18)*, volume 11237 of LNCS, pages 113–137. Springer, 2018. doi:10.1007/978-3-030-03769-7_8.
- 5 Bruno Bernardo, Raphaël Cauderlier, Zhenlei Hu, Basile Pesin, and Julien Tesson. Mi-Chocoq, a framework for certifying Tezos smart contracts. In *Proc. of the FM 2019 International Workshops, Part I*, volume 12232 of LNCS, pages 368–379. Springer, 2019. doi:10.1007/978-3-030-54994-7_28.
- 6 Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fourneta, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella Béguelin. Formal verification of smart contracts: Short paper. In *Proc. of Workshop on Programming Languages and Analysis for Security (PLAS@CCS'16)*, pages 91–96. ACM, 2016. doi:10.1145/2993600.2993611.
- 7 Peio Borthelle, Tom Hirschowitz, Guilhem Jaber, and Yannick Zakowski. Games and strategies using coinductive types. In *International Conference on Types for Proofs and Programs*, 2023.
- 8 Peio Borthelle, Tom Hirschowitz, Guilhem Jaber, and Yannick Zakowski. An abstract, certified account of operational game semantics. In *European Symposium on Programming*, (to appear in) 2025.
- 9 Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform, 2014. Accessed: May 6, 2025. URL: <https://github.com/ethereum/wiki/wiki/White-Paper>.
- 10 Ran Canetti, Ben Riva, and Guy N. Rothblum. Refereed delegation of computation. *Information and Computation*, 226:16–36, 2013. doi:10.1016/J.IC.2013.03.003.
- 11 Margarita Capretto, Martín Ceresa, and César Sánchez. Transaction monitoring of smart contracts. In Thao Dang and Volker Stolz, editors, *Proc. of the 22nd Int'l Conf. on Runtime Verification (RV'22)*, volume 13498 of LNCS, pages 162–180. Springer, 2022. doi:10.1007/978-3-031-17196-3_9.
- 12 Margarita Capretto, Martín Ceresa, Antonio Fernández Anta, Pedro Moreno-Sánchez, and César Sánchez. A decentralized sequencer and data availability committee for rollups using set consensus, 2025. doi:10.48550/arXiv.2503.05451.
- 13 Martín Ceresa and César Sánchez. Fraud Proof Games. Software, version 1., swbId: `swb:1:dir:761ba38f606c1b4a0a9e202e6518d092d51ff381` (visited on 2025-05-06). URL: <https://gitlab.software.imdea.org/martin.ceresa/leanfpgames>, doi:10.4230/artifacts.23003.
- 14 Martín Ceresa and César Sánchez. Multi: A Formal Playground for Multi-Smart Contract Interaction. In Zaynah Dargaye and Clara Schneidewind, editors, *4th International Workshop on Formal Methods for Blockchains (FMBC 2022)*, volume 105 of *Open Access Series in Informatics (OASICS)*, pages 5:1–5:16, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASICS.FMBC.2022.5.
- 15 Sylvain Conchon, Alexandrina Korneva, and Fatiha Zaïdi. Verifying smart contracts with Cubicle. In *Proc. of the 1st Workshop on Formal Methods for Blockchains (FMBC'19)*, volume 12232 of LNCS, pages 312–324. Springer, 2019. doi:10.1007/978-3-030-54994-7_23.
- 16 Tyler Crain, Christopher Natoli, and Vincent Gramoli. Redbelly: A secure, fair and scalable open blockchain. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 466–483, 2021. doi:10.1109/SP40001.2021.00087.

- 17 Joshua Ellul and Gordon J. Pace. Runtime verification of Ethereum smart contracts. In *Proc. of the 14th European Dependable Computing Conference (EDCC'18)*, pages 158–163. IEEE Computer Society, 2018. doi:10.1109/EDCC.2018.00036.
- 18 J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *J. ACM*, 24(1):68–95, January 1977. doi:10.1145/321992.321997.
- 19 Abdelatif Hafid, Abdelhakim Senhaji Hafid, and Mustapha Samih. Scaling blockchains: A comprehensive survey. *IEEE Access*, 8:125244–125262, 2020. doi:10.1109/ACCESS.2020.3007251.
- 20 Peter Hancock and Pierre Hyvernat. Programming interfaces and basic topology. *Annals of Pure and Applied Logic*, 137(1):189–239, 2006. doi:10.1016/j.apal.2005.05.022.
- 21 Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium*, pages 1353–1370. USENIX Assoc., 2018. URL: <https://www.usenix.org/conference/usenixsecurity18/presentation/kalodner>.
- 22 Ryan Lavin, Xuekai Liu, Hardhik Mohanty, Logan Norman, Giovanni Zaarour, and Bhaskar Krishnamachari. A survey on the applications of zero-knowledge proofs, 2024. doi:10.48550/arXiv.2408.00243.
- 23 Ao Li, Jemin Andrew Choi, and an. Long. Securing smart contract with runtime validation. In *Proc. of ACM PLDI'20*, pages 438–453. ACM, 2020. doi:10.1145/3385412.3385982.
- 24 Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. *SIGPLAN Not.*, 49(1):411–423, January 2014. doi:10.1145/2578855.2535851.
- 25 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, December 2008. Accessed: May 6, 2025. URL: <https://bitcoin.org/bitcoin.pdf>.
- 26 Zeinab Nehaï and François Bobot. Deductive proof of industrial smart contracts using Why3. In *Proc. of the 1st Workshop on Formal Methods for Blockchains (FMBC'19)*, volume 12232 of *LNCS*, pages 299–311. Springer, 2019. doi:10.1007/978-3-030-54994-7_22.
- 27 Sebastiano Peluso, Alexandru Turcu, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Making fast consensus generally faster. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 156–167, 2016. doi:10.1109/DSN.2016.23.
- 28 Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachler-Cohen, and Martin Vechev. VerX: Safety verification of smart contracts. In *Proc of the 41st IEEE Symp. on Security and Privacy (S&P'20)*, pages 1661–1677. IEEE, 2020. doi:10.1109/SP40000.2020.00024.
- 29 Daian Phil. Analysis of the DAO exploit, 2016. URL: <https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/>.
- 30 Matija Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic Notes in Theoretical Computer Science*, 319:19–35, 2015. The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI). doi:10.1016/j.entcs.2015.12.003.
- 31 Jonas Schiffel, Wolfgang Ahrendt, Bernhard Beckert, and Richard Bubel. Formal analysis of smart contracts: Applying the KeY system. In Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, and Mattias Ulbrich, editors, *Deductive Software Verification: Future Perspectives - Reflections on the Occasion of 20 Years of KeY*, volume 12345 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 2020. doi:10.1007/978-3-030-64354-6_8.
- 32 Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a smart contract intermediate-level Language. *CoRR*, abs/1801.00687, 2018. arXiv:1801.00687.
- 33 Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dilig. SmartPulse: Automated checking of temporal properties in smart contracts. In *Proc. of the 42nd IEEE Symp. on Security and Privacy (S&P'21)*. IEEE, May 2021. URL: <https://www.microsoft.com/en-us/research/publication/smartpulse-automated-checking-of-temporal-properties-in-smart-contracts/>.
- 34 Nick Szabo. Smart contracts: Building blocks for digital markets. *Extropy*, 16, 1996.

- 35 Louis Tremblay Thibault, Tom Sarry, and Abdelhakim Senhaji Hafid. Blockchain scaling using rollups: A comprehensive survey. *IEEE Access*, 10:93039–93054, 2022. doi:10.1109/ACCESS.2022.3200051.
- 36 Gang Wang, Zhijie Jerry Shi, Mark Nixon, and Song Han. SoK: Sharding on blockchain. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 41–61, 2019. doi:10.1145/3318041.3355457.
- 37 Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.
- 38 Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:10.1145/3371119.

Validity, Liquidity, and Fidelity: Formal Verification for Smart Contracts in Cardano

Tudor Ferariu 

University of Edinburgh, UK

Philip Wadler 

University of Edinburgh, UK

Orestis Melkonian 

Input Output, London, UK

Abstract

Good news for researchers in formal verification: smart contracts regularly suffer exploits such as the DAO bug, which lost the equivalent of 60 million USD on Ethereum. This makes a strong case for applying formal methods to guarantee essential properties.

Which properties would we like to prove? Most previous studies focus on contract-specific properties that do not generalize to a wide class of smart contracts. There is currently no commonly agreed upon list of properties to use as a starting point in writing a formal specification.

We propose three properties that we believe are relevant to all smart contracts: Validity, Liquidity, and Fidelity. Focusing on the concrete case of the Cardano platform, we show how these properties stop exploits similar to the DAO bug, as well as preventing other common issues such as the locking of funds and double satisfaction.

We model an account simulation, a multi-signature wallet, and an order book decentralized exchange, as example smart contract specifications using state transition systems in the Agda proof assistant. We formalize the above properties and prove they hold for the models. The models are then separately proven to be functionally equivalent to a validator implementation in Agda, which is translated to Haskell using `agda2hs`. The Haskell code can then be compiled and put on the Cardano blockchain directly. We use the Cardano Node Emulator to run property-based tests and confirm that our validator works correctly.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Logic and verification; Theory of computation → Program specifications

Keywords and phrases blockchain, Agda, UTxO, EUTxO, smart contract, formal verification, specification, transition systems, Cardano

Digital Object Identifier 10.4230/OASICS.FMBC.2025.6

Supplementary Material *Software (Zenodo archive):* <https://doi.org/10.5281/zenodo.15118668> [39]

Software (Source Code): <https://github.com/tferariu/agda2plinth> [40]
archived at `swb:1:dir:03dd5b9f802117f3b5cccf0ed56d4214fee41238`

1 Introduction

Millions of dollars worth of value are managed by smart contracts. Alas, as made famous by numerous exploits, such contracts often contain flaws. Many such flaws might be eliminated by the application of formal methods, but the costs are high. Hence, most developers make do with property-based tests [50, 23] and auditing [43]. Formal methods might be less expensive to apply if we could identify a few common properties that most or all contracts should share. Such general properties are not widely studied.



© Tudor Ferariu, Philip Wadler, and Orestis Melkonian;
licensed under Creative Commons License CC-BY 4.0

6th International Workshop on Formal Methods for Blockchains (FMBC 2025).

Editors: Diego Marmsoler and Meng Xu; Article No. 6; pp. 6:1–6:21

OpenAccess Series in Informatics



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

Here we make a first step, by presenting three properties we believe generalize to a wide variety of smart contracts. We also introduce a novel technique for proving such properties, based on factoring smart contracts into a specification (written as inference rules) and a validator (written as a boolean-valued function) and demonstrating their equivalence. We focus on the UTxO-based blockchain Cardano.

We investigate three main properties:

- **Validity** : No operation allowed by the contract can take it to an invalid state.
- **Liquidity**: All currency can be eventually extracted from the contract.
- **Fidelity** : The actual value locked in the contract is the same as its internal value.

Many real-life smart contracts fail these properties. The Aku Dreams Project [2, 9] and the Perfect Finance contract [10, 11] violated Liquidity, permanently locking 34 million dollars and 1 million dollars respectively. The DAO exploit [37] and the Wormhole bug [20, 8] violated Fidelity, allowing the theft of 60 million dollars and over 320 million dollars, respectively.

Our specifications, validators, and the proof of their equivalence, are written using the Agda proof assistant [52]. A proof assistant provides a higher standard of rigour than property-based tests or auditing. We export the Agda code of the implementation to the Cardano target language, Haskell, using `agda2hs` [35]. Even though `agda2hs` is not formally verified itself,¹ we still get a lot more confidence that the code used in our proofs aligns with the on-chain code compared to writing the Haskell code by hand. Finally, we test the exported code with an emulator and property-based tests for an extra layer of confirmation.

Due to limitations of UTxO-based blockchains, a smart contract has no control over its initial state. The most common solution to this problem is using the minting policy of a Thread Token (Chakravarty et. al., 2020, [32]) to ensure the validity of the initial state. Since this mechanism comes with a *meta-theorem* that is proven correct for all possible instantiations of the notion of initiality for each contract, we assume the initial state is valid by construction in our work so as to focus solely on the consequent execution steps of the contract after initialisation. The main contributions of this paper are:

- Providing an example smart contract and a way to model its runtime as a state transition system in Section 2 and Section 3 respectively.
- Describing our three central properties in Section 4.
- Explaining how we implement the contract and transfer the properties of the model to the implementation in Section 5 and Section 6 respectively.
- Combining all these steps to obtain a practical pipeline for mechanized formal verification using Agda in Section 7.
- Relating our work to the overarching issue of double satisfaction in Section 8.

Full files and any other relevant code can be accessed at the following repository [39]:

<https://tferariu.github.io/agda2plinth>

2 Example: simulating accounts on UTxOs

Smart contract development for UTxO-based blockchains has some advantages, as exploits such as double spending, re-entrancy, and replay attacks are not possible (Guggenberger et. al., 2021, [42]). Nonetheless, undesirable behaviour might occur when dealing with the inputs and outputs of a transaction incorrectly.

¹ It is expected that Agda extraction will eventually be formally verified in the future, akin to what MetaCoq [61] has delivered for the more mature ecosystem of the Rocq proof assistant [62].

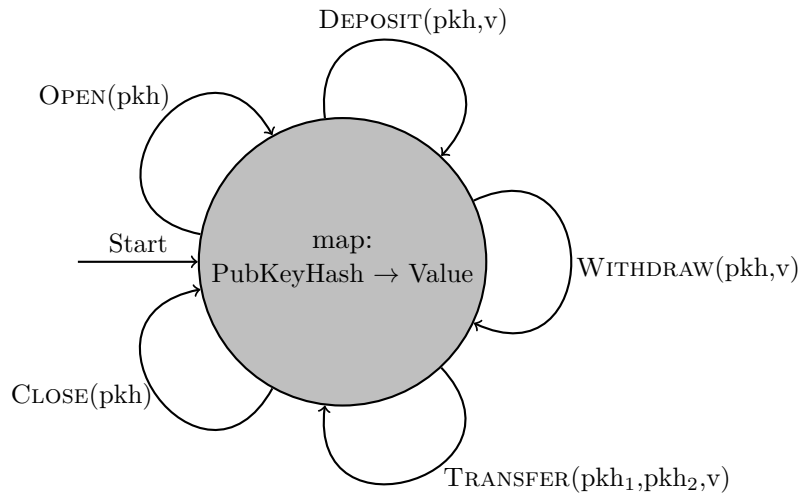
The Cardano blockchain is based on EUTxOs (Chakravarty et. al., 2020, [33]), UTxOs “extended” with a datum that can carry stateful information. In contrast with Ethereum and other account-based systems, Cardano benefits from completely predictable fees and outputs [26]. On Cardano, smart contracts take the form of validators; Boolean-valued functions that decide whether the UTxO may be spent as part of a transaction.

Account-based blockchains such as Tron, EOS, and Tezos [41] are widely used and popular. Chief among them is Ethereum [31], which ranks highest for Total Value Locked (TVL) [7] among contemporary blockchains. Since we are primarily investigating the UTxO-based Cardano blockchain, a simple smart contract simulating accounts serves as a good example for showcasing our properties and proof process.

Chakravarty et. al. (2020, [33]), uses the formalism of state machines [49] to model stateful computation for Cardano validators on the EUTxO ledger. Following in their footsteps, we also use state machines to specify smart contracts and then model their transactions as a state transition system in Section 3.

Consider the simple specification seen in Figure 1. There is a single state, which internally stores a map matching the public key hash of each user to their account value. There are multiple possible ways to model such a contract, but we have chosen to monolithically store accounts for our proof of concept as it is the simplest. With a UTxO-based system, it is potentially better for each individual account to be a separate UTxO, but we investigate this idea with the decentralized exchange example in Section A instead. There are five possible transitions, equivalent to entry points for the smart contract:

- **Open(pkh)**: Open an account associated with pkh
- **Deposit(pkh,v)**: Deposit value v into the account associated with pkh
- **Withdraw(pkh,v)**: Withdraw value v from the account associated with pkh
- **Transfer(pkh₁,pkh₂,v)**: Transfer value v from one account to another
- **Close(pkh)**: Close the account associated with pkh



■ **Figure 1** Account Simulation.

3 Modelling smart contracts as a State Transition System

As previously mentioned, Cardano validators are Boolean-valued functions. They take as arguments: a Datum; an Input (sometimes called a “redeemer”), a ScriptContext, and optionally some additional parameters. Since validator scripts are attached to UTxOs, the

smart contract itself is an output of a previous transaction, where the Datum is supplied. When we attempt to spend our smart contract in a different transaction, we supply it with an Input, and the blockchain infrastructure supplies the ScriptContext of the transaction. The additional parameters are supplied only when the smart contract is put on the blockchain for the first time, and do not change during runtime.

In the context of our account simulation example, the Datum is represented by the map of public key hashes and values and the Inputs correspond to our five possible transitions. The ScriptContext contains all the relevant metadata of the transaction, including signatories, outputs, and all other information necessary to ascertain if our constraints are being fulfilled. There are no additional parameters for the account simulation. An example of such parameters could be the number of signatures required for payment in a multi-signature wallet. The type signature for most validators has the following shape:

```
validator :: Params -> Datum -> Input -> ScriptContext -> Bool
```

Expressing and proving properties for such functions directly is cumbersome at best. Instead, we look for an alternative that is better suited to formal verification. Just like many other papers in the field [65, 32, 41, 31], it makes sense for us to model our contracts as state transition systems. This allows for a much easier way to translate specifications into Agda and prove our desired properties. We define a transition as:

$$P \vdash S \xrightarrow{I} S'$$

Here P refers to any parameters of the script that do not change throughout the contract lifespan. These are equivalent to the optional parameters of the validator script. A contract can have no parameters, as they are optional. In such cases, we can omit this field entirely. S is the combined knowledge necessary to deduce the current state of the smart contract. For the account simulation, this contains the map present in the Datum, as well as the value locked in the UTxO, which has to be extracted from the ScriptContext. I is the input being used and distinguishes which transition we are attempting to perform out of the five possible options. Finally, S' is the state of the contract if and when the transaction succeeds. It consists of the resulting map of accounts after any modifications occur, the value of the new UTxO representing our continuing script, as well as any signatories necessary to satisfy the constraints. All of this information needs to be extracted from the ScriptContext of the transaction as that is where all information about outputs is contained.

We will often need to consider multiple transitions applied consecutively for a list of inputs Is . Lists are either empty \emptyset , or built with $\text{cons } I :: Is$. We define a multi-step relation that corresponds to the reflexive transitive closure of our transition relation:

$$\frac{}{P \vdash S \xrightarrow{\emptyset} S} \qquad \frac{P \vdash S \xrightarrow{I} S' \quad P \vdash S' \xrightarrow{Is} S''}{P \vdash S \xrightarrow{I::Is} S''}$$

4 Smart contract properties

Blockchains are large and complex, with potentially millions of smart contracts running concurrently. Thus, it is nearly impossible to prove the properties of the system as a whole. Instead, we focus on individual contracts and what can be proven about them in isolation. Our use-case of Cardano is particularly amenable to this approach, as each contract only ever has access to information about the current transaction it is participating in.

4.1 Validity

Our abstract state is just a collection of data, but not all possible states are reachable during the runtime of our smart contract. What makes a state **Valid** then? This depends on the contract itself but primarily relates to what is stored in the Datum. In our account simulation example, states are valid if all values stored in the internal map are positive. Cardano allows values to be negative for easier numerical operations and to represent the burning of native tokens. We need to account for the possibility of negative values in our proofs because we are using the same types as the on-chain code. That being said, it should be impossible for a negative value to be stored in the Datum if the validator functions as intended. For other smart contracts, the predicate describing Valid states might be completely different.

State transitions from a valid state must lead to another valid state. Validity acts as an invariant on our state, which gives us exactly this property. It is necessary as both a sense check on our model and as a component to other proofs.

VALIDITY. $\forall S, P, I$ and S' , if S is **Valid** and $P \vdash S \xrightarrow{I} S'$, then S' is also **Valid**.

It is possible to simply have the validator fail and return false whenever you attempt to run it on non-valid inputs, but this is inadvisable. Consider the case in which a user is trying to deposit some currency into their account. If we needed to check the Validity of every transaction, we would have to scan all other accounts and check that their values remain positive. We believe that this is an unreasonable design choice, as it needlessly increases the complexity of the code and the model, as well as raising gas costs for the transaction. Instead, it is more sensible to keep things simple and prove the property as described.

With our definition, the property trivially holds if there are no Valid states. As mentioned in Section 1, there is a separate minting policy which guarantees that our initial state is valid and non-empty. In this paper we assume this and only consider what follows, with minting policies and their properties being the subject of future work. As long as we have initiality, we are guaranteed an execution trace that is valid throughout. It is also important that the Valid predicate correctly describes states that actually exist and are relevant to the smart contract but such definitions are largely contract dependent. Consequently, we only show what the predicate looks like for our example instead of describing it in detail.

If state invariants are a superset of the accessible states, then transition invariants [54] are a superset of the transitive closure of our transition relation, restricted to accessible states. Validity corresponds to a state invariant, but what about transition invariants? Certain properties would indeed take the form of transition invariants when we are concerned with the relationship between states before and after a transition (e.g. after a Withdraw v , the contract value decreases by v). However, this type of property is once again highly contract-specific and does not generalize well.² Such properties can easily be expressed and proven in our framework, but are not foundational to our approach.

4.2 Liquidity

Liquidity first appeared in Tsankov et. al (2018) [63] as a property that holds when the contract always admits a trace that decreases its balance. Other papers such as Bartoletti and Zunino (2019, [27]) and Bartoletti et. al. (2024, [25]) also refer to similar properties as Liquidity. Unfortunately, it is trivially easy to permanently lock away your funds in a UTxO

² It is also standard practice to include the previous history of transitions in the state (purely as the “ghost” state [53, *auxiliary variables*] that is erased in the runtime), in which case any transition invariant can be expressed as a state invariant.

that may never be spent, and comparatively much harder to ensure that your script behaves in such a way that no funds are ever lost. Recall that for the account simulation example, we mentioned one of the state components is the value locked by the UTxO. Given that the primary function of smart contracts is manipulating currency in some way, it is safe to say that most, if not all, contracts would have such a value component for their state. We refer to the value locked by a contract in a certain state as $\text{value}(S)$.

LIQUIDITY. $\forall P$ and S , if S is **Valid**, then $\exists I_S$ and S' such that $P \vdash S \xrightarrow{I_S}^* S'$ and $\text{value}(S') = 0$.

This is a liveness property [21] stating that there exists a series of inputs that allows us to get all the value out of our smart contract. The list of inputs can be empty when the value in S is already zero, in which case S' will be the same as S .

We mentioned previously that Validity will be a needed component for other properties, but why is this? The straightforward way to empty the smart contract of currency is to withdraw the amount in each account one by one. If one of the accounts had a negative value, we could not transition using withdraw, which would in turn complicate our proof. In this case, the proof does not become impossible, but it is much more laborious.

For a case in which Validity is necessary, consider instead the contrived example of a smart contract that only allows transactions if it contains a token in its value. The specification would require that the token is never paid out or burned until the contract is closed. This is more or less how thread tokens operate. Naturally, this contract is not liquid if the token vanishes, however, this action should not be possible within the state transition system due to Validity. Although Liquidity may not be provable for all states, it is only relevant to do so for valid ones.

Note that for this property we do not care *who* is able to extract the value, only that it is extractable. A property denoting authorized access, that limits which users can extract value and when, is desirable but will be the subject of future work instead.

4.3 Fidelity

When thinking about the DAO bug and its cause, developers focused on the exploit of re-entrancy [37, 56]. Re-entrancy itself is not possible on a UTxO blockchain, but it does have a root cause. The real issue was that the smart contract's internal state did not accurately represent its true balance (Herlihy, 2019, [44]) (Schrans et. al., 2018, [57]). We want to prove that the true value locked by our contract remains equal to the internal representation of value in the code throughout the runtime of the contract.

The internal value of a state varies depending on the contract itself. For our example of account simulations on UTxO, this would be the sum total of all currency in all accounts. We refer to the function that computes internal value from a state as $\text{internalV}(S)$. It is worth noting that sometimes there is no internal representation of value, in which case $\text{internalV}(S) = \text{value}(S)$, which makes Fidelity trivially true. As such, the contract is immune to this attack vector by design. Conversely, internal value might sometimes be difficult to specify, but it should always be definable if the contract aims to have any control over it.

Such a property would not only prevent re-entrancy on an account-based blockchain, but also any number of other potentially undiscovered exploits that might abuse this via currently unknown attack vectors. As we have not found any papers that analyze this property in-depth, we will be naming it Fidelity:

FIDELITY. $\forall P, S, I$, and S' , if $\text{value}(S) = \text{internalV}(S)$ and $P \vdash S \xrightarrow{I} S'$, then $\text{value}(S') = \text{internalV}(S')$.

Once again this takes the form of an invariant on our state, which means it could be collapsed into a single larger invariant alongside Validity, but we believe it is important enough to state on its own. It might be the case that Validity and Fidelity depend on each other for certain contracts, but this has not yet been observed. Having the two invariants be separate also has the benefit of more modular and legible proofs. Fidelity also has the added advantage of giving us freedom from double satisfaction on inputs, see Section 8 for details.

5 The validator and agda2hs

Because we are using Agda for our proofs, the validator implementation must also be written in Agda. This allows us to relate our model to the implementation. To compile blockchain code, we eventually need to export our Agda validator. The end target is Plutus [13], for which we use the main language of the platform, namely Plinth (formerly PlutusTx) [12]. Plinth is a high level purely functional language, which borrows most of the syntax and properties of Haskell. This allows it to leverage Haskell’s powerful type system as well as inherit many security features of its parent language.

Cockx et. al. (2022. [35]) describes agda2hs as a tool that translates an expressive subset of Agda to readable Haskell. Compared to other tools for program extraction, agda2hs uses a syntax that is already familiar to functional programmers and allows for both intrinsic and extrinsic approaches to verification. Conveniently, we can use agda2hs to export our code directly as Haskell, which can then be compiled to Plutus with minimal adjustments and some wrapper code.

To maximize compatibility with Plinth, the type signatures and function names used in Agda are carefully chosen to match those that are expected by the Haskell counterpart. That being said, we need to use a certain level of abstraction. The type of ScriptContext for instance is very complex, so we flatten it into the components relevant to our Validator. In the case of account simulations, we only care about the value of our input and output UTXOs, as well as any signatories of the transaction. The full ScriptContext contains many fields including the validity interval of our transaction, whether or not any native tokens are minted or burned, etc. As our contract does not interact with any of these components, we abstract them away to have a simpler type. We then use some helper functions and wrapper code in Haskell to translate the real ScriptContext into our abstracted one.

6 Equivalence between the model and validator

Recall that validators are just functions returning a Boolean. How do we establish a link between our model and the actual implementation code? To prove that the validator returning true is functionally equivalent to our model performing a state transition, we define a bijection relation \approx between the two.

```
record  $\approx$  {A : Set} (f : A → Bool) (R : A → Set) : Set where
  field to   : ∀ {a} → f a ≡ true → R a
  from      : ∀ {a} → R a      → f a ≡ true
```

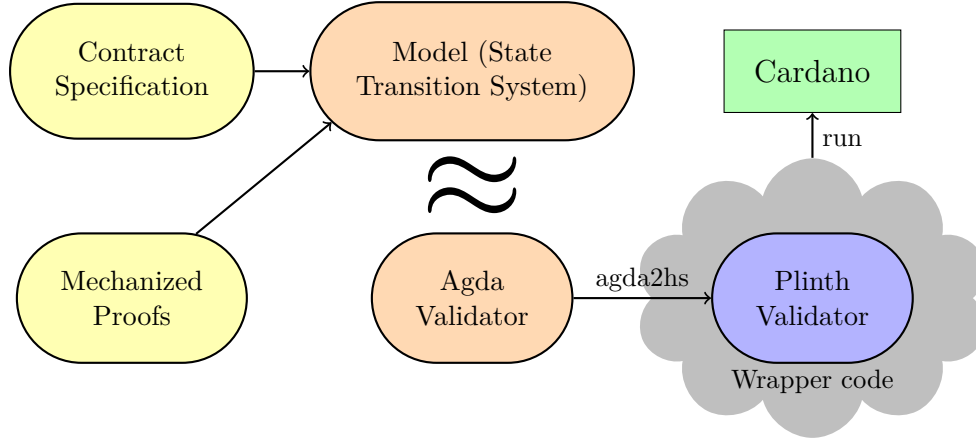
This connects a Boolean-valued function f , representing our validator, and a predicate R describing the set inhabited by our state transition system. In this relation, we say that if our validator returns true for an argument, then the same argument can be used to represent a transition from state to state, and vice-versa. This argument a is just a tuple containing the arguments of our validator function. Its type A describes that tuple, which for our validators and state transition system is $(\text{Params} \times \text{Datum} \times \text{Input} \times \text{ScriptContext})$. It is simple to see

how our validator can take the place of f and then be applied to a , but how do we turn a state transition system into predicate R ? Our states S and S' are derived from the Datum and ScriptContext, both of which are components of the tuple. Once the states are extracted from the argument, the parameters and Input slot directly into our transition relation to give us the desired predicate. This is discussed in more detail within the context of our account simulation example in the next section.

Most of the difficulty lies in proving this equivalence, as it carries the weight of all previous proofs. The properties of the model mean nothing unless they can be transferred to the actual implementation, which is why this relation is essential to our efforts.

7 The full pipeline

In Figure 2 we can see an overview of the process as a whole. As we have now covered all of the individual steps, we can go into more detail and show how we applied them to our account simulation smart contract.



■ **Figure 2** Formally verified Cardano smart contracts.

First, let us consider the state transition system. We translate the specification seen in Figure 1 to inference rules. The diagram itself only vaguely describes what the contract is expected to do, so here we can go into more detail on the actual constraints of our transitions. Examples of constraints include value comparisons, signature checks, and map operations on our Datum. These rules are then transcribed directly into Agda code, which can be seen in Appendix B. To keep things as simple as possible, we simplify the definition from Section 3 by removing the parameters because there are none for the account simulation.

Signatures, the map representing accounts, and the value locked by our UTxO, are all represented in the state, so we can extract them directly from it using $\text{signature}(S)$, $\text{map}(S)$, and the previously defined $\text{value}(S)$. For map operations (lookup , insert , delete), we use the same type signatures as the Haskell Map library. Maps are encoded as lists of pairs in our code because they offer a simple and effective solution, while Agda maps are much more complex than what we need and do not translate well into Haskell. It also bears mentioning that values on Cardano contain multiple currencies in various amounts. We write $\mathbf{0}$ for the value that represents zero amount of all currencies.

$$\frac{\begin{array}{l} \text{signature}(S') \equiv phk \quad \text{lookup } phk \text{ map}(S) \equiv \text{Nothing} \\ \text{map}(S') \equiv \text{insert } phk \, \mathbf{0} \text{ map}(S) \quad \text{value}(S') \equiv \text{value}(S) \end{array}}{S \xrightarrow{\text{OPEN}(phk)} S'}$$

$$\begin{array}{c}
\text{signature}(S') \equiv \text{phk} \quad \text{lookup } \text{phk} \quad \text{map}(S) \equiv \text{Just } \text{val} \quad v \geq 0 \\
\text{map}(S') \equiv \text{insert } \text{phk} \ (\text{val} + v) \ \text{map}(S) \quad \text{value}(S') \equiv \text{value}(S) + v \\
\hline
S \xrightarrow{\text{DEPOSIT}(\text{phk}, v)} S' \\
\\
\text{signature}(S') \equiv \text{phk} \quad \text{lookup } \text{phk} \quad \text{map}(S) \equiv \text{Just } \text{val} \quad v \geq 0 \quad \text{val} \geq v \\
\text{map}(S') \equiv \text{insert } \text{phk} \ (\text{val} - v) \ \text{map}(S) \quad \text{value}(S') \equiv \text{value}(S) - v \\
\hline
S \xrightarrow{\text{WITHDRAW}(\text{phk}, v)} S' \\
\\
\text{signature}(S') \equiv \text{phk}_1 \quad \text{lookup } \text{phk}_1 \quad \text{map}(S) \equiv \text{Just } \text{valFrom} \\
\text{lookup } \text{phk}_2 \quad \text{map}(S) \equiv \text{Just } \text{valTo} \quad v \geq 0 \quad \text{valFrom} \geq v \quad \text{phk}_1 \neq \text{phk}_2 \\
\text{map}(S') \equiv \text{insert } \text{phk}_1 \ (\text{valFrom} - v) \ (\text{insert } \text{phk}_2 \ (\text{valTo} + v) \ \text{map}(S)) \\
\text{value}(S') \equiv \text{value}(S) \\
\hline
S \xrightarrow{\text{TRANSFER}(\text{phk}_1, \text{phk}_2, v)} S' \\
\\
\text{signature}(S') \equiv \text{phk} \quad \text{lookup } \text{phk} \quad \text{map}(S) \equiv 0 \\
\text{map}(S') \equiv \text{delete } \text{phk} \ 0 \ \text{map}(S) \quad \text{value}(S') \equiv \text{value}(S) \\
\hline
S \xrightarrow{\text{CLOSE}(\text{phk})} S'
\end{array}$$

All transactions must be signed by the owner of the account being modified. An empty account can be opened, if it does not already exist for that user's public key hash. This leaves the value locked by the UTxO unchanged. Once an account exists and its value can be looked up, the owner can deposit a non-negative value, or withdraw up to the total amount they currently have. The map and value need to change accordingly. A user can transfer from their account to another existing account different from their own up to the total value they own, with the map changing but not the value since the change is only internal. Finally, an empty account can be closed, which again only affects the internal map.

Note that these rules do not mention anything about the user state. For example, a value withdrawn from an account does not necessarily need to go to the user who withdrew it. This is an intentional design choice. In this case, the owner has full control over the transaction being put on chain, as they need to sign it in order for it to ever be approved. They can then send the funds to themselves, or someone else, or perhaps even another smart contract in the same transaction, without needing to submit two separate transactions. This approach is not always correct. In the case of a the Multi-Signature Wallet, it is imperative that the funds get sent to a specific address once payment is approved.

We can then prove our three properties in Agda, with their type signatures being almost identical to the definitions in Section 4. Validity is relatively simple, with about one hundred total lines of code including all ancillary lemmas which mostly consist of simple mathematical manipulation of integers. A state is valid if all elements of the Datum (map) are positive.

Valid : State → Set

Valid s = All (λ (_ , v) → geq v 0 ≡ true) (s . datum)

We then prove the invariant for a single transition, which extends to the reflexive transitive closure via simple induction.

validity : ∀ (s s' : State) (i : Input) → **Valid** s → s ~ [i] ~> s' → **Valid** s'

The Agda proof of Liquidity for simulating accounts takes the form of withdrawing all value from each account one by one, which naturally results in a contract with no value. Similar to Validity, the code including ancillary lemmas also occupies about one hundred lines. Note that below we also use the Fidelity statement as a pre-requisite, because it results in a simpler proof. This lends more credence to our previously mentioned point of collapsing Validity and Fidelity into a single general state invariant, which can then be used for other potential liveness properties.

```
liquidity : ∀ (s : State) → s .context .value ≡ sumVal (s .datum)
          → Valid s
          → ∃[ s' ] ∃[ is ] (s ~[ is ]~* s') × (s' .context .value ≡ 0)
```

Proving Fidelity once again totals around one hundred lines of code and lemmas. These lemmas consist of simple mathematical manipulation and amount to proving the properties of *insert*, *delete*, and *lookup*, which are not part of the Agda standard library.

```
fidelity : ∀ (s s' : State) (i : Input) → s .context .value ≡ sumVal (s .datum)
          → s ~[ i ]~> s'
          → s' .context .value ≡ sumVal (s' .datum)
```

Our validator code is simple and straightforward, mirroring the specification. Shared code, such as checking signatures and the UTxO value, is exposed directly. The more specific internal checks pertaining to the map and values stored in it are collapsed into helper functions to increase legibility. The code can be found in Appendix C.1.

With the implementation in hand, we can prove it equivalent to our model. For our desired relation we need to transform the validator into a function, and our state transition system into a predicate, both of which need to apply to the same argument. Recall from Section 3 that the initial state of our model is extracted from the Datum and ScriptContext, and the final state only from the context. We use some helper functions for that:

```
getS : Datum → ScriptContext → State
getS' : ScriptContext → State
```

In order to even state the bijection relation of Section 6, we need to also convert our validator and the model relation to their unary equivalents:

$\text{Argument} = \text{Datum} \times \text{Input} \times \text{ScriptContext}$	<pre>toF : Argument → Bool toF (d , i , ctx) = agdaValidator d i ctx toR : Argument → Set toR (d , i , ctx) = getS d ctx ~[i]~> getS' ctx</pre>
--	---

It is now possible to state the desired equivalence:

```
functionalEquiv : toF ≈ toR
```

Finally, we prove the two halves of this relation separately. The proofs themselves take about 150 lines of code each, but mostly involve simple operations. A large portion of the work necessary is reconciling the *agda2hs* library with the Agda standard library. This involves lemmas for easy properties such as integer equality implying that the integers are also equivalent. Thankfully, such proofs can be reused across multiple contracts and in the future could be compiled into a small library. Being able to use *agda2hs* is a great boon, as it allows us to directly export our code, but comes with some disadvantages. Agda is dependently typed and much more expressive than Haskell. Consequently, only a certain subset can be properly translated. The *agda2hs* library is enough for our implementation,

but the full power of Agda is necessary for the proofs. Thankfully, if we prove that the model and validator are functionally equivalent, we only need to export the implementation. The two sides of the relation have the following type signature:

<p>transitionImpliesValidator :</p> $\forall (l : \text{Datum}) (i : \text{Input}) (ctx : \text{ScriptContext})$ $\rightarrow \text{getS } l \text{ ctx } \sim [i] \sim \text{getS'} \text{ ctx}$ $\rightarrow \text{agdaValidator } l \text{ i ctx } \equiv \text{true}$	<p>validatorImpliesTransition :</p> $\forall (l : \text{Datum}) (i : \text{Input}) (ctx : \text{ScriptContext})$ $\rightarrow \text{agdaValidator } l \text{ i ctx } \equiv \text{true}$ $\rightarrow \text{getS } l \text{ ctx } \sim [i] \sim \text{getS'} \text{ ctx}$
--	--

The Haskell version of the code looks almost identical to the Agda version and can be found alongside it in Appendix C.2. Once the validator is exported to Plinth, we use some additional helper functions and wrapper code to compile our implementation directly to blockchain code. Using the Cardano Node Emulator [3], we also ran a simple battery of tests. QuickCheck (Claessen and Hughes, 2000, [34]), specifically a version of quickcheck-dynamic [14] designed to integrate with Plinth, was used to write a generator for randomized tests. These tests serve as an extra layer of protection to make sure that our contract correctly implemented its specification and that our properties were effective in preventing bugs. The validator passed all tests successfully.

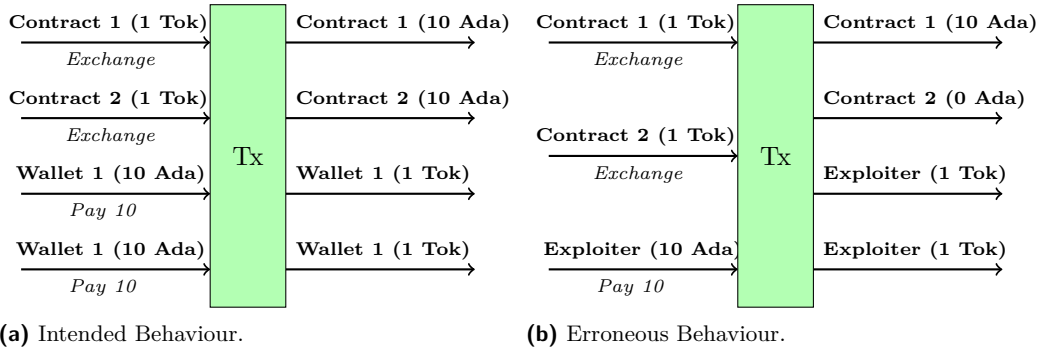
We have also applied this full process to contracts for a multi-signature wallet and order book decentralized exchange, which are described in Appendix A. We omit the proof details for brevity, but they were similar in complexity to the account simulation ones and are fully accessible in the GitHub repository.

8 Double Satisfaction on UTxO

Vinogradova and Melkonian (2025, [64]) define double satisfaction as a vulnerability that occurs for any constraint when two separate contracts impose the same constraint in a transaction (Tx). This is a serious issue which has also been covered in several audits [4, 17] of Cardano smart contracts. Building on their work, we focus on the two main cases where this phenomenon causes problems, namely when the constraints are fulfilled by the same input or output of a transaction.

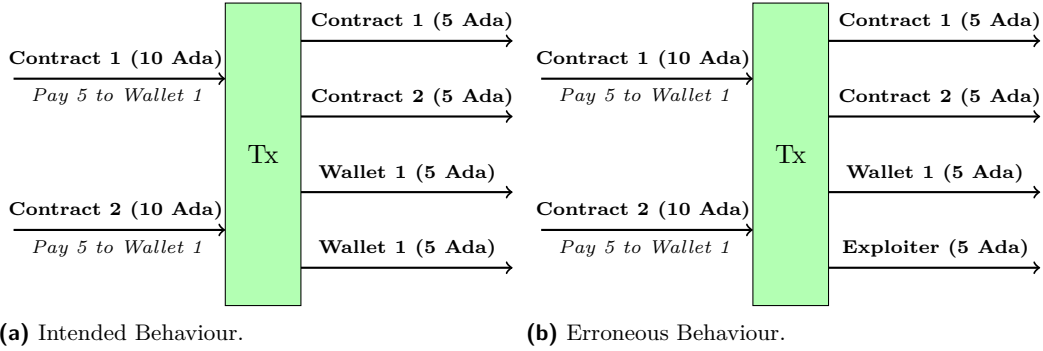
For inputs, let us consider the example in Figure 3 where two different contracts are attempting to exchange 1 “Token” for 10 Ada (the main currency of Cardano). In Subfigure (a), we can see the desired result of both contracts being paid separately by Wallet 1, which then receives two different outputs containing the token, one from each contract. Unfortunately, issues can occur if the two contracts attempt to claim ownership over the same input. In Subfigure (b), the scripts see the input of an exploiter, expect to be paid, and approve the transaction even though not enough money is being paid in for both of them. The exploiter then gains the two tokens even though he has only paid for one. This form of double satisfaction has many possible solutions, but having a property that guarantees you will not run into it at all is still desirable, especially to novice developers.

Fidelity gives us exactly this property. If our model expects to receive some funds as part of a transition, this will be reflected in the internal representation. The account simulator only consumes outside inputs for the Deposit action. According to Fidelity, all transitions including Deposit explicitly guarantee that $\text{internalV}(S) = \text{value}(S)$, thus ensuring double satisfaction cannot happen. Consequently, our validator only approves transactions when given the necessary amount of currency, but does this without needing to inspect individual inputs. It does not strictly matter where the value comes from as long as the validator receives its required share.



■ **Figure 3** Double satisfaction for inputs.

In Figure 4 we look at double satisfaction for outputs using two different contracts that are required to make a payment of 5 Ada to the same wallet address. We can see the intended behaviour on the left, with Wallet 1 receiving proper payment from the two contracts. Again, problems may arise when the same output fulfils the constraints of two separate scripts. On the right, a malicious exploiter places both of these contracts in the same transaction with only one payment output to Wallet 1 while redirecting the rest to themselves. This is possible because both scripts scan the transaction outputs and find the required payment but do not know that another script in the same transaction is expected to perform the same action.



■ **Figure 4** Double satisfaction for outputs.

This is a much more nuanced issue, as it is unreasonable for a validator to check what all other scripts in the same transaction are doing. Marlowe (Seijas et. al, 2020, [48]) is a domain specific language for Cardano that attempts to solve this issue by limiting what scripts may run alongside each other. While this technically solves the issue, it is also very limiting and can potentially make submitting transactions cumbersome. Message-passing (Hoare, 1987, [46])(Vinogradova and Melkonian, 2025, [64]) can be used to deal with double satisfaction for both inputs and outputs, but this will have to be the subject of future work.

9 Related work

Anton Setzer’s work **modelling Bitcoin in Agda** [60] offers a different approach, by formalizing transactions on the ledger not as state transitions, but by using transaction trees. Many other blockchains have also been spearheading the use of fully mechanized formalizations using proof assistants. **Scilla** [59] is the high-level smart contract language

developed for the Zilliqa [58] blockchain. It is particularly significant because it also uses a state transition system to define contracts. Scilla is formalized in Coq, and attempts to address security and correctness issues via formal verification. A similar approach using Coq is also applied by both the Tezos [29] and Concordium [22] blockchains for smart contract certification and verification.

The **K Framework** [55] is a robust tool for language semantics that offers a structured approach to specifying the semantics of programming languages. It has been used to specify the formal semantics of the Ethereum virtual machine [45], and is of great interest for specifying smart contracts in general, as well as their properties. While our proposed properties generalize well to different contracts, the verification pipeline specifically targets the Cardano ecosystem. The K Framework could potentially be used to bridge this gap.

Tools such as **Certora** [28, 5] for Solidity [16] and the **Move Prover** [66, 38] for Move [30] and the Diem Blockchain [6] attempt to automate the process by leveraging SMT-based proof techniques. This generally involves compiling contracts and their associated properties into a logical formula, which can then be sent to an SMT solver. When compared to our proposed method, this approach has the potential to significantly reduce the manual work required, but offers fewer guarantees than full formal verification and poses the risk of significantly limiting the scope of provable properties.

Smart Code Verifier [15] is a formal verification tool currently under development for the Cardano blockchain that attempts to bridge the gap between the two approaches. Smart contract code (written in Aiken [1], Plinth, etc.) is annotated with specifications in a dedicated language. The annotations and source code are translated into Lean4 [51] proof obligations. These obligations are used to generate an SMTLib [24] formula which is then automatically discharged using Z3 [36]. Any proofs that the solver cannot complete automatically can instead be solved manually, and counterexamples are generated when specifications are violated.

10 Future Work

Cardano has support for native custom tokens [32], which means there is also interest in **token properties**. This is particularly important because as mentioned previously, minting policies and thread tokens are integral to assuring our initial state. Such guarantees are needed for invariant properties like Validity and Fidelity which state that a contract maintains the property once it has it, but not that it starts with said property. A special token minted in the transaction where a smart contract is first put onto the blockchain can guarantee our desired constraints on the initial state. Furthermore, non-fungible tokens are commonly used in various blockchain applications, for which proving NFT uniqueness would be desirable. Minting policies on Cardano decide when a certain token can be minted or burned, and are very similar to validator scripts because they are also functions that return a Boolean. It stands to reason that we can use the same methods to specify, model, and prove their properties, which will be the next step in our research.

For the validator implementation on the side of Agda, we use several layers of abstraction instead of using the exact same types as the blockchain to simplify proof efforts. This decision was made to keep the proof of concept for this process as simple as possible. That being said, a **formalized Cardano Ledger** [47] already exists in Agda. Optimally, we would fully integrate our method with the ledger model to be as close as possible to the target language of Plinth. It is currently unclear how compatible the existing formalisation is with our current approach, so more investigation will be necessary.

Double satisfaction remains a major issue for UTxO-based smart contracts. We have found a way to resolve it for inputs with Fidelity, but work on the output side is still ongoing. Having a property which guarantees that your code is safe from double output satisfaction would be extremely desirable, so we aim to investigate what that property would look like and how we can prove it using our method.

Finally, we also intend to eventually also showcase more contract-specific properties such as the ones mentioned about transition invariants, authorized access, and making sure that funds leaving the contract are given to the correct recipient.

11 Conclusion

We have outlined three major properties we believe could serve as the baseline for future formal verification of various smart contracts regardless of blockchain. Narrowing our use-case down to the Cardano blockchain, we also proposed a method for modelling specifications as state transition systems, which is well suited for mechanized proofs using Agda. Using the Agda proof assistant has the added benefit of allowing us to easily export our implementation as Haskell code, which can then be compiled directly to the blockchain.

The proof of concept pipeline can be observed for the account simulation example, including all the intermediate steps: specification, modelling, proofs, implementation, equivalence of model and implementation, exporting using `agda2hs`, and finally testing on an emulated ledger node. We have successfully applied this formula to three Cardano scripts, but more work is required to determine if our process generalizes well. We believe that it will be applicable to a variety of smart contracts and a broad number of different properties.

We hope our work can provide a common baseline for the process of modelling and verifying smart contracts in a landscape where there is no common baseline for this process. Frameworks such as ours will become increasingly attractive as the trust requirements of user-generated code continues rising.

References

- 1 Aiken. <https://aiken-lang.org/>. Accessed: 2025-03-25.
- 2 Aku's nightmare: \$34m locked forever as flaw highlights danger of smart contracts. <https://www.pymnts.com/blockchain/2022/akus-nightmare-34m-locked-forever-as-flaw-highlights-danger-of-smart-contracts/>. Accessed: 2025-02-02.
- 3 Cardano node emulator. <https://github.com/IntersectMBO/cardano-node-emulator>. Accessed: 2025-02-02.
- 4 Cardano vulnerabilities #1 – double satisfaction. https://medium.com/@vacuumlabs_auditing/cardano-vulnerabilities-1-double-satisfaction-219f1bc9665e. Accessed: 2025-02-02.
- 5 Certora white paper. <https://www.certora.com/blog/white-paper>. Accessed: 2025-03-25.
- 6 The diem blockchain. <https://developers.diem.com/docs/technical-papers/the-diem-blockchain-paper/>. Accessed: 2025-03-25.
- 7 Largest blockchains in crypto ranked by TVL. <https://coinmarketcap.com/chain-ranking/>. Accessed: 2025-02-02.
- 8 Lessons from the wormhole exploit: Smart contract vulnerabilities introduce risk. <https://www.chainalysis.com/blog/wormhole-hack-february-2022/>. Accessed: 2025-02-02.
- 9 NFT project Aku dreams loses \$34 million to smart contract flaw. <https://bitcoinist.com/nft-project-aku-dreams-loses-34-million-to-smart-contract-flaw/>. Accessed: 2025-02-02.

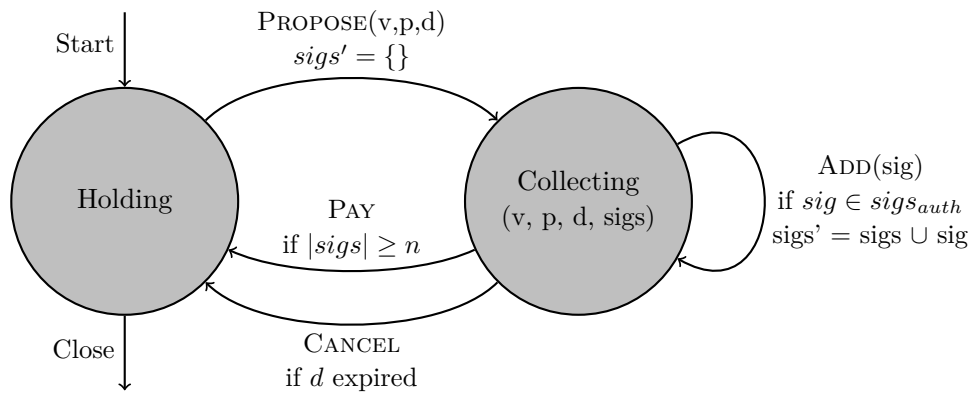
- 10 Over \$1 million permanently locked in defi smart contract. <https://coingeek.com/over-1-million-permanently-locked-in-defi-smart-contract/>. Accessed: 2025-02-02.
- 11 Over \$1m in user funds on compound fork perfect finance are frozen after code change error. <https://www.theblock.co/linked/83792/user-funds-perfect-finance-frozen-code-error>. Accessed: 2025-02-02.
- 12 Plinth and Plutus documentation. <https://plutus.cardano.intersectmbo.org/docs/>. Accessed: 2025-02-02.
- 13 Plutus Github. <https://github.com/IntersectMBO/plutus>. Accessed: 2025-02-02.
- 14 QuickCheck-dynamic. <https://github.com/input-output-hk/quickcheck-dynamic>. Accessed: 2025-02-02.
- 15 Smart code verifier. <https://github.com/input-output-hk/smartcode-verifier>. Accessed: 2025-03-25.
- 16 Solidity documentation. <https://app.readthedocs.org/projects/solidity/downloads/pdf/develop/>. Accessed: 2025-03-25.
- 17 Tweag technical review of marlowe. <https://github.com/tweag/tweag-audit-reports/blob/main/Marlowe-2023-03.pdf>. Accessed: 2025-02-02.
- 18 What is a DEX? <https://www.coinbase.com/en-gb/learn/crypto-basics/what-is-a-dex>. Accessed: 2025-02-02.
- 19 What is a DEX (decentralized exchange)? <https://chain.link/education-hub/what-is-decentralized-exchange-dex>. Accessed: 2025-02-02.
- 20 Wormhole cryptocurrency platform hacked for \$325 million after error on GitHub. <https://www.theverge.com/2022/2/3/22916111/wormhole-hack-github-error-325-million-theft-ethereum-solana>. Accessed: 2025-02-02.
- 21 Bowen Alpern and Fred B Schneider. Defining liveness. *Information processing letters*, 21(4):181–185, 1985. doi:10.1016/0020-0190(85)90056-0.
- 22 Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. Concert: a smart contract certification framework in coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 215–228, 2020. doi:10.1145/3372885.3373829.
- 23 Morena Barboni, Andrea Morichetta, and Andrea Polini. Smart contract testing: challenges and opportunities. In *Proceedings of the 5th International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 21–24, 2022. doi:10.1145/3528226.3528370.
- 24 Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, page 14, 2010.
- 25 Massimo Bartoletti, Angelo Ferrando, Enrico Lipparini, and Vadim Malvone. Solvent: liquidity verification of smart contracts. In *International Conference on Integrated Formal Methods*, pages 256–266. Springer, 2024. doi:10.1007/978-3-031-76554-4_14.
- 26 Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. A theory of transaction parallelism in blockchains. *Logical Methods in Computer Science*, 17, 2021. doi:10.46298/LMCS-17(4:10)2021.
- 27 Massimo Bartoletti and Roberto Zunino. Verifying liquidity of bitcoin contracts. In *International Conference on Principles of Security and Trust*, pages 222–247. Springer, 2019. doi:10.1007/978-3-030-17138-4_10.
- 28 Thomas Bernardi, Nurit Dor, Anastasia Fedotov, Shelly Grossman, Neil Immerman, Daniel Jackson, Alexander Nutz, Lior Oppenheim, Or Pistiner, Noam Rinetzky, et al. Wip: Finding bugs automatically in smart contracts with parameterized invariants. Retrieved July, 14:2020, 2020.
- 29 Bruno Bernardo, Raphaël Cauderlier, Guillaume Claret, Arvid Jakobsson, Basile Pesin, and Julien Tesson. Making tezos smart contracts more reliable with coq. In *International Symposium on Leveraging Applications of Formal Methods*, pages 60–72. Springer, 2020. doi:10.1007/978-3-030-61467-6_5.

- 30 Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario Rossi Rain, Stephane Sezer, et al. Move: A language with programmable resources. *Libra Assoc*, 1, 2019.
- 31 Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3(37):2–1, 2014.
- 32 Manuel MT Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann Müller, Michael Peyton Jones, Polina Vinogradova, and Philip Wadler. Native custom tokens in the extended UTXO model. In *International Symposium on Leveraging Applications of Formal Methods*, pages 89–111. Springer, 2020.
- 33 Manuel MT Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The extended UTXO model. In *Financial Cryptography and Data Security: FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers 24*, pages 525–539. Springer, 2020. doi:10.1007/978-3-030-54455-3_37.
- 34 Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, 2000. doi:10.1145/351240.351266.
- 35 Jesper Cockx, Orestis Melkonian, Lucas Escot, James Chapman, and Ulf Norell. Reasonable Agda is correct Haskell: writing verified haskell using agda2hs. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*, pages 108–122, 2022. doi:10.1145/3546189.3549920.
- 36 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
- 37 Vikram Dhillon, David Metcalf, Max Hooper, Vikram Dhillon, David Metcalf, and Max Hooper. The DAO hacked. *blockchain enabled applications: Understand the blockchain Ecosystem and How to Make it work for you*, pages 67–78, 2017.
- 38 David Dill, Wolfgang Grieskamp, Junkil Park, Shaz Qadeer, Meng Xu, and Emma Zhong. Fast and reliable formal verification of smart contracts with the move prover. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 183–200. Springer, 2022.
- 39 Tudor Ferariu and Orestis Melkonian. agda2plinth: Formal verification of Cardano smart contracts in Agda. <https://github.com/tferariu/agda2plinth>, March 2025. doi:10.5281/zenodo.15118668.
- 40 Tudor Ferariu, Philip Wadler, and Orestis Melkonian. tferariu/agda2plinth. Software, sw-hId: swh:1:dir:03dd5b9f802117f3b5cccf0ed56d4214fee41238 (visited on 2025-05-06). URL: <https://github.com/tferariu/agda2plinth>, doi:10.4230/artifacts.23006.
- 41 LM Goodman. Tezos—a self-amending crypto-ledger white paper. URL: https://www.tezos.com/static/papers/white_paper.pdf, 4:1432–1465, 2014.
- 42 Tobias Guggenberger, Vincent Schlatt, Jonathan Schmid, and Nils Urbach. A structured overview of attacks on blockchain systems. *PACIS*, page 100, 2021. URL: <https://aisel.aisnet.org/pacis2021/100>.
- 43 Daojing He, Zhi Deng, Yuxing Zhang, Sammy Chan, Yao Cheng, and Nadra Guizani. Smart contract vulnerability analysis and security audit. *IEEE Network*, 34(5):276–282, 2020. doi:10.1109/MNET.001.1900656.
- 44 Maurice Herlihy. Blockchains from a distributed computing perspective. *Communications of the ACM*, 62(2):78–85, 2019. doi:10.1145/3209623.
- 45 Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. KEVM: A complete formal semantics of the Ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217. IEEE, 2018.

- 46 Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978. doi:10.1145/359576.359585.
- 47 Andre Knispel, Orestis Melkonian, James Chapman, Alasdair Hill, Joosep Jääger, William DeMeo, and Ulf Norell. Formal specification of the Cardano blockchain ledger, mechanized in Agda. In *5th International Workshop on Formal Methods for Blockchains (FMBC 2024)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024.
- 48 Pablo Lamela Seijas, Alexander Nemish, David Smith, and Simon Thompson. Marlowe: implementing and analysing financial contracts on blockchain. In *Financial Cryptography and Data Security: FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers 24*, pages 496–511. Springer, 2020. doi:10.1007/978-3-030-54455-3_35.
- 49 George H Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955.
- 50 Mikkel Milo, Eske Hoy Nielsen, Danil Annenkov, and Bas Spitters. Finding smart contract vulnerabilities with concert’s property-based testing framework. *arXiv preprint arXiv:2208.00758*, 2022. doi:10.48550/arXiv.2208.00758.
- 51 Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*, pages 625–635. Springer, 2021. doi:10.1007/978-3-030-79876-5_37.
- 52 Ulf Norell. Dependently typed programming in Agda. In *International School on Advanced Functional Programming*, pages 230–266. Springer, 2008. doi:10.1007/978-3-642-04652-0_5.
- 53 Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976. doi:10.1145/360051.360224.
- 54 Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.*, pages 32–41. IEEE, 2004. doi:10.1109/LICS.2004.1319598.
- 55 Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. doi:10.1016/J.JLAP.2010.03.012.
- 56 Francisco Santos and Vasileios Kostakis. The DAO: a million dollar lesson in blockchain governance. *School of Business and Governance, Ragnar Nurkse Department of Innovation and Governance*, 2018.
- 57 Franklin Schrans, Susan Eisenbach, and Sophia Drossopoulou. Writing safe smart contracts in Flint. In *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming*, pages 218–219, 2018. doi:10.1145/3191697.3213790.
- 58 A Secure. The zilliqa project: A secure, scalable blockchain platform, 2018.
- 59 Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. Safer smart contract programming with Scilla. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019. doi:10.1145/3360611.
- 60 Anton Setzer. Modelling bitcoin in agda. *arXiv preprint arXiv:1804.06398*, 2018. arXiv:1804.06398.
- 61 Matthieu Sozeau, Abhishek Anand, Simon Boulrier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq project. *J. Autom. Reason.*, 64(5):947–999, 2020. doi:10.1007/S10817-019-09540-0.
- 62 The Coq Development Team. The Coq proof assistant, September 2024. doi:10.5281/zenodo.14542673.
- 63 Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 67–82, 2018.
- 64 Polina Vinogradova and Orestis Melkonian. Message-passing in the extended UTxO ledger. In Jurlind Budurushi, Oksana Kulyk, Sarah Allen, Theo Diamandis, Arian Klages-Mundt,

- Andrea Bracciali, Geoffrey Goodell, and Shin'ichiro Matsuo, editors, *Financial Cryptography and Data Security. FC 2024 International Workshops*, pages 150–169, Cham, 2025. Springer Nature Switzerland.
- 65 Polina Vinogradova, Orestis Melkonian, Philip Wadler, Manuel Chakravarty, Jacco Krijnen, Michael Peyton Jones, James Chapman, and Tudor Ferariu. Structured contracts in the EUTxO ledger model. In *5th International Workshop on Formal Methods for Blockchains (FMBC 2024)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024.
- 66 Jingyi Emma Zhong, Kevin Cheang, Shaz Qadeer, Wolfgang Grieskamp, Sam Blackshear, Junkil Park, Yoni Zohar, Clark Barrett, and David L Dill. The move prover. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I 32*, pages 137–150. Springer, 2020. doi: 10.1007/978-3-030-53288-8_7.

A Additional example contracts

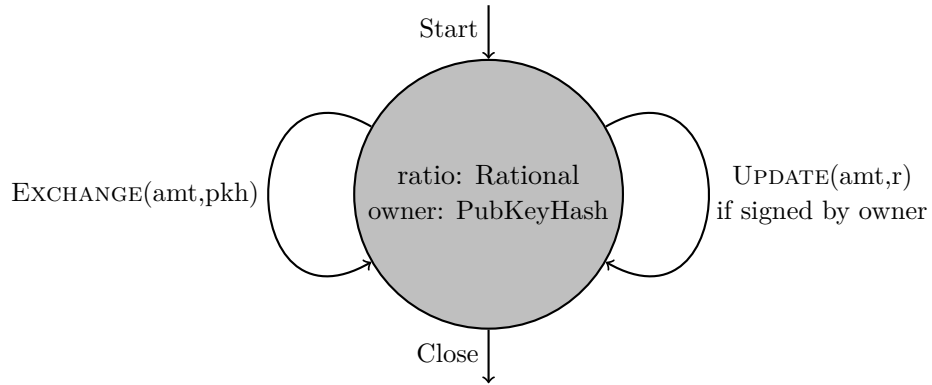


■ **Figure 5** Multi-Signature Wallet.

Chakravarty et. al. (2020, [33, 32]), specifies a Multi-Signature Wallet that is also a worthwhile target for investigation. In Figure 5, we can see a simple specification for this contract, with the idea being that it cycles between two possible states, one where we are holding for a payment to be proposed, and one where signatures are collected for the requested payment. Users can propose a payment of a certain value, to a certain address, and set a deadline by which the signatures must be gathered for this payment. After a payment is proposed, users can add their signatures one by one. Once enough are gathered, the payment can be put into effect. Alternatively, once the deadline has passed, the collecting state can be canceled alongside the proposed payment.

Last but not least, we consider a slightly more complex smart contract in the form of an Order Book Decentralized Exchange. Contrary to its name, the traditional description of such peer-to-peer marketplaces is quite centralized [19, 18], with all of the different orders being stored in the same place. Instead, we try to take advantage of the nature of UTxO and have many small specialized contracts, where each instance is a single user exchanging one currency for another. The script is parametrized by the currency being traded, so the blockchain can be easily scanned for scripts exchanging exactly the desired currencies.

The specification of one such instance can be seen in Figure 6. When interacting with the contract, the owner can update the amount or rate of the exchange, and all other users can consume all or part of the limit order to exchange between the two specified currencies.



■ **Figure 6** Decentralized Exchange.

Finally, the owner can always close the script when they want to recover their funds. Market orders can be handled entirely off-chain, by crafting a transaction with multiple instances of this smart contract from different owners.

B State transition system for account simulation

- Single-step transition

data $_ \sim _ \rightarrow _ : \text{State} \rightarrow \text{Input} \rightarrow \text{State} \rightarrow \text{Set where}$

```

TOpen :  $\forall \{pkh\ s\ s'\}$ 
   $\rightarrow pkh \equiv s.\text{context}.\text{tsig}$ 
   $\rightarrow \text{lookup } pkh\ (s.\text{datum}) \equiv \text{Nothing}$ 
   $\rightarrow s'.\text{datum} \equiv \text{insert } pkh\ 0\ (s.\text{datum})$ 
   $\rightarrow s.\text{context}.\text{value} \equiv s.\text{context}.\text{value}$ 
  -----
   $\rightarrow s \sim [(\text{Open } pkh)] \rightarrow s'$ 

TClose :  $\forall \{pkh\ s\ s'\}$ 
   $\rightarrow pkh \equiv s'.\text{context}.\text{tsig}$ 
   $\rightarrow \text{lookup } pkh\ (s.\text{datum}) \equiv \text{Just } 0$ 
   $\rightarrow s'.\text{datum} \equiv \text{delete } pkh\ (s.\text{datum})$ 
   $\rightarrow s'.\text{context}.\text{value} \equiv s.\text{context}.\text{value}$ 
  -----
   $\rightarrow s \sim [(\text{Close } pkh)] \rightarrow s'$ 

TWithdraw :  $\forall \{pkh\ val\ s\ s'\ v\}$ 
   $\rightarrow pkh \equiv s'.\text{context}.\text{tsig}$ 
   $\rightarrow \text{lookup } pkh\ (s.\text{datum}) \equiv \text{Just } v$ 
   $\rightarrow val \geq \text{emptyValue}$ 
   $\rightarrow v \geq val$ 
   $\rightarrow s'.\text{datum} \equiv \text{insert } pkh\ (v - val)\ (s.\text{datum})$ 
   $\rightarrow s'.\text{context}.\text{value} \equiv s.\text{context}.\text{value} - val$ 
  -----
   $\rightarrow s \sim [(\text{Withdraw } pkh\ val)] \rightarrow s'$ 

```

```

TDeposit :  $\forall \{pkh\} \{val\} \{s\} \{s'\} \{v\}$ 
  →  $pkh \equiv s'.context.tsig$ 
  →  $lookup\ pkh\ (s.datum) \equiv Just\ v$ 
  →  $val \geq emptyValue$ 
  →  $s'.datum \equiv insert\ pkh\ (v + val)\ (s.datum)$ 
  →  $s'.context.value \equiv s.context.value + val$ 
  -----
  →  $s \sim [Deposit\ pkh\ val] \sim> s'$ 

TTransfer :  $\forall \{from\ to\ val\} \{s\} \{s'\} \{vF\} \{vT\}$ 
  →  $from \equiv s'.context.tsig$ 
  →  $lookup\ from\ (s.datum) \equiv Just\ vF$ 
  →  $lookup\ to\ (s.datum) \equiv Just\ vT$ 
  →  $vF \geq val$ 
  →  $val \geq emptyValue$ 
  →  $from \neq to$ 
  →  $s'.datum \equiv insert\ from\ (vF - val)\ (insert\ to\ (vT + val)\ (s.datum))$ 
  →  $s'.context.value \equiv s.context.value$ 
  -----
  →  $s \sim [Transfer\ from\ to\ val] \sim> s'$ 

- Multi-step transition (a.k.a reflexive-transitive closure)
data  $\sim[_] \sim^*$  : State → List Input → State → Set where

root :  $\forall \{s\}$ 
  -----
  →  $s \sim [ ] \sim^* s$ 

cons :  $\forall \{s\} \{s'\} \{s''\} \{i\} \{is\}$ 
  →  $s \sim [i] \sim> s'$ 
  →  $s' \sim [is] \sim^* s''$ 
  -----
  →  $s \sim [(i :: is)] \sim^* s''$ 

```

C Validator code for account simulation

C.1 Original Agda validator

```

agdaValidator : Datum → Input → ScriptContext → Bool
agdaValidator dat inp ctx = case inp of λ where
  (Open pkh)      → checkSigned pkh ctx && not (checkMembership (lookup pkh dat)) &&
    newDatum ctx == insert pkh 0 dat && newValue ctx == oldValue ctx
  (Close pkh)     → checkSigned pkh ctx && checkEmpty (lookup pkh dat) &&
    newDatum ctx == delete pkh dat && newValue ctx == oldValue ctx
  (Withdraw pkh val) → checkSigned pkh ctx && checkWithdraw (lookup pkh dat) pkh val dat ctx &&
    newValue ctx == oldValue ctx - val
  (Deposit pkh val) → checkSigned pkh ctx && checkDeposit (lookup pkh dat) pkh val dat ctx &&

```

```

                                newValue ctx == oldValue ctx + val
(Transfer from to val) → checkSigned from ctx &&
                        checkTransfer (lookup from dat) (lookup to dat) from to val dat ctx &&
                                newValue ctx == oldValue ctx
{-# COMPILE AGDA2HS agdaValidator #-}

```

C.2 Extracted Haskell validator

```

agdaValidator :: Datum -> Input -> ScriptContext -> Bool
agdaValidator dat inp ctx = case inp of
  Open pkh      -> checkSigned pkh ctx && not (checkMembership (lookup pkh dat)) &&
                    newDatum ctx == insert pkh 0 dat && newValue ctx == oldValue ctx
  Close pkh     -> checkSigned pkh ctx && checkEmpty (lookup pkh dat) &&
                    newDatum ctx == delete pkh dat && newValue ctx == oldValue ctx
  Withdraw pkh val -> checkSigned pkh ctx &&
                    checkWithdraw (lookup pkh dat) pkh val dat ctx &&
                    newValue ctx == oldValue ctx - val
  Deposit pkh val -> checkSigned pkh ctx &&
                    checkDeposit (lookup pkh dat) pkh val dat ctx &&
                    newValue ctx == oldValue ctx + val
  Transfer from to val -> checkSigned from ctx &&
                        checkTransfer (lookup from dat) (lookup to dat) from to val dat ctx &&
                        newValue ctx == oldValue ctx

```


A Readable and Computable Formalization of the Streamlet Consensus Protocol

Mauro Jaskelioff ✉ 

Input Output, Rosario, Argentina

Orestis Melkonian ✉ 

Input Output, London, UK

James Chapman ✉ 

Input Output, London, UK

Abstract

Consensus protocols are the fundamental building block of blockchain technology. Hence, correctness of the consensus protocol is essential for the construction of a reliable system. In the past few years, we saw the introduction of a myriad of new protocols of the BFT family of consensus protocols. The Streamlet protocol is one of these new protocols, which while not the fastest, it is certainly the simplest one.

In order to have strong guarantees for the protocol and its implementations we want to obtain formalizations that are readable enough to be used to communicate between formalizers and implementors, that have a mechanized proof of correctness and that can support the testing of implementations.

We present a readable and computable formalization of the Streamlet protocol in Agda, provide a mechanization of its proof of consistency, and show how one may use the formalization for testing implementations of it.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Logic and verification; Theory of computation → Program specifications

Keywords and phrases blockchain, Streamlet, consensus, formal verification, Agda

Digital Object Identifier 10.4230/OASICS.FMBC.2025.7

Supplementary Material *Software (Zenodo archive)*: <https://doi.org/10.5281/zenodo.15101644> [20]

Software (Source Code): <https://github.com/input-output-hk/formal-streamlet> [19]
archived at `swh:1:dir:70b9f1e274a05bad6f0e9fd5fe4e0f70033f503f`

1 Introduction

Consensus protocols are the fundamental building block of blockchain technology. Any mistake in their design or implementation could result in huge losses. Therefore, it is imperative to provide as strong guarantees as possible to ensure their correctness.

Consensus protocols can be *permissioned* or *permissionless*. Nakamoto-style consensus protocols are permissionless (all participants can be part of the decision process), while classical protocols like BFT [18] are permissioned (a few designated ones make the decision). With the advent of proof-of-stake blockchains, permissioned protocols can be adapted to work in a blockchain setting: a committee is formed based on the stake of all participants, which makes all decisions until a new committee is designated.

Consensus protocols have been around for a long time [18, 17]. In the past few years, we saw the introduction of a myriad of new consensus protocols of the BFT family [8, 28, 13, 6, 2, 11]. Given that many published consensus algorithms have been shown to be incorrect [3, 25], before adopting one of these new protocols, we would like to have strong guarantees that the protocol is correct and that we can thoroughly test its implementations. Therefore, we



© Mauro Jaskelioff, Orestis Melkonian, and James Chapman;
licensed under Creative Commons License CC-BY 4.0

6th International Workshop on Formal Methods for Blockchains (FMBC 2025).

Editors: Diego Marmosoler and Meng Xu; Article No. 7; pp. 7:1–7:18

OpenAccess Series in Informatics




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

are interested in their formalization and mechanization of proof of correctness, as well as extracting computational content from the formalization in order to test implementations. Because the formalization is not only meant to be read by formal methods engineers, but also taken as a ground truth for implementors, another goal is for the formalization to be as readable as possible.

We conduct our work in a mechanized fashion using the Agda proof assistant [21]. Agda has a very flexible syntax that allows us to write readable specifications. Additionally, because Agda is based on constructive type theory, it is possible to use the specification to *compute* and obtain a testing mechanism.

In this work, we formalize the STREAMLET consensus protocol [7]; while not the fastest out of these new BFT-style protocols, it is certainly the simplest one. Its simplicity makes it the ideal candidate to start investigating approaches to our goal of obtaining a **readable** formalization (§2 and §3), a **mechanized proof** of correctness (§4), and a means for **testing** implementations (§5).

The formalization is public [20] and we provide hyperlinks () throughout the paper:

<https://input-output-hk.github.io/formal-streamlet/>

2 **A general formal model for consensus protocols**

Consensus protocols are protocols for distributed system, and therefore consist of several nodes, each with their local state, sending messages across a network. The protocol itself is described by the behavior of these nodes, but we need to model the complete system in order to state global properties, such as Consistency (§4) as global properties involve relations among the state of *different* nodes. Therefore in this section we present a rather general formalization of a complete system, with the specific behavior of the protocol abstracted away in a relation (the local-step relation) which describes how each node behaves.

The formalization is done in terms of a global-step relation whose only concerns are:


- describing how a local-step changes the local state of each node;
- describing what a dishonest node can do;
- modeling the network;
- modeling the passage of time.

These last two depend on the network model one uses, which might be asynchronous, synchronous, or partially-synchronous [12]. Streamlet, as most BFT protocols, rely on a partially-synchronous network. However this is not relevant for safety, which is our concern here, so we do not model message delays.

Following the Streamlet paper, we assume synchronized clocks and therefore only consider a discrete notion of time divided into *epochs*: $\text{Epoch} = \mathbb{N}$.

The adversarial behavior is needed, since in the consensus protocols we are considering it is assumed that certain nodes might be dishonest and will actively try to disrupt the expected behavior of reaching consensus.

2.1 **Assumptions**

 **Assumptions**


First, we postulate the necessary cryptography, as it is completely orthogonal to our concerns:

- A type of **Hashes** and an ideal hash function $\#$ with no collisions; we assume we can compute hashes on base types and type formers, such as natural numbers (\mathbb{N}), products (\times), sums (\oplus), and lists (**List**).
- A type of **Keys** and **Signatures**, as well as a way to **sign** any data and verify signatures.

The assumptions pertaining to the specific setup of the consensus protocol are:

- A fixed number of participants ($\text{nodes} : \mathbb{N}$) each assigned a unique identifier Pid .
- A (decidable) predicate $\text{Honest} : \text{Pid} \rightarrow \text{Type}$ that distinguishes between honest and dishonest nodes. We will later use the notion of a vector that stores information for each *honest* node (HonestVec), since we do not want to keep local state for dishonest participants. For a given vector xs , $xs @ p$ retrieves the state of honest participant p , while $xs @ p := y$ locally updates the state of p .
- Crucially, all honest participants (honestPids) should form a 2/3-majority. Hence we assume $\text{honest-majority} : 3 * \text{length honestPids} > 2 * \text{nodes}$.
- Each epoch has a designated leader given by $\text{epochLeader} : \text{Epoch} \rightarrow \text{Pid}$ (the leader is chosen at random via a hash function, but there is no need to model that here).
- Last, transactions ($\text{Transaction} : \text{Type}$) that comprise a block are kept entirely abstract.

2.2 Global state

[ Global.State]

The global-step relates one global state to the next. A global state consists of a collection of local states (§3.2), one for each *honest* node.


$\text{StateMap} = \text{HonestVec LocalState}$

Other than that, it records the current epoch, in-transit messages, and the whole history of previous messages.

```
record GlobalState : Type where
  field e-now      : Epoch          networkBuffer : List Envelope
  stateMap         : StateMap       history         : List Message
```

Dishonest nodes do not get a local state as we cannot assume anything about their state. Recording the history of messages is not needed to specify the behavior of honest nodes, but it has proven to be an invaluable tool for proving properties about it (§4). Furthermore, keeping the history is essential if we want to give adversaries the power to reuse and re-transmit signed messages sent in the past by honest participants. The network buffer is described in terms of an *envelope*: a pair of a message and its recipient. The initial global state starts at epoch 1 with no messages and initial local states.

2.3 The global-step relation

[ Global.Step]

The global-step is a relation between global states. It has a constructor for each of the concerns described in the previous section:

```
data  $\longrightarrow$  (s : GlobalState) : GlobalState  $\rightarrow$  Type where
  LocalStep : {l : Honest p}  $\rightarrow$  Deliver :
    (p  $\triangleright$  s.e-now  $\vdash$  s @ p  $\dashv$  m? ]  $\rightarrow$  ls') (env  $\in$  : env  $\in$  s.networkBuffer)  $\rightarrow$ 
     $\hline$ 
    s  $\longrightarrow$  broadcast p m? (s @ p := ls') s  $\longrightarrow$  deliverMsg s env  $\in$ 

  DishonestStep : AdvanceEpoch :
    •  $\dashv$  Honest p • NoSignatureForging m s  $\hline$ 
    s  $\longrightarrow$  broadcast p (just m) s s  $\longrightarrow$  advanceEpoch s
```

LocalStep delegates control to the local-step relation (§3.4) if an honest participant p makes a local step from the current global state s , optionally producing a message $m?$ and resulting in a new local state ls' , then the whole system transitions to a new global state obtained by broadcasting the message $m?$ and updating the local state of p to ls' . In case there is a message, **broadcast** will add it to history, as well as place envelopes addressed to each other node into the **networkBuffer**.

The **DishonestStep** rule applies to dishonest nodes, who can broadcast any message as long as they do not forge signatures, *i.e.* messages signed by honest participants have to be replayed from history, while messages signed by dishonest participants have no restrictions:

NoSignatureForging : $\text{Message} \rightarrow \text{GlobalState} \rightarrow \text{Type}$
NoSignatureForging $m\ s = \text{Honest}\ (m \bullet \text{pid}) \rightarrow m \in s.\text{history}$

The **Deliver** step takes any in-transit envelope and delivers it to its recipient. The new state after this step is obtained by removing the envelope from the network buffer and modifying the recipient's local state using the **deliverMsg** function. By design, this rule does not follow a queue order, allowing for messages to be delivered in an arbitrary order.

The **AdvanceEpoch** global step increments the current epoch (**advanceEpoch**) and notifies nodes to update their local state (**epochChange**, §3.2).

3 A formal model of the Streamlet consensus protocol


Onto our main object of study: the STREAMLET consensus protocol [7], aimed to provide an idealistic model for a recent class of protocols [28, 2, 13] that are geared towards the setting of proof-of-stake blockchains and follow a “streamlined” approach that does not require a distinction between happy path and fallback mode [16, 5], or single-shot consensus [18].

Since the generic scaffolding described in Section 2 applies to the case of STREAMLET, we only need to concern ourselves with the local behavior of a node.

Informal description. STREAMLET follows a very simple *propose-vote* paradigm: each epoch, a leader is elected and made responsible for *proposing* a new block, while honest nodes *vote* for these proposals. Once a block gets a majority of votes it becomes *notarized*, and any three adjacent notarized blocks *finalize* the chain up to the second block. Given that each node is only partially aware of the votes in the whole network, they each have their own perspective on which blocks are notarized and which chains they consider final.

Picking up our mechanization from Section 2, completing the protocol definition amounts to providing a specification of the local step used in the **LocalStep** rule of the global-step relation. But first, we have to define how blockchains are formed, the state information kept locally by honest nodes, as well as the precise definitions of notarization and finalization.

3.1 Blockchains

 **Local.Chain**

A *blockchain* consists of a sequence of blocks, where each *block* points to the hash of the block it extends, records its epoch, and carries a payload of transactions.

Chain = List Block
record Block : Type where
 constructor $\langle _, _, _ \rangle$
 field parentHash : Hash
 epoch : Epoch
 payload : List Transaction

Participants will typically communicate blocks alongside their signature ([SignedBlock](#)).


Not all chains are valid though: for any block extending the previous chain, their hashes should match and epochs have to be strictly increasing.¹

```
record _-connects-to_ (b : Block) (ch : Chain) : Type where
  field hashesMatch : b.parentHash ≡ ch #
  epochAdvances : b.epoch > ch.epoch
```

We express this inductively: starting from the empty blockchain, we extend it block-by-block, making sure the validity requirements are met.

```
data ValidChain : Chain → Type where
  [] : ValidChain []
   $\frac{\bullet \text{ValidChain } ch \quad \bullet b \text{ -connects-to- } ch}{\bullet \text{ValidChain } (b :: ch)}$ 
```

3.2 Local state

[ Local.State]

Each node keeps track of a local view consisting of the following:

- its current *phase*, either [Ready](#) or [Voted](#);
- an *inbox* of messages received from the network, but still not processed;
- a *database* of processed messages, as well as ones sent by this node;
- the (longest) blockchain this node considers *final*.

```
record LocalState : Type where
  field phase : Phase      inbox : List Message
  db       : List Message  final  : Chain
```

Initially, each node's state is empty and its phase set to [Ready](#). Once a node proposes/votes a proposal, it sets its phase to [Voted](#), which is reset to [Ready](#) at each [epochChange](#).

The node's *inbox* is populated with messages externally via the global step's [deliverMsg](#) (§2). A *message* is either a proposal or a vote of a [SignedBlock](#):

```
data Message : Type where
  Propose : SignedBlock → Message
  Vote    : SignedBlock → Message
```

It is possible that messages appear out-of-order in the database, therefore we need to define when a node “has seen” a (valid) blockchain in their list of messages.

```
data _chain-∈_ : Chain → List Message → Type where
  [] : _chain-∈_ []
   $\frac{\bullet \text{Any } (\lambda m \rightarrow b \equiv m.\text{block}) \text{ ms} \quad \bullet ch \text{ chain-}\in \text{ ms} \quad \bullet b \text{ -connects-to- } ch}{\bullet (b :: ch) \text{ chain-}\in \text{ ms}}$ 
```

¹ A chain's epoch (accessed via function [epoch](#)) is either the epoch of its most-recent block, or 0 for the empty “genesis” chain.

3.3 Finalization

Given a list of messages ms , we can now precisely specify when a block b is **notarized**: exactly when the nodes who have voted for this block form a majority (*i.e.* at least 2/3 of total participants).

```
votes : List Message → Block → List Message
votes ms b = filter (λ m → b  $\stackrel{?}{=}$  m • block) ms
```

```
NotarizedBlock : List Message → Block → Type
NotarizedBlock ms b = IsMajority (votes ms b)
```

A blockchain is notarized when all of its constituent blocks are, while a block b_3 finalizes its prefix chain whenever three blocks (b_1, b_2, b_3 in chronological order) with consecutive epoch numbers have been notarized.

```
NotarizedChain : List Message → Chain → Type
NotarizedChain ms ch = All (NotarizedBlock ms) ch
```

```
data FinalizedChain (ms : List Message) : Chain → Block → Type where
  Finalize :
```


- NotarizedChain ms ($b_3 :: b_2 :: b_1 :: ch$)
- b_3 .epoch \equiv suc (b_2 .epoch)
- b_2 .epoch \equiv suc (b_1 .epoch)

```
FinalizedChain ms ( $b_2 :: b_1 :: ch$ )  $b_3$ 
```

We will often care about a blockchain both occurring in a list of messages *and* being notarized, as well as being the longest one.

```
_notarized-chain-∈_ _longest-notarized-chain-∈_ : Chain → List Message → Type
ch notarized-chain-∈ ms = ch chain-∈ ms
    × NotarizedChain ms ch
ch longest-notarized-chain-∈ ms = ch notarized-chain-∈ ms
    × (∀ {ch'} → ch' notarized-chain-∈ ms → length ch' ≤ length ch)
```

3.4 The local-step relation

 Local.Step

We are finally ready to formally specify the behavior of an *honest* node, as an inductively defined relation between said node p , the current epoch e , the starting state ls , possibly a message m , and the resulting state ls' :

```
data _▷_⊢_—[_]→_ (p : Pid) (e : Epoch) (ls : LocalState) : Maybe Message → LocalState → Type where
```

The participant, epoch, and starting state are promoted to *parameters*² as they remain constant across the possible actions of the node, while the rest of the relation's arguments are kept as *indices*³ since they might vary across constructors of this datatype.

² <https://agda.readthedocs.io/en/v2.7.0.1/language/data-types.html#parametrized-datatypes>

³ <https://agda.readthedocs.io/en/v2.7.0.1/language/data-types.html#indexed-datatypes>

The first rule models the proposals made by the epoch leader:

ProposeBlock :

```

let  $L = \text{epochLeader } e$ 
     $b = \langle ch \# , e , txs \rangle$ 
     $m = \text{Propose } (\text{sign } p \ b)$ 
in
  •  $ls.\text{phase} \equiv \text{Ready}$            •  $ch \text{ longest-notarized-chain} \in ls.db$ 
  •  $p \equiv L$                        •  $\text{ValidChain } (b :: ch)$ 

```

$p \triangleright e \vdash ls \multimap [\text{just } m] \rightarrow \text{record } ls \{ \text{phase} = \text{Voted}; db = m :: ls.db \}$

At the **Ready** phase, the leader can vote for a (valid) block extending the longest notarized chain in their view. The phase is updated to **Voted** to avoid double proposals, and the leader signed the proposed block and broadcasts it to the other nodes in a **Propose** message.

Other nodes instead follow the second rule, where they vote for proposals by the leader:

VoteBlock :

```

let  $L = \text{epochLeader } e$ 
     $b = \langle ch \# , e , txs \rangle$ 
     $sb^L = \text{sign } L \ b$ 
     $m^L = \text{Propose } sb^L; m = \text{Vote } (\text{sign } p \ b)$ 
in
  •  $\forall (m \in : m^L \in ls.inbox) \rightarrow$            •  $p \neq L$ 
  •  $sb^L \notin \text{map } \_ \bullet \text{signedBlock } (ls.db)$        •  $ch \text{ longest-notarized-chain} \in ls.db$ 
  •  $ls.\text{phase} \equiv \text{Ready}$                        •  $\text{ValidChain } (b :: ch)$ 

```

$p \triangleright e \vdash ls \multimap [\text{just } m] \rightarrow \text{record } ls \{ \text{phase} = \text{Voted}; db = m :: m^L :: ls.db; inbox = ls.inbox _ m \in \}$

The hypotheses ensure that they vote for the *first* proposal they have seen, as long as it has not been registered in their database and is a valid extension to the longest blockchain in their view. The node also signs the voted block and broadcasts it via a **Vote** message. Again, the phase is updated accordingly to avoid duplicate votes.

While the previous two rules modeled the propose-vote paradigm employed by STREAMLET, the next rule facilitates the message exchange between nodes by providing the counterpart to the **Deliver** global step that populates inboxes:

RegisterVote : let $m = \text{Vote } sb$ in

```

  •  $\forall (m \in : m \in ls.inbox) \rightarrow$ 
  •  $sb \notin \text{map } \_ \bullet \text{signedBlock } (ls.db)$ 

```

$p \triangleright e \vdash ls \multimap [\text{nothing}] \rightarrow \text{record } ls \{ db = m :: ls.db; inbox = ls.inbox _ m \in \}$

Concretely, the node moves **Vote** messages from their inbox to their local database, as long as this vote has not been registered before (to avoid duplicates).

Finally, a node can finalize a valid chain they have seen thus far, as long as the finalization conditions of Section 3.3 are obeyed:

FinalizeBlock : $\forall ch \ b \rightarrow$

```


  •  $\text{ValidChain } (b :: ch)$            •  $\text{FinalizedChain } (ls.db) \ ch \ b$ 

```

$p \triangleright e \vdash ls \multimap [\text{nothing}] \rightarrow \text{record } ls \{ \text{final} = ch \}$


Et voila! Putting together these local node actions with the global step of Section 2, we now have a fully mechanized, readable, and complete specification of STREAMLET.

4 Mechanizing Streamlet's consistency proof

 Properties

A consensus protocol is safe if it maintains *consistency*. Consistency means that two honest nodes cannot have divergent chains: their corresponding finalized chains must always be a prefix of, or equal to the other. It is perfectly fine for a node to lag behind, in which case its final chain would be a prefix of another.

4.1 Formalizing consistency

 Consistency

We formalize the consistency property (c.f. [7, Theorem 3]) as a predicate on **GlobalStates**.

Consistency : **StateProperty**

Consistency $s = \forall \{p \ p' \ b \ ch \ ch'\} \{ _ : \text{Honest } p \} \{ _ : \text{Honest } p' \} \rightarrow$

$\text{let } ms = (s @ p) .db ; ms' = (s @ p') .db \text{ in}$

- $(b :: ch) \text{ chain-}\in ms$ • $ch' \text{ notarized-chain-}\in ms'$
- **FinalizedChain** $ms \ ch \ b$ • $\text{length } ch \leq \text{length } ch'$

$ch \preceq ch'$


Here ms and ms' are the respective message databases of two honest nodes p and p' . Node p has finalized a chain ch and p' has seen a notarized chain ch' which is longer than ch . Consistency assures us that the finalized chain must be a prefix of or equal to ch' .

Further, we could prove how **FinalizedChains** correspond to the **final** fields of each node's state, but this is immediately derivable by inspecting the **FinalizeBlock** rule which makes sure only **FinalizedChains** are committed locally.

Also note that the **Consistency** property is slightly stronger than the informal description of consistency above, as we do not require the longer chain ch' to be part of a final chain; only notarization is required.

How do we prove consistency? We establish that the **StateProperty** is an *invariant*. That is, we prove that it holds for all states which are reachable from the initial one.

4.2 Proof infrastructure

 Global.Traces

We consider **traces** of the (global) step relation, defined as its *reflexive-transitive closure*.

data $_ * \leftarrow _ : \text{GlobalState} \rightarrow \text{GlobalState} \rightarrow \text{Type}$ **where**


$_ \triangleleft : \forall x \rightarrow$ <hr style="width: 50%; margin-left: 0;"/> $x * \leftarrow x$	$_ \langle _ \rangle \leftarrow _ : \forall z \rightarrow$ <div style="display: flex; justify-content: space-around; margin-top: 5px;"> • $z \leftarrow y$ • $y * \leftarrow x$ </div> <hr style="width: 50%; margin-left: 0;"/> $z * \leftarrow x$
--	---

A state property is a predicate on global states: **StateProperty** = $\text{GlobalState} \rightarrow \text{Type}$. In general, we are only interested in global states that are reachable from the initial global state s_0 , so one of the most useful state properties is reachability: **Reachable** $s = s * \leftarrow s_0$. A **StateProperty** is an invariant if it holds for every reachable global state.

Invariant : **StateProperty** $\rightarrow \text{Type}$

Invariant $P = \forall \{s\} \rightarrow \text{Reachable } s \rightarrow P \ s$

4.3 Example proof

 **Invariants.History**

Let us consider the **HistorySound** property to illustrate how we can use the tools we just introduced. It states that all messages in history are actually sent by their sender, and therefore are in the sender's database of messages.

HistorySound : **StateProperty**

HistorySound $s = \forall \{p\ m\} \{ _ : \text{Honest } p \} \rightarrow$
 $\bullet\ p \equiv m \bullet \text{pid} \quad \bullet\ m \in s.\text{history}$

$m \in (s @ p).\text{db}$

We prove that **HistorySound** is an **Invariant**, by induction on the reachability of the current state. The base case is trivially met, as history is empty in the initial state. In the case where a step $s \rightarrow$ is taken (transitioning from s to s'), we name IH the inductive hypothesis and do a case analysis on what kind of the step $s \rightarrow$ is.

historySound : **Invariant HistorySound**

historySound $(s' \langle s \rightarrow \mid s \rangle \leftarrow Rs) \{p\}\{m\} p \equiv m \in$
 $\text{with } IH \leftarrow \text{historySound } Rs \{p\}\{m\} p \equiv$
 $\text{with } s \rightarrow$

In the case of a step taken by a dishonest participant, we can use the inductive hypothesis, since the message necessarily has to be in history ($m \in$).⁴

| **DishonestStep** $_ \text{replay}$
 $\text{with } \gg m \in$
 $\dots \mid \gg \text{here refl rewrite } p \equiv = IH \text{ (replay it)}$
 $\dots \mid \gg \text{there } m \in = IH \text{ } m \in$

The most interesting case is when the step is a **LocalStep**. As is quite often, we need to consider whether the step is by the node p in question or by another node. In the former case, we rewrite with equality **lookup✓** : $(s @ p := ls') @ p \equiv ls'$ to simplify the goal and continue reasoning about p 's updated state ls' . In the latter case where p' is different than p , we instead rewrite with **lookup×** : $(s @ p' := ls') @ p \equiv s @ p$ and appeal to the induction hypothesis.

The proof of the **LocalStep** case proceeds by analyzing the four different cases for local step $ls \rightarrow$:

| **LocalStep** $\{p = p'\}\{mm\}\{ls'\} ls \rightarrow$
 $\text{with } \gg ls \rightarrow$
 $\dots \mid \gg \text{ProposeBlock } _ _ _ _$
 $\text{with } \gg m \in$
 $\dots \mid \gg \text{here refl rewrite } p \equiv \mid \text{lookup✓} = \text{here refl}$
 $\dots \mid \gg \text{there } m \in \text{with } p \stackrel{?}{=} p'$
 $\dots \mid \text{yes refl rewrite lookup✓} = \text{there } \$ IH \text{ } m \in$
 $\dots \mid \text{no } p \neq \text{rewrite lookup× } p \neq = IH \text{ } m \in$

We only show the case of **ProposeBlock**, as the other three cases are analogous. We also omit the cases of the global steps **Deliver** and **AdvanceEpoch** as they are trivial invocations of the inductive hypothesis, much like the case of **DishonestStep**.

⁴ The use of singleton types (\gg) is a technical artifact; it circumvents Agda's limitation to perform **with**-matching on a telescope variable.

4.4 Proving consistency

[Consistency]

The proof of consistency, although it follows the informal paper proof [7], required some changes to the proof structure. The (strengthened) consensus property considers the case of a finalized chain ch and a notarized one ch' , where the length of ch is less than or equal to the length of ch' . Because the longer chain can be shortened to the length of the shorter one, we can simplify consensus to the case where the two chains are of *equal length* (in the formalization, property [ConsistencyEqualLen](#)). Asking ch' to only be notarized is key in this reasoning, as any prefix of a notarized chain is notarized, while this is not the case for finalized chains, which require three consecutive epochs.

Having made this modification, we can follow the paper proof, which is based on two results: the [ConsistencyLemma](#) [7, Lemma 14] and [UniqueNotarization](#) [7, Lemma 10].

Unique notarization states that there can only be a unique notarization per epoch in honest view.

[UniqueNotarization](#) : [StateProperty](#)

[UniqueNotarization](#) $s = \forall \{p \ p' \ b \ b'\} \{ _ : \text{Honest } p \} \{ _ : \text{Honest } p' \} \rightarrow$
 $\text{let } ms = (s @ p) .db ; ms' = (s @ p') .db \text{ in}$
 $\bullet \text{NotarizedBlock } ms \ b \quad \bullet \text{NotarizedBlock } ms' \ b' \quad \bullet b .epoch \equiv b' .epoch$

$b \equiv b'$

The core of the consistency proof is the [ConsistencyLemma](#). It states that if some honest node sees a notarized chain with three adjacent blocks b_0 , b_1 , b_2 with consecutive epoch numbers e , $e + 1$, and $e + 2$, then there cannot be a conflicting block $b \neq b_1$ that also gets notarized in honest view at the same length as b_1 .

[ConsistencyLemma](#) : [StateProperty](#)

[ConsistencyLemma](#) $s = \forall \{p \ p' \ b_1 \ b_2 \ b \ ch \ ch'\} \{ _ : \text{Honest } p \} \{ _ : \text{Honest } p' \} \rightarrow$
 $\text{let } ms = (s @ p) .db ; ms' = (s @ p') .db \text{ in}$
 $\bullet (b_2 :: b_1 :: ch) \text{chain-}\in ms \quad \bullet (b :: ch') \text{notarized-chain-}\in ms'$
 $\bullet \text{FinalizedChain } ms \ (b_1 :: ch) \ b_2 \quad \bullet \text{length } ch' \equiv \text{length } ch$

$b_1 \equiv b$

As it often happens when formalizing a paper proof, many hidden details of the proof must be made apparent. An example of this is the [IncreasingEpochs](#) property which is a key to proving [ConsistencyLemma](#), but left implicit in the paper proof. It states that honest nodes cannot vote for a block of a previous epoch, *i.e.* the epochs of blocks being voted is *monotonic*. In other words, honest participants never backtrack on their votes, *i.e.* if an honest participant p'' votes for a block b extending chain ch , but also votes for another block b' now extending a longer chain ch' , then it must be the case that the epoch of b' is strictly greater than that of b .

[IncreasingEpochs](#) : [StateProperty](#)

[IncreasingEpochs](#) $s = \forall \{p \ p' \ p'' \ b \ ch \ b' \ ch'\} \{ _ : \text{Honest } p \} \{ _ : \text{Honest } p' \} \{ _ : \text{Honest } p'' \} \rightarrow$
 $\text{let } ms = (s @ p) .db ; ms' = (s @ p') .db \text{ in}$
 $\bullet p'' \in \text{votelds } ms \ b \quad \bullet p'' \in \text{votelds } ms' \ b' \quad \bullet \text{length } ch < \text{length } ch'$
 $\bullet b \text{-connects-to- } ch \quad \bullet b' \text{-connects-to- } ch'$

$b .epoch < b' .epoch$

where, $\text{votelds } ms \ b = \text{map } _ \bullet \text{pid } (\text{votes } ms \ b)$ computes the voters for block b in ms .

The use of the **history** field of **GlobalState** is essential for connecting local state properties across different nodes. For instance, we prove the general invariant of *message sharing*: if we find an honest vote in the database of another honest participant, then it is certainly also stored in the sender’s database.

MessageSharing : **StateProperty**

MessageSharing $s = \forall \{p \ p' \ b\} \{ _ : \text{Honest } p \} \{ _ : \text{Honest } p' \} \rightarrow$

$\text{let } ms = (s @ p) .db ; ms' = (s @ p') .db \text{ in}$

$p' \in \text{votelds } ms \ b$

$p' \in \text{votelds } ms' \ b$

Its proof relies on properties like **historySound** (presented in Section 4.3) and its inverse **historyComplete**, which ensures every local database is included in **history**.

5 Testing

One of the most crucial reasons for conducting our work in *constructive* type theory is to be able to compute with our specification: proof assistants of this sort – Agda included – typically provide facilities to *extract* one’s formalization to executable code.


While we have claimed to provide an *executable* specification of STREAMLET, we should clarify that this is only partly true due to the non-deterministic nature of the protocol. That is, the relational specification of Section 3 is *non-deterministic*, thus specifying a whole *set of implementations* that would be valid with respect to such a relation.

Furthermore, the assumptions made in Section 2.1 and left abstract for the rest of the formal development, should now be made concrete by instantiating all assumptions with actual implementations in order for extraction to executable code to make sense.

Therefore, we cannot hope to extract a full STREAMLET implementation out of our formal development, but there are still many constituent parts of our formalization that are indeed computable:

- We can prove that all of the logical propositions defined throughout the paper are indeed *decidable*. Proving that a proposition is decidable amounts to providing a *decision procedure* that answers whether the proposition holds or does not together a corresponding proof (§A).
- It is now possible to exhibit example traces of protocol execution without the need to explicitly discharge proof obligations for each rule invocation (§5.1). Traces manifest as proof derivations of the step relation, and all proof obligations are discharged by invoking the decision procedure that corresponds to each hypothesis, a technique known as *proof-by-computation* [27].
- Once extracted, the decision procedures enable us to test an actual implementation for *conformance* with respect to our mechanized semantics. To illustrate this point, we sketch a *trace verifier* that can validate traces randomly generated by an (unverified) implementation (§5.2).

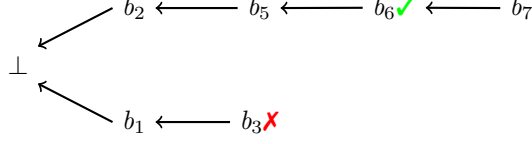
5.1 Example trace

 **Test.ExampleTrace**

One crucial step to allow for computation and extraction is to provide a concrete instantiation of the assumptions (§2.1), otherwise computation would get stuck on encountering such a postulate. To do so amounts to giving a term of type **Assumptions**; we use naive hash functions and signature schemes and restrict to a set of three participants **L**, **A**, **B** where **L** is chosen as the leader at every epoch.

7:12 A Readable and Computable Formalization of the Streamlet Consensus Protocol

We will demonstrate an execution trace corresponding to the running example of the original Streamlet paper [7, Figure 1], where out of two competing chains $b_2 \leftarrow b_5 \leftarrow b_6 \leftarrow b_7$ and $b_2 \leftarrow b_3$ only the top one finalizes its prefix chain up to block b_6 :



A block b_i is proposed on epoch i , thus the property of consistency mechanized in Section 4 makes it impossible for any extension of the bottom chain to be considered final anymore (due to the consecutive epochs of b_5 , b_6 , and b_7). The leaders makes proposals p_i for every block b_i and nodes vote for the same block with v_i , where \mathbb{A} exclusively votes for the top chain and \mathbb{B} for the bottom one. We are finally ready to make use of the proof automation of Appendix A to demonstrate an execution trace where b_6 eventually gets finalized:

```

begin
  initGlobalState
  →< Propose?  $\mathbb{L}$  [ ] [ ] > - leader proposes  $b_1$ 
  record { e-now = 1
    ; history = [  $p_1$  ]
    ; networkBuffer = [ [  $\mathbb{A}$  |  $p_1$  ] ; [  $\mathbb{B}$  |  $p_1$  ] ]
    ; stateMap = [ { -  $\mathbb{L}$  - } ( Voted , [  $p_1$  ] , [ ] , [ ] )
                  ; { -  $\mathbb{A}$  - } ( Ready , [ ] , [ ] , [ ] )
                  ; { -  $\mathbb{B}$  - } ( Ready , [ ] , [ ] , [ ] ) }
  →< Deliver? [  $\mathbb{B}$  |  $p_1$  ] >

  -
  →< Vote?  $\mathbb{B}$  [ ] [ ] > -  $b_1$  becomes notarized
  record { e-now = 1
    ; history = [  $v_1$  ;  $p_1$  ]
    ; networkBuffer = [ [  $\mathbb{A}$  |  $p_1$  ] ; [  $\mathbb{L}$  |  $v_1$  ] ; [  $\mathbb{A}$  |  $v_1$  ] ]
    ; stateMap = [ ( Voted , [  $p_1$  ] , [ ] , [ ] )
                  ; ( Ready , [ ] , [ ] , [ ] )
                  ; ( Voted , [  $v_1$  ;  $p_1$  ] , [ ] , [ ] ) }

  :
  →< Propose?  $\mathbb{L}$  [  $b_6$  ;  $b_5$  ;  $b_2$  ] [ ] > - leader proposes  $b_7$ 
  :
  →< Vote?  $\mathbb{A}$  [  $b_6$  ;  $b_5$  ;  $b_2$  ] [ ] > -  $b_7$  becomes notarized
  :
  →< Finalize?  $\mathbb{A}$  [  $b_6$  ;  $b_5$  ;  $b_2$  ]  $b_7$  > -  $b_6$  becomes finalized
  record { e-now = 7
    ; history = [  $v_7$  ;  $p_7$  ;  $v_6$  ;  $p_6$  ;  $v_5$  ;  $p_5$  ;  $v_3$  ;  $p_3$  ;  $v_2$  ;  $p_2$  ;  $v_1$  ;  $p_1$  ]
    ; networkBuffer = -
    ; stateMap = [ ( Voted , _ , [ ] , [ ] )
                  ; ( Voted , _ , [ ] , [  $b_6$  ;  $b_5$  ;  $b_2$  ] )
                  ; ( Ready , _ , [ ] , [ ] ) }
  ◀

```

For the sake of brevity, we have elided many intermediate steps and states, but it should still be clear that the above demonstrates a provably correct derivation chain of steps, at the end of which node \mathbb{A} has finalized the top chain up to b_6 .

5.2 Conformance Testing

 TraceVerifier

The question remains: can we leverage the functions extracted from our STREAMLET mechanization in any other way outside the formalization itself?

We believe there is a strong case to be made for a **conformance testing** approach, where there already exists an implementation that is developed independently and is not formally verified, and we wish to ensure that it *conforms* to the formal specification. The central properties and invariants we have identified in Section 4 can inform the behavior being tested in the actual implementation. In particular, this seems to be an excellent fit to *property-based testing* [9], since the types of our theorems should easily translate to properties embedded in the implementation language.

This however relies on randomly generating traces of execution to feed as input to said tests. One possible way to bridge the gap between our Agda formal model of STREAMLET and its actual implementation is to extract a *trace verifier* that decides whether a trace generated by the implementation indeed respects the semantics of the global-step relation.

We first need to define a simple interface of actions, which will comprise the traces we communicate to external systems:

```
data Action : Type where
  Propose      : Pid → Chain → List Transaction → Action
  Vote         : Pid → Chain → List Transaction → Action
  RegisterVote : Pid → ℕ → Action
  FinalizeBlock : Pid → Chain → Block → Action
  DishonestStep : Pid → Message → Action
  Deliver      : ℕ → Action
  AdvanceEpoch : Action
```

```
Actions = List Action
```

Actions provide the necessary input to make the rule selection *deterministic*: there is no ambiguity as to which rule applies at any given point. Equivalently, you can think of the action data being the same as the input we had to provide in the proof-automated steps of the example trace in Section 5.1.

Not all sequences of actions are valid though; we define a predicate that precisely characterizes the sequences that correspond to a valid trace: `ValidTrace : Actions → Type`, which relies on an evaluator `[[_]]` that executes a given action and returns the next state. We have omitted their definitions as they are just trivial repetitions of the rules: validity can be read off the rule hypotheses, while the next evaluated state can be read off each rule's conclusion.

We then provide a decision procedure to decide whether a sequence of actions is indeed valid, i.e. a *trace verifier*: `instance Dec-ValidTrace : ∀ {tr} → ValidTrace tr ?`. Although the correspondence between the trace verifier and the relational semantics of the previous sections is clear from the use of the same logical propositions, there is still no *formal* connection between them. We bridge this gap by proving the trace verifier *sound* and *complete* w.r.t. the global-step relation:

ValidTrace-sound :

```
(tr : ValidTrace αs) →
  [[ tr ]] *← initGlobalState
```

ValidTrace-complete :

```
(st : s *← initGlobalState) →
  ∃ λ (tr : ValidTrace (getLabels st)) →
    [[ tr ]] ≡ s
```

Soundness amounts to reconstructing a logical trace from a sequence of (valid) actions, while completeness ensures that all logical traces have a corresponding sequence of actions that results in the same state after execution.

6 Related Work

Given the importance of having strong guarantees for consensus protocols, it is no wonder that there are many formalizations of them; we are especially interested in ones that are conducted in an interactive proof assistant [24, 22, 1, 26, 4, 15]. However the objectives of each of these are slightly different, leading to different design choices.

Thomsen and Spitters [26] formalize a Nakamoto-style consensus algorithm (essentially Ouroboros Praos [10]) in Coq, and prove both safety and liveness. Their local state is based on an abstract block tree structure allowing for greater flexibility, while ours is a concrete list of messages received. This work inspired us to include **history** in the global state.

Carr *et al.* [4] formalize the LibraBFT protocol (which is based on Hotstuff [28]) in Agda and prove safety. Being a formalization of a BFT protocol in Agda, this work is closest to ours, however we had slightly different objectives, resulting in different approaches. Readability was not one of the main concerns so the model favors abstraction, allowing to potentially conclude safety from the properties of the instantiations of the abstract structures. Our model is more concrete and direct, but is more suitable for extracting a testing oracle.

Another line of research is concerned with generic frameworks for building consensus algorithms [14, 29, 23]; these however heavily rely on high-level abstractions, making it harder to relate a formalized protocol to the informal paper description. Here, we opt for a more direct approach.

7 Conclusion

We have presented our formalization of the BFT protocol STREAMLET using the Agda proof assistant. Using a relational approach for the step semantics, we have obtained a readable specification and proven consistency (safety). By implementing decision procedures we have made it possible to easily write verified traces of execution, and shown a path towards conformance testing.


References

- 1 Musab A. Alturki, Jing Chen, Victor Luchangco, Brandon M. Moore, Karl Palmskog, Lucas Peña, and Grigore Rosu. Towards a verified model of the Algorand consensus protocol in Coq. *CoRR*, abs/1907.05523, 2019. [arXiv:1907.05523](#).
- 2 Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017. [arXiv:1710.09437](#).
- 3 Christian Cachin and Marko Vukolic. Blockchain consensus protocols in the wild. *ArXiv*, abs/1707.01873, 2017. [arXiv:1707.01873](#).
- 4 Harold Carr, Christa Jenkins, Mark Moir, Victor Cacciari Miraldo, and Lisandra Silva. Towards formal verification of HotStuff-based byzantine fault tolerant consensus in Agda. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors, *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, volume 13260 of *Lecture Notes in Computer Science*, pages 616–635. Springer, 2022. [doi: 10.1007/978-3-031-06773-0_33](#).
- 5 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In Margo I. Seltzer and Paul J. Leach, editors, *Proceedings of the Third USENIX Symposium on Operating Systems*

- Design and Implementation (OSDI)*, New Orleans, Louisiana, USA, February 22-25, 1999, pages 173–186. USENIX Association, 1999. URL: <https://dl.acm.org/citation.cfm?id=296824>.
- 6 Benjamin Y. Chan and Rafael Pass. Simplex consensus: A simple and fast consensus protocol. In Guy N. Rothblum and Hoeteck Wee, editors, *Theory of Cryptography - 21st International Conference, TCC 2023, Taipei, Taiwan, November 29 - December 2, 2023, Proceedings, Part IV*, volume 14372 of *Lecture Notes in Computer Science*, pages 452–479. Springer, 2023. doi:10.1007/978-3-031-48624-1_17.
 - 7 Benjamin Y. Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. In *AFT '20: 2nd ACM Conference on Advances in Financial Technologies, New York, NY, USA, October 21-23, 2020*, pages 1–11. ACM, 2020. doi:10.1145/3419614.3423256.
 - 8 T-H. Hubert Chan, Rafael Pass, and Elaine Shi. Pala: A simple partially synchronous block-chain. *IACR Cryptol. ePrint Arch.*, 2018:981, 2018. URL: <https://api.semanticscholar.org/CorpusID:53238268>.
 - 9 Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.
 - 10 Bernardo Machado David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros Praos: An adaptively-secure, semi-synchronous proof-of-stake protocol. *IACR Cryptology ePrint Archive*, 2017:573, 2017. URL: <http://eprint.iacr.org/2017/573>.
 - 11 Isaac Doidge, Raghavendra Ramesh, Nibesh Shrestha, and Joshua Tobkin. Moonshot: Optimizing chain-based rotating leader bft via optimistic proposals, 2024. doi:10.48550/arXiv.2401.01791.
 - 12 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988. doi:10.1145/42282.42283.
 - 13 Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and Ditto: Network-adaptive efficient consensus with asynchronous fallback. In Ittay Eyal and Juan A. Garay, editors, *Financial Cryptography and Data Security - 26th International Conference, FC 2022, Grenada, May 2-6, 2022, Revised Selected Papers*, volume 13411 of *Lecture Notes in Computer Science*, pages 296–315. Springer, 2022. doi:10.1007/978-3-031-18283-9_14.
 - 14 Wolf Honoré, Longfei Qiu, Yoonseung Kim, Ji-Yong Shin, Jieung Kim, and Zhong Shao. AdoB: Bridging benign and byzantine consensus with atomic distributed objects. *Proc. ACM Program. Lang.*, 8(OOPSLA1), April 2024. doi:10.1145/3649826.
 - 15 Elliot Jones and Diego Marmosoler. Towards Mechanised Consensus in Isabelle. In Bruno Bernardo and Diego Marmosoler, editors, *5th International Workshop on Formal Methods for Blockchains (FMBC 2024)*, volume 118 of *Open Access Series in Informatics (OASICS)*, pages 4:1–4:22, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASICS.FMBC.2024.4.
 - 16 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998. doi:10.1145/279227.279229.
 - 17 Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)* 32, 4 (Whole Number 121, December 2001), pages 51–58, 2001.
 - 18 Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982. doi:10.1145/357172.357176.
 - 19 Orestis Melkonian and Mauro Jaskelioff. input-output-hk/formal-streamlet. Software, swhId: swh:1:dir:70b9f1e274a05bad6f0e9fd5fe4e0f70033f503f (visited on 2025-04-14). URL: <https://github.com/input-output-hk/formal-streamlet>, doi:10.4230/artifacts.23005.
 - 20 Orestis Melkonian and Mauro Jaskelioff. Agda formalization of the Streamlet protocol. <https://github.com/input-output-hk/formal-streamlet>, March 2025. doi:10.5281/zenodo.15101644.

- 21 Ulf Norell. Dependently typed programming in Agda. In *International School on Advanced Functional Programming*, pages 230–266. Springer, 2008. doi:10.1007/978-3-642-04652-0_5.
- 22 George Pirlea and Ilya Sergey. Mechanising blockchain consensus. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 78–90. ACM, 2018. doi:10.1145/3167086.
- 23 Longfei Qiu, Yoonseung Kim, Ji-Yong Shin, Jieung Kim, Wolf Honoré, and Zhong Shao. LiDO: Linearizable byzantine distributed objects with refinement-based liveness proofs. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024. doi:10.1145/3656423.
- 24 Vincent Rahli, Ivana Vukotic, Marcus Völz, and Paulo Jorge Esteves Veríssimo. Velisarios: Byzantine fault-tolerant protocols powered by Coq. In Amal Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 619–650. Springer, 2018. doi:10.1007/978-3-319-89884-1_22.
- 25 Pierre Tholoniati and Vincent Gramoli. Formal verification of blockchain byzantine fault tolerance. In *Handbook on Blockchain*, pages 389–412. Springer, 2022. doi:10.1007/978-3-031-07535-3_12.
- 26 Søren Eller Thomsen and Bas Spitters. Formalizing Nakamoto-style proof of stake. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*, pages 1–15. IEEE, 2021. doi:10.1109/CSF51468.2021.00042.
- 27 Paul Van Der Walt and Wouter Swierstra. Engineering proof by reflection in Agda. In *Symposium on Implementation and Application of Functional Languages*, pages 157–173. Springer, 2012. doi:10.1007/978-3-642-41582-1_10.
- 28 Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC '19*, pages 347–356, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293611.3331591.
- 29 Qiyuan Zhao, George Pirlea, Karolina Grzeszkiewicz, Seth Gilbert, and Ilya Sergey. Compositional verification of composite byzantine protocols. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS '24*, pages 34–48, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3658644.3690355.

A Decidability

 Decidability

For any given proposition P , proving that it is decidable amounts to providing a program of type `Dec P` that decides whether the proposition holds (yes) or does not (no):

<pre>data Dec (P : Type) : Type where yes : P → Dec P no : ¬ P → Dec P</pre>	<pre>record _?? (P : Type) : Type where field dec : Dec P !_! : ∀ P → { P ?? } → Dec P ! _ ! = dec</pre>
---	---

We collect all decidable propositions in a typeclass (`??`), and use the notation `! P !` to acquire the corresponding decision procedure by *instance search*.⁵

⁵ <https://agda.readthedocs.io/en/v2.7.0/language/instance-arguments.html>

Decidability of basic types and type formers is already defined in the standard library:

```
instance
  Dec-⊥ : ⊥ ??
  Dec-⊥ .dec = no λ()

  Dec-⊤ : ⊤ ??
  Dec-⊤ .dec = yes tt

module _ {A : A ??} {B : B ??} where instance
  Dec-→ : (A → B) ??
  Dec-→ .dec with i A i | i B i
  ... | no ¬a | _ = yes λ a → contradict (¬a a)
  ... | yes a | yes b = yes λ _ → b
  ... | yes a | no ¬b = no λ f → ¬b (f a)

  Dec-× : (A × B) ??
  Dec-× .dec with i A i | i B i
  ... | yes a | yes b = yes (a , b)
  ... | no ¬a | _ = no λ (a , _) → ¬a a
  ... | _ | no ¬b = no λ (_, b) → ¬b b

  Dec-⊕ : (A ⊕ B) ??
  Dec-⊕ .dec with i A i | i B i
  ... | yes a | _ = yes (inj₁ a)
  ... | _ | yes b = yes (inj₂ b)
  ... | no ¬a | no ¬b = no λ where (inj₁ a) → ¬a a; (inj₂ b) → ¬b b
```

Since these would take care of the most trivial combinations of other properties, we are only tasked with proving decidability of only the *interesting* propositions that we introduced in this paper that cannot be trivially solved by instance search.

Let us illustrate with the example of deciding whether a chain has been finalized:

```
instance
  Dec-Finalized : ∀ {ms ch b} → FinalizedChain ms ch b ??
  Dec-Finalized {ch = ch} .dec
    with ch
  ... | [] = no λ ()
  ... | _ :: [] = no λ ()
  ... | _ :: _ :: _
    with dec | dec | dec
  ... | yes p | yes q | yes r = yes (Finalize p q r)
  ... | no ¬p | _ | _ = no λ where (Finalize p _ _) → ¬p p
  ... | _ | no ¬q | _ = no λ where (Finalize _ q _) → ¬q q
  ... | _ | _ | no ¬r = no λ where (Finalize _ _ r) → ¬r r
```

We first check whether the chain in question does not even have three blocks, in which case we immediately decide the proposition does not hold. We then decide whether the finalization conditions of Section 3.3 hold and respond accordingly. Notice that we do not even have to state the propositions we are deciding in the process; the type system takes care of this for us!

Once all propositions that are explicitly or implicitly used in rule hypotheses have been proven decidable, we can provide an alternative version of the rules where the user no longer needs to provide explicit proofs in the case of *closed* examples (*i.e.* ones without any free variables). Instead, the corresponding decision procedures automatically discharge the proof obligations, otherwise we would get a typechecking error that the proposition under question

is not true. Concretely, we prefix a proposition P with `auto:` to invoke its decision procedure; since computation will not block on any variables, we will eventually compute either a `yes` and replace the obligation with the trivial unit type (\top), or trigger an error by returning the absurd empty type (\perp) which can never be discharged.

```

auto:_ : (P : Type) → { P ?? } → Type
auto: P with  $\iota$  P  $\iota$ 
... | yes _ =  $\top$ 
... | no _ =  $\perp$ 

```

As an example, the `ProposeBlock` rule would remain mostly unchanged, except that all logical hypotheses are annotated as *implicit arguments*⁶ and prefixed with `auto:` to trigger the aforementioned proof-by-computation.

```

Propose? : ∀ ch txs → let
  ...
  ls' = proposeBlock ls m in
  { _ : p ≡ L }
  { _ : auto: ls .phase ≡ Ready }
  { _ : auto: ch longest-notarized-chain-∈ ls .db }
  { _ : auto: ValidChain (b :: ch) } →
  -----
  s → broadcast L (just m) (updateLocal p ls' s)

```

⁶ <https://agda.readthedocs.io/en/v2.7.0/language/implicit-arguments.html>

Formal Verification of a Fail-Safe Cross-Chain Bridge

Filip Marić 

University of Belgrade, Serbia

Bernhard Scholz 

Sonic Research, Sydney, Australia

Pavle Subotić 

Sonic Research, Belgrade, Serbia

Abstract

Cross-chain bridges are financial services that interconnect blockchains. High monetary values flow through these bridges, and their security must be safeguarded. However, designing real-world cross-chain bridges is a difficult endeavor. Due to blockchain's closed-world nature, tokens cannot be transferred from a sender to a receiver chain; on the contrary, they need complex logic that maintains an equilibrium on both chains, even if either the chains or the bridge fail. This paper formally verifies a model of a novel fail-safe cross-chain bridge to ensure correctness. We define formal requirements and prove the bridge is safe using the Isabelle/HOL proof assistant.

2012 ACM Subject Classification Software and its engineering → Software verification

Keywords and phrases Cross-Chain Bridge, Formal Verification, Logic, Security

Digital Object Identifier 10.4230/OASICS.FMBC.2025.8

Supplementary Material

Software (Source Code): https://github.com/filipmaric/bridge_formalization

Acknowledgements We want to thank Sonic Labs engineers Jan Kalina and Jirka Malek for designing and implementing the fail-safe bridge. Without their insights and explanations, we would not have been able to build the model.

1 Introduction

The blockchain ecosystem has rapidly evolved over the last decade, most notably with the rise of Ethereum [32] and Bitcoin [23]. At the end of 2024, the top 636 smart contract blockchains had a market cap of 2.9 trillion dollars [11].

As the blockchain ecosystem continues to grow, it has become increasingly heterogeneous, consisting of blockchains with unique designs and varying characteristics (proof-of-work, proof-of-stake, costs, performance, programming languages, etc.). Thus, each blockchain provides users with a diverse experience, security guarantees, and financial incentives, making it unlikely that a single dominant blockchain design will emerge [7].

Unfortunately, without a method of communication between blockchains, the advantages of a multichain world are limited. Since blockchains have *closed-world assumption* about their tokens, digital assets are not universal entities and cannot exist outside of their chains.

To this end, decentralized cross-chain bridges have been proposed (e.g., [33]) to provide interoperability between blockchains. Given two blockchains, a cross-chain bridge allows users to transfer tokens from a *sender chain* C_s to a *receiver chain* C_r . When an asset is transferred from C_s to C_r , the asset is *locked* on C_s , and a new asset is *minted* on C_r , representing the original asset on C_s . When the newly created asset is transferred back to chain C_s , the corresponding asset on C_r is *burned*, and the locked asset on C_s is *released*.



© Filip Marić, Bernhard Scholz, and Pavle Subotić;
licensed under Creative Commons License CC-BY 4.0

6th International Workshop on Formal Methods for Blockchains (FMBC 2025).

Editors: Diego Marmosoler and Meng Xu; Article No. 8; pp. 8:1–8:18

OpenAccess Series in Informatics



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

Smart contracts on both blockchains implement this protocol, and a *relay network* is used to establish communication between the two chains.

Given the various security exploits targeting bridges [18, 19, 34], designing a safe bridge is paramount. Developing a safe bridge involves guaranteeing several *safety and liquidity invariants*. For instance, we want to ensure that there is always a 1-to-1 relationship between locked and minted assets. Otherwise, double-spending [17] attacks may arise. Liquidity invariants guarantee that the user can always perform an action to access their asset. The bridge has *fail safety*, i.e., that all tokens will be unlocked in a blockchain C_s if blockchain C_r irrecoverably crashes.

Standard audit and testing methods can be employed to increase the likelihood of safety on the bridge. However, these methods, due to their inability to account for all possible behaviors, are insufficient. This is evidenced by the fact that bridges have become notorious targets for hackers. As of 2024, bridge-related hacks cost more than \$2.8 billion, representing approximately 40% of all hacks in the Web3 [16]. Given the potential for sizable losses, relying only on ad-hoc testing and auditing techniques is perilous. A more rigorous method to ensure bridge safety is to employ *formal verification*. Here, we construct mathematical objects describing the bridge behavior and mathematical proofs to ensure the bridge protocol is correct for all possible inputs and configurations.

This paper presents an industrial case study in which we formally prove the safety of the formal abstract model of SONIC GATEWAY. This cross-chain bridge provides interoperability between the Ethereum and SONIC blockchains. While formal methods have been employed for individual blockchain components [6], to the best of our knowledge, our work is the first to prove the safety of a fail-safe cross-chain bridge model formally.

We formalize the bridge within the interactive theorem prover Isabelle/HOL [25] and establish safety by identifying and proving several fundamental safety properties. We perform our verification using a *stepwise refinement approach* [31, 9]: We start with an abstract model and then refine it by adding more and more implementation details. This process has verified the final SONIC GATEWAY design and contributed to the design of cross-chain bridges by identifying essential properties that are pervasive to all fail-safe cross-chain bridges.

Our contributions are summarized as follows:

1. A detailed formal description of the fail-safe SONIC GATEWAY cross-chain bridge including safety properties,
2. A formal specification and correctness proof for the SONIC GATEWAY cross-chain bridge,
3. Experimental evaluation and experience of the verification effort.

This paper is structured as follows. Section 2 describes the SONIC GATEWAY bridge. In Section 3, we detail the formal specification of the SONIC GATEWAY bridge, and in Section 4, we provide a corresponding proof of its safety. We evaluate the verification process in Section 5, and Section 6 details related work. We conclude in Section 7.

2 The Fail Safe Bridge

This section describes the SONIC GATEWAY fail-safe cross-chain bridge. A detailed example of bridge operations is given in Appendix A. The bridge is illustrated in Figure 1. Here, the bridge operates on a pair of blockchains C_s and C_r , where C_s is the sender chain, used to deposit (aka. lock) tokens, and C_r is the receiver chain, used to claim (aka. mint) tokens. The blockchain C_s has a smart contract $SC_{deposit}$, and C_r has a smart contract SC_{claim} . For simplicity, we assume that users transfer fungible tokens from an ERC20 [29] contract

T_{orig} (although these could also be native tokens of C_s) to C_r , where they are represented by tokens on a dedicated ERC20 contract T_{mint} .

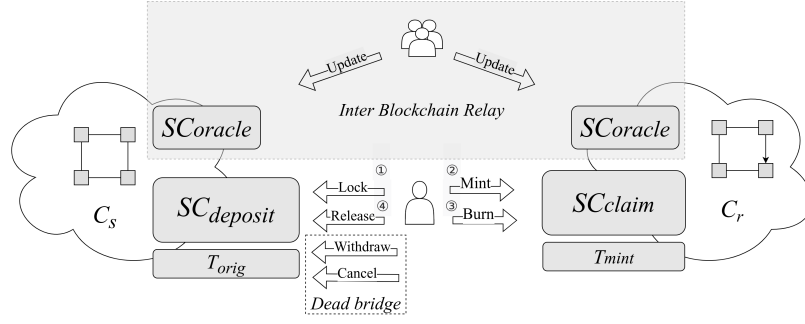
2.1 Token Exchange

During normal bridge operations, users can exchange original tokens on C_s for minted tokens on C_r . At any time, the minted tokens can be converted back to the original tokens in C_s . Token transfers from C_s to C_r operate using a pair of transactions denoted as *Lock* and *Mint*. Assuming that a user already has at least $v T_{orig}$ tokens available on C_s , the user invokes a *Lock* transaction on C_s (step ①) to transfer $v T_{orig}$ tokens from their balance to the $SC_{deposit}$ balance. To verify that a *Lock* transaction has been successfully executed, $SC_{deposit}$ keeps a log *locks*, containing data about all executed lock transactions (a unique transaction ID is mapped to data about the caller, token, and amount, usually combined into a hash value). Before $v T_{mint}$ tokens can be issued by SC_{claim} on C_r (using a *Mint*, step ②), the solvency needs to be ensured. SC_{claim} issues new tokens only if the user has locked tokens on $SC_{deposit}$. This requires SC_{claim} (operating on C_r) to read the state of $SC_{deposit}$ from a different blockchain (C_s), which it cannot do directly. To enable this, the bridge uses a Merkle proof (a path of a Merkle tree [15]), which acts as a trusted witness for blockchain C_r of the existence of a specific transaction (or its effect) at a block height t on blockchain C_s . This Merkle proof is an argument for the *Mint* transaction. Simultaneously, the state root hash of the world state of C_s is shipped via the trusted channel, which we call the *Inter-Blockchain Communication (IBC) Relay*. We assume that a special contract SC_{oracle} keeps track of state root hashes and that a consensus of trusted users regularly updates it by invoking *Update* transactions. When executing a *Mint* transaction, the contract SC_{claim} on C_r verifies that the *Lock* transaction took place on C_s at block height t using the Merkle proof and state root hash of C_s read from SC_{oracle} , which confirm that the log *locks* contains valid data about the *Lock* transaction. If the proof verification succeeds, the user's balance of T_{mint} is increased by v (v tokens are minted “out of thin air”). To prevent double claims of the same deposit, SC_{claim} keeps a directory *mints* of minted deposits. Once the user obtains T_{mint} tokens on C_r , they can trade with them as with all other ERC20 tokens.

At any point, a user with T_{mint} tokens on C_r can exchange them for the same amount of T_{orig} on C_s . This is done by performing a sequence of *Burn* and *Release* operations, which are pretty similar to *Lock* and *Mint*, so we do not describe them in detail. The user invokes a *Burn* transaction (step ③), which removes v tokens from their balance on T_{mint} , and the information about this is logged in the *burns* log on SC_{claim} on C_r . After SC_{oracle} has been informed about the root of the world state of C_r , the user can invoke a *Release* transaction (step ④) in C_s that contains a Merkle tree proof confirming that the *Burn* transaction has been successfully executed. Once the proof verification succeeds, $v T_{orig}$ tokens are transferred from the $SC_{deposit}$ balance to the user's one.

2.2 Fail-Safety

What is unique to the SONIC GATEWAY is that it implements a mechanism that enables users to retrieve their funds on C_s , even if the bridge goes out of operation (if SC_{claim} or T_{mint} contracts or even the C_r blockchain stop operating). The bridge is declared dead once the SC_{oracle} contract (on C_s) does not execute a state root update transaction within a fixed amount of time (e.g., 7 days). The bridge is declared dead by setting a *dead* flag in the $SC_{deposit}$ contract and by copying the last known state root of C_r from SC_{oracle} to $SC_{deposit}$. Once the bridge is declared dead, this status cannot be undone. When the bridge



■ **Figure 1** The SONIC GATEWAY cross-chain bridge between blockchains C_s and C_r .

is dead, users retrieve their assets using *Withdraw* and *Cancel* transactions. Furthermore, note that *Release* transactions do not depend on C_r and are executed similarly, regardless of whether the bridge is dead.

The last known state root hash of C_r enables verifying the user's balance of T_{mint} . If the user had v tokens of T_{mint} on C_r at the point when the last state root was given to the SC_{oracle} , they can generate a Merkle Tree proof for that and invoke a *Withdraw* transaction on $SC_{deposit}$. When the proof verification succeeds, v tokens of T_{orig} are transferred from the $SC_{deposit}$ contract to the user's balance. Every withdrawal transaction is logged in the *withdrawals* array to prevent double withdrawals.

Finally, it is possible that the user has deposited and locked some v tokens of T_{orig} on C_s , but has not yet claimed them, or they claimed them but there was no state root update after that claim, so there are no corresponding tokens on T_{mint} that they could retrieve using the *Withdraw* operation. A *Cancel* transaction must be used in such cases. With this transaction, the user provides a *Lock* transaction ID (which can be easily checked to exist using the *locks* log of $SC_{deposit}$ on C_s) and the Merkle tree proof that there was no corresponding *Mint* on SC_{claim} on C_r in the last known state (verified by checking the *mints* log). Once the *Cancel* transaction is successful, to prevent double cancellation, the information about the canceled *Lock* transaction is removed from the *locks* log.

3 Specification

This section briefly outlines the Isabelle/HOL specification of the Sonic fail-safe bridge. Due to space constraints, many details are omitted. For full details, we refer the reader to the Isabelle/HOL source, available at https://github.com/filipmaric/bridge_formalization. We expressed the Sonic bridge implementation, originally in Solidity, as a purely functional program in Isabelle/HOL. Translation from Solidity to Isabelle/HOL is done manually, trying to be as close as possible to Solidity's semantics. For simplicity, the basic types of Solidity (`uint256`, `bytes32`, `address`) are all modeled as natural numbers in Isabelle/HOL, ignoring overflow issues. In Solidity, mappings storing natural numbers default to 0 for non-existent keys. The function `lookup_nat` (and similarly `lookup_bool`) that we define and use mimics that behavior. An Isabelle/HOL record models the internal state of each contract.

```
record ERC20State =
  balances :: "(address, uint256) mapping"
  ...
record SCDepositState =
  locks :: "(uint256, bytes32) mapping"
```

```

withdrawals :: "(address, bool) mapping"
SCOracleAddr :: address
deadState :: bytes32
...

```

The `balances` mapping in the `ERC20State` assigns token amounts to user addresses. In the `SCDepositState`, the `locks` mapping records all successful *Lock* transactions, mapping transaction IDs to hash values derived from the sender address, token, and amount. The `withdrawals` mapping tracks users who executed *Withdraw* transactions to retrieve balances from the dead bridge. The address of `SCoracle` is stored in `SCOracleAddr`, while `deadState` holds the last valid state root before the bridge failure (0 indicates the bridge is not declared dead). States of other contracts (e.g., `SCClaimState`) are similarly defined.

The storage states of all contracts define the blockchain state, mapping contract addresses to their respective states. Due to the Isabelle/HOL type system, each smart contract type requires a separate mapping.

```

record Contracts =
  IERC20 :: "(address, ERC20State) mapping"
  ISCDeposit :: "(address, SCDepositState) mapping"
  ISCClaim :: "(address, SCClaimState) mapping"
  ...

```

We define functions to read and modify smart contract states. For example, the following helper function reads a user's balance from the specified ERC20 contract state.

```

definition balanceOf :: "ERC20State  $\Rightarrow$  address  $\Rightarrow$  uint256" where
  "balanceOf state account = lookup_nat (balances state) account"

```

As in Solidity, contract functions can be called on a chain via a contract address. For instance,

```

definition callBalanceOf :: "Contracts  $\Rightarrow$  address  $\Rightarrow$  address  $\Rightarrow$  Status  $\times$  uint256" where
  "callBalanceOf C address account =
    (case ERC20state C address of
      None  $\Rightarrow$  (Fail "wrong address", 0)
    | Some state  $\Rightarrow$  (Success, balanceOf state account))"

```

Each call may fail, e.g., if no valid contract exists at the given address. In such cases, the chain state remains unchanged. We defined an Isabelle/HOL counterpart for each contract function by following our Solidity implementation closely. For example, the following Solidity function in the `SCDeposit` contract checks whether the bridge is dead. The last valid state root hash is stored in the `deadState` field when the bridge fails.

```

function getDeadStatus() public returns (bool) {
  if (deadState != 0) // we already know that the bridge is dead
    return true;
  // if too much time has passed since the last UPDATE, we declare that
  // the bridge is dead and remember its last known state root hash
  uint256 lastUpdateTime = IStateOracle(stateOracle).lastUpdateTime();
  if (lastUpdateTime != 0 &&
    lastUpdateTime < block.timestamp - TIME_UNTIL_DEAD) {
    deadState = IStateOracle(stateOracle).lastState();
    return true;
  }
  return false;
}

```

8:6 Formal Verification of a Fail-Safe Cross-Chain Bridge

This function is implemented in Isabelle/HOL as follows:

```

definition getDeadStatus where
"getDeadStatus  $\mathcal{C}$  state block =
  (if deadState state  $\neq$  0 then (Success, True, state)
   else let (status, lastUpdateTime) = callLastUpdateTime  $\mathcal{C}$  (SCOracleAddr state) in
    if status  $\neq$  Success then (status, False, state)
    else if lastUpdateTime  $\neq$  0  $\wedge$ 
      lastUpdateTime < (timestamp block) - TIME_UNTIL_DEAD then
      let (status, lastState) = callLastState  $\mathcal{C}$  (SCOracleAddr state) in
      if status  $\neq$  Success then (status, False, state)
      else (Success, True, state ( $\mid$  deadState := lastState  $\mid$ ))
    else (Success, False, state) )"

```

The function returns a triple: the first element indicates success or failure, the second whether the bridge is dead, and the third the updated state. Monad syntax could remove explicit state passing and status checks.

In our Solidity implementation, some contract functions verify Merkle tree proofs to confirm the presence of specific memory content in another chain, ensuring a key-value pair exists in a mapping. Our specification follows a stepwise refinement approach, omitting proof details and instead postulating axioms that these proofs must satisfy. These axioms are expressed as Isabelle/HOL [4]. First, we postulate that the following functions can be defined so that they satisfy the two given axioms:

- generateStateRoot creates a state root hash value,
- generateLockProof generates a proof for the fact that lock[ID] = val,
- verifyLockProof verifies if the given proof is valid for the given state root.

```

locale ProofVerifier =
  fixes SCDepositAddress :: "address"
  fixes generateStateRoot :: "Contracts  $\Rightarrow$  bytes32"
  fixes generateLockProof :: "Contracts  $\Rightarrow$  uint256  $\Rightarrow$  bytes"
  fixes verifyLockProof :: "uint256  $\Rightarrow$  bytes32  $\Rightarrow$  bytes32  $\Rightarrow$  bytes  $\Rightarrow$  bool"
  - if a proof for ID and val verifies, then locks[ID] = val
  assumes verifyLockProofE: " $\bigwedge \mathcal{C}$  state ID stateRoot proof val.
    [ $\llbracket$ SCDepositState  $\mathcal{C}$  SCDepositAddress = Some state;
    generateStateRoot  $\mathcal{C}$  = stateRoot;
    verifyLockProof ID val stateRoot proof = True $\rrbracket \implies$ 
    getLock state ID = val]"
  - if locks[ID] = val, then a proof for ID and val can be generated
  assumes verifyLockProofI: " $\bigwedge \mathcal{C}$  ID state stateRoot proof val.
    [ $\llbracket$ SCDepositState  $\mathcal{C}$  SCDepositAddress = Some state;
    generateLockProof  $\mathcal{C}$  ID = proof;
    generateStateRoot  $\mathcal{C}$  = stateRoot;
    getLock state ID = val $\rrbracket \implies$ 
    verifyLockProof ID val stateRoot proof = True]"

```

We also postulate the honesty assumption for the consensus of users who perform the regular state root updates, by assuming that whenever an *Update* operation succeeds the state root hash given to the oracle is indeed the state root hash of the current blockchain state (as generated by the generateStateRoot function).

```

assumes updateSuccess: " $\bigwedge \mathcal{C}$  address block blockNum stateRoot  $\mathcal{C}'$ .
  callUpdate  $\mathcal{C}$  address block blockNum stateRoot = (Success,  $\mathcal{C}'$ )  $\implies$ 
  stateRoot = generateStateRoot  $\mathcal{C}'$ "

```

This is sufficient to define the token mint operation and prove its properties (for other operations, in the same manner we introduce mint proofs, burn proofs, balance proofs etc.). A concrete implementation for these three abstract functions should be provided at later stages. This implementation can be based on Merkle-tree proofs (as used in our Solidity implementation), but other types of proofs, such as zero-knowledge proofs, could also be used. For illustration, we present our specifications for the mint function.

definition mint where

```
"mint C msg state ID token amount proof =
  (if getMint state ID then (Fail "Already claimed", state, C)
   else – verify proof of the deposit on the sender chain
     let hash = hash3 (sender msg) token amount;
     (status, lastState) = callLastState C (SCOracleAddr state) in
     if status ≠ Success then (status, state, C)
     else let status = callVerifyLockProof C ID hash lastState proof in
       if status ≠ Success then (status, state, C)
       else – find the address of the minted token ERC20 contract
         let (status, mintedToken) = callOriginalToMinted C token in
         if status ≠ Success then (status, state, C)
         else if mintedToken = 0 then
           (Fail "No minted token for given token", state, C)
         else – mint the tokens and log the claim
           let state' = setMint state ID True;
           (status, C') = callERC20Mint C mintedToken (sender msg) amount in
           if status ≠ Success then (status, state, C)
           else (Success, state', C'))"
```

A blockchain state is considered reachable from a starting state if a sequence of successful operations (steps, transactions) exists that leads to the final state when executed starting from the initial state. To define this relation, we first introduce a representation of the steps (with comments after each step type indicating the step parameters).

datatype Step =

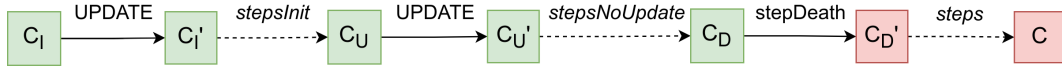
```
LOCK address address uint256 address uint256
  – SCDepositAddress caller ID token amount
| MINT address address uint256 address uint256 bytes
  – SCClaimAddress caller ID token amount proof
...
```

Each step is executed by calling its corresponding function on some chain contract.

primrec executeStep ::

```
"Contracts ⇒ nat ⇒ Block ⇒ Step ⇒ Status × Contracts" where
  "executeStep C blockNum block (LOCK address caller ID token amount) =
    callLock C address block (message caller amount) ID token amount"
| "executeStep C blockNum block (MINT address caller ID token amount proof) =
    callMint C address (message caller amount) ID token amount proof"
...
```

Finally, we inductively define reachability between two blockchain states. Each state is reachable from itself by the empty list of steps. If C' is reachable from C by a list **steps**, and executing **step** reaches C'' from C' , then C'' is reachable from C by the list obtained by joining **steps** and **step**.



■ **Figure 2** General configuration of a dead bridge.

```

inductive reachable :: "Contracts ⇒ Contracts ⇒ Step list ⇒ bool" where
  reachable_base: "∧ C. reachable C C []"
| reachable_step: "∧ C C' blockNum block step.
  [[reachable C C' steps;
   executeStep C' blockNum block step = (Success, C'')]] ⇒
  reachable C C' (step # steps)"

```

Steps issued by a specific caller are interleaved with steps issued by other users. When distinguishing between them is important, we use the predicate `reachableInterlaved caller C C' stepsCaller stepsOther`, which is also defined inductively. The following function, defined by primitive recursion, checks whether the given list of steps can be successfully executed, regardless of the steps taken by other users.

```

primrec executableSteps :: "address ⇒ Contracts ⇒ Step list ⇒ bool" where
  "executableSteps caller C [] = True"
| "executableSteps caller C (stepCaller # stepsCaller) <=>
  executableSteps caller C stepsCaller ∧
  (∀ C' stepsOther. reachableInterlaved caller C C' stepsCaller stepsOther →
   executableStep caller C' stepCaller)"

```

4 Correctness proofs

We have proven numerous properties of our bridge, beginning with simple, low-level technical properties and culminating in high-level properties that express user safety. The proof concludes with the central fail-safety theorem, which states that even when the bridge is dead, every user can execute a series of transactions to retrieve all their assets (i.e., the tokens they locked, plus the tokens received from others, minus the tokens transferred to others). Our fail-safety features ensure this guarantee and the primary challenge was to prove it formally. The main idea of the proof is as follows.

- If the bridge is alive, after the deposit by a **LOCK**, the user can make a **MINT**, then a **BURN**, and finally **RELEASE** to retrieve their tokens on the sender chain.
- Assume that the bridge is dead and that the last valid state known on *SCDeposit* is C (that is the state when the last **UPDATE** happened before the bridge became unresponsive).
 - If only the **LOCK** occurred before C , and the tokens have not been claimed (or if **MINT** occurred after the last state update), the user can retrieve their funds using the **CANCEL_WD** operation.
 - If a **MINT** is recorded in C , and there was no **BURN** after it (or if it occurred after the last state update), the user has minted tokens in C , and can retrieve them using the **WITHDRAW_WD** operation.
 - Finally, if a **BURN** is recorded in C , the user can retrieve their tokens using the standard **RELEASE** operation (just as they would while the bridge is active).

Many theorems share assumptions describing the situation where the bridge is dead. Instead of repeating these assumptions in every theorem and deriving their consequences in each proof, we used Isabelle/HOL locales [4]. We defined the locale `BridgeDead` to describe

the state of a dead bridge. This situation is depicted in Fig. 2. The execution begins in some initial state C_I . For simplicity, we assume that the first operation after the bridge is deployed is a state **UPDATE**, which transitions it to state C'_I . After this, the bridge operates normally, executing a sequence of steps denoted by **stepsInit**. At some state C_U , the final state **UPDATE** occurs, leading to state C'_U . Following that, additional steps (denoted by **stepsNoUpdate**) are executed before the bridge becomes inactive. The step that indicates the bridge is dead is denoted by **stepDeath**. At this point, the field **deadState** in the *SCDeposit* contract is set to the state root of the last known valid state, which in this case is C_U . Subsequently, more steps (denoted by **steps**) are executed on the *SCDeposit* contract, leading to the current state C . The complete sequence of all steps from C_I to C is denoted by **stepsAll**.

Several auxiliary functions are defined to compute token amounts to formulate the central fail-safety theorem.

- **depositedAmountTo** – this function calculates the total amount of tokens T_{orig} that a caller has deposited into *SCDeposit* by executing *Lock* transactions.
- **retrievedAmountFrom** – this function calculates the total amount of tokens T_{orig} that a caller has retrieved from the contract *SCDeposit* by executing *Release*, *Withdraw*, and *Cancel* transactions.
- **transferredAmountFrom** – this function calculates the total amount of minted tokens T_{mint} on the bridge *SCclaim* that the caller has transferred to other users, while the function **transferredAmountTo** calculates the total amount of T_{mint} that other users have transferred to the caller.

When the bridge dies, *SCOracle* contains a state root that encodes aggregated information about all transactions that occurred before the last update before the bridge's failure (i.e., transactions in the list **stepsInit** in Figure 2). After this update, transactions (including mints, burns, and transfers) are ignored. We consider a user to have retrieved all their assets if the total amount of tokens they invested (defined as the sum of T_{orig} they locked up to the current state and the T_{mint} they transferred to other users before the last update) is equal to the total amount of tokens they gained (defined as the sum of T_{orig} they retrieved via a *Release*, *Withdraw*, or *Cancel* transaction and the T_{mint} they received from other users before the last update).

definition `allTokensRetrieved where`

```
"allTokensRetrieved SCDepositAddr SCclaimAddr token caller stepsAll stepsInit ≡
  retrievedAmountFrom SCDepositAddr token caller stepsAll +
  transferredAmountFrom SCclaimAddr token caller stepsInit =
  depositedAmountTo SCDepositAddr token caller stepsAll +
  transferredAmountTo SCclaimAddr token caller stepsInit"
```

Our central theorem demonstrates that, from any state C in which the bridge is dead, each user can issue a list of transactions that will be executable regardless of the transactions made by other users. Moreover, executing these transactions will always result in a state where the user has retrieved all of their tokens.

theorem `paybackPossibleBridgeDead:`

```
shows "∃ steps.
  (∀ step ∈ set steps. isCaller caller step) ∧
  executableSteps caller C steps ∧
  (∀ C' stepsOther. reachableInterleaved caller C C' steps stepsOther →
    allTokensRetrieved token caller
      ((interleaveSteps steps stepsOther) @ stepsAll) stepsInit)"
```

The steps in the list `stepsAll` lead from \mathcal{C}_{init} to \mathcal{C} . The steps `interleaveSteps stepsOther`, obtained by interleaving the unpredictable steps of other users `stepsOther` with the steps `steps` issued by the current user, lead from \mathcal{C} to \mathcal{C}' , where all tokens have been successfully retrieved.

Note that a similar, less interesting theorem can be proven when the bridge is not dead (under some additional assumptions on the frequency of updates). In this scenario, the caller only needs to issue MINT, BURN, and RELEASE transactions. However, there is no guarantee that the bridge will not fail during the execution of these transactions. If this occurs, the user must switch to the recovery strategy used when the bridge is dead.

Many necessary lemmas contribute to the proof of the central theorem. We formulated and proved lemmas that ensure the executability of each specific operation. For instance, the following theorem guarantees lock cancellation is possible if no claim of minted tokens was made before the last update before the bridge became inactive and if no cancellation occurred before the current state.

lemma cancelPossible:

```

assumes "LOCK SCDepositAddr caller ID token amount  $\in$  set stepsAll"
assumes " $\neg$  isMintedID SCClaimAddr token ID stepsInit"
assumes " $\neg$  isCanceledID SCDepositAddr token ID stepsAll"
assumes "proof = generateClaimProof  $\mathcal{C}_U$  ID"
shows "let step = CANCEL_WD SCDepositAddr caller ID token amount proof
      in fst (executeStep  $\mathcal{C}$  block blockNum step) = Success"
```

Theorems for other “rescue” operations are similar. It is typically necessary to prove that a particular step is possible by showing that some data is set in the required manner. For example, canceling a deposit ID by a caller who requests a specific token amount is only possible if the proof that guarantees that `locks[ID] = hash(caller, token, amount)` can be verified. It is proved (by induction) that this data guarantees that there was a prior LOCK operation from the `caller` for the given `amount` of the give `token` T_{orig} (assuming that hash function is injective, which is a widespread assumption in a blockchain setting). However, repaying tokens for a canceled deposit also requires enough tokens T_{orig} in the `SCDeposit` contract balance. This is the most challenging part of the entire proof, as it requires careful analysis of the “cash flow” across all parts of the bridge, considering all operations and steps. Such invariants are proven both globally (characterizing the `SCDeposit` contract balance after transactions from all users) and on a per-user basis (characterizing T_{orig} and T_{mint} balances for each user). Due to space constraints, we will describe only the former. We introduce the following quantities (all are natural numbers, so they are non-negative):

- `SCDepositBalance` – the current token balance of the given $SC_{Deposit}$ address.
- `locked / canceled / withdrawn / released` – the total sum of amounts in all LOCK / CANCEL_WD / WITHDRAW_WD / RELEASE steps for a given $SC_{deposit}/SC_{claim}$ address and token.
- `mintedBeforeDeath` – the total sum of amounts in all MINT steps executed before the last update before the bridge becomes inactive.
- `nonMintedBeforeDeath` – the total sum of amounts in all LOCK steps whose ID is not claimed (minted) before the last update before the bridge becomes inactive.
- `nonCanceledNonMintedBeforeDeath` – the total sum of amounts in all LOCK steps whose ID is not minted before the last update before the bridge becomes inactive and which has not been canceled.
- `nonWithdrawnNonBurnedMintedBeforeDeath` – the total amount of tokens produced by MINT steps that have not been burned before the last update before the bridge becomes inactive, and have not been withdrawn.

- `burnedBeforeDeath` – the total sum of amounts in `BURN` steps for a given address and token that were executed before the last update, when the bridge becomes inactive.
- `nonReleasedBurnedBeforeDeath` – the total sum of amounts in all `BURN` steps for a given address and token, that were executed before the last update before the bridge becomes inactive whose ID has not yet been released.

Then, we proved the following five invariants.

1. `locked = SCDepositBalance + canceled + withdrawn + released`
2. `locked = mintedBeforeDeath + nonMintedBeforeDeath`
3. `nonMintedBeforeDeath = canceled + nonCanceledNonMintedBeforeDeath`
4. `mintedBeforeDeath = withdrawn + nonWithdrawnNonBurnedMintedBeforeDeath + burnedBeforeDeath`
5. `burnedBeforeDeath = released + nonReleasedBurnedBeforeDeath`

The first invariant is proved by induction by analyzing the transactions that change the token deposit balance. The second invariant is trivial. The others focus on specific operations and are proven by induction and case analysis of the last applied step, where most steps are irrelevant (for example, only `MINT` and `CANCEL_WD` operations are relevant for the third invariant). Combining these four invariants yields

$$\text{SCDepositBalance} = \text{nonCanceledNonMintedBeforeDeath} + \text{nonWithdrawnNonBurnedMintedBeforeDeath} + \text{nonReleasedBurnedBeforeDeath}$$

From this, we can guarantee sufficient funds for each rescue operation. For instance, when proving that a cancel step is possible, there is a deposit that was not claimed before the last update before the bridge died and that has not yet been canceled. Therefore, the its amount is included in the `nonCanceledNonMintedBeforeDeath` quantity, which, by the previous invariant, is less than or equal to the `SCDepositBalance`, i.e., the current token balance of the `SCDeposit` contract. Since the amount the caller requires must match the amount they deposited (which we know from the value of `locks[ID] = hash(caller, token, amount)`), and the assumed injectivity of the `hash` function), the contract contains the required tokens, making the payback possible. Similar reasoning applies to all other operations.

Many other lower-level properties had to be proven to support the correctness proof. For example, each *Mint* must be preceded by a *Lock*, only the entity who made a lock can mint the tokens, once the bridge dies, it can never become life again, etc. We refer readers to the Isabelle/HOL proof documents for their formalization.

5 Evaluation and Discussion

The Isabelle formalization was completed by a single individual who had worked part-time on this project for over six months. A rough estimate suggests the formalization required approximately 2–3 full person-months of effort. Isabelle’s definition of the bridge’s formal model comprises around 900 lines of code (LOC), including approximately 50 defined functions, while the accompanying proofs span roughly 15,000 LOC. The current proof contains approximately 750 lemmas and theorems. The proof-checking process takes around 8 minutes on a standard laptop computer (Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz, 8GB of RAM). However, this time could be significantly reduced by replacing several time-consuming, fully automated proofs with more detailed manual proofs.

Several underlying assumptions in our model pose threats to validity and must be addressed to ensure the total correctness of a real-world bridge implementation. Some of these assumptions can be easily ensured through a correct deployment process.

8:12 Formal Verification of a Fail-Safe Cross-Chain Bridge

- It is assumed that contracts are correctly deployed and initialized, i.e., that all addresses are correctly set up.
- It is assumed that each bridge and original token has a dedicated minted token, i.e., that minted tokens are not shared.
- It is assumed that users initially have no minted tokens after the bridge is deployed.
- It is assumed that an `UPDATE` operation occurs immediately after the bridge is deployed, i.e., that the bridge never operates in a state where state roots are uninitialized.

Other assumptions include the following:

- It is assumed that arithmetic overflows will not occur while the bridge is in operation. This is reasonable, as the only arithmetic operations involve updating user token balances.
- It is assumed that the hash function is injective (i.e., no hash collisions) and always produces a nonzero value. This is a common assumption underlying blockchain security.
- It is assumed that (Merkle-tree) proof checking is fully sound. This is also reasonable, as the correctness of Merkle trees has been formally verified.
- Transaction fees and gas prices are not included in our model.
- Reentrancy attacks are not considered. Although reentrancy poses a significant threat in Solidity smart contracts, it can be prevented through careful implementation (e.g., by following the checks-effects-interactions pattern).
- It is assumed that updates are reliable, i.e., that validators are always honest. This assumption may be the most problematic, as many successful bridge attacks have been carried out by corrupting validators [34]. While specifying the details of a validation protocol can help relax this assumption, some risks will always remain. For example, private keys must be kept secure.

6 Related Work

BFT consensus protocols have been verified using interactive theorem provers. The work in [6] develops a modular proof in TLA+ for DAG-based consensus protocols. The work in [26] uses the IVy interactive theorem prover [1] to formally verify a variant of the Moonshine consensus protocol [26]. The work in [21] uses IVy and Isabelle/HOL [24] to verify the Stellar Consensus Protocol [20]. The Algorand [13] consensus protocol has been verified using Coq [12]. The safety of several non-Byzantine protocols, such as variants of Paxos [10, 28] has been verified using interactive theorem proving. To the best of our knowledge, we are the first to propose formal proof for a cross-chain bridge, particularly bridge failure safety.

There has been foundational work in the verification of Ethereum smart contracts. These contracts are compiled into Ethereum Virtual Machine (EVM) bytecode, formally defined for use with interactive theorem provers [14]. A sound program logic at the bytecode level has also been developed [3]. Typically, smart contracts are written in the Solidity programming language, with formal semantics defined in Isabelle/HOL [22]. Another proposed approach in smart contract verification is to define them in specialized languages converted to low-level byte code using verified compilers. For example, Britten's PhD thesis [8] explores techniques to improve the reliability and security of smart contracts by leveraging formal verification methods through interactive theorem proving in Coq, combined with a verified compiler from the language DeepSEA [30]. In [27], Ribeiro defines the imperative language SOLI within Isabelle/HOL that captures a significant subset of Solidity, and develops big-step, Hoare logic, and a proof system for it, with soundness and completeness results formally established.

Our approach differs in that the formal bridge model is currently represented as a functional program within Isabelle/HOL, derived through manual translation from Solidity. Consequently, our current work establishes only the correctness of a high-level bridge model, leaving the verification of its Solidity implementation for future work.

Various automated tools analyze Solidity code and detect bugs or violations of specific safety properties (for example, since 2019, the Solidity compiler has included a model checker, SolCMC [2]). However, a key limitation of these tools is their lack of effectiveness in expressing and verifying liquidity properties. Some automated tools are also designed explicitly for verifying liquidity properties, such as Solvent [5].

Our decision to use the interactive theorem prover Isabelle/HOL instead of automated tools was influenced by our prior experience and the fact that any limitations in automated prover support can be overcome by reverting to manual proofs. One possible research direction would be to examine whether state-of-the-art automated tools are capable of proving the liquidity properties of large-scale projects such as a crypto-bridge.

7 Conclusion

This paper describes the SONIC GATEWAY fail-safe bridge design and its formalization in Isabelle/HOL. We have formally proven that users can still retrieve all their assets on the sender chain even if the bridge becomes unresponsive on the receiver chain.

Our current abstract model does not account for numerous essential real-world issues, such as gas consumption, potential overflows, reentrancy vulnerabilities, and other Solidity-specific implementation challenges. Addressing these practical concerns will be an integral part of our efforts to ensure that the model reflects these complexities and security requirements in a real-world deployment. Therefore, we plan to bridge the gap between the current abstract Isabelle/HOL bridge model and its real Solidity implementation in our future work. To achieve this, we must provide a detailed description of the Inter-Blockchain Communication (IBC) Relay, which utilizes Merkle-tree proofs and whose correctness is currently only assumed in our formalization. Additionally, we plan to leverage the formal semantics of Solidity in Isabelle/HOL [22] to verify our bridge up to the Solidity implementation level fully formally.

References

- 1 Ki Yung Ahn and Ewen Denney. Testing first-order logic axioms in program verification. In Gordon Fraser and Angelo Gargantini, editors, *Tests and Proofs - 4th International Conference, TAP@TOOLS 2010, Málaga, Spain, July 1-2, 2010. Proceedings*, volume 6143 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2010. doi:10.1007/978-3-642-13977-2_4.
- 2 Leonardo Alt, Martin Blich, Antti E. J. Hyvärinen, and Natasha Sharygina. Solcmc: Solidity compiler’s model checker. In *Computer Aided Verification: 34th International Conference, CAV 2022, Haifa, Israel, August 7–10, 2022, Proceedings, Part I*, pages 325–338, Berlin, Heidelberg, 2022. Springer-Verlag. doi:10.1007/978-3-031-13185-1_16.
- 3 Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards verifying ethereum smart contract bytecode in isabelle/hol. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018*, pages 66–77, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3167084.
- 4 Clemens Ballarín. Interpretation of Locales in Isabelle: Theories and Proof Contexts. In *Mathematical Knowledge Management, MKM, Proceedings*, pages 31–43, 2006. doi:10.1007/11812289_4.

- 5 Massimo Bartoletti, Angelo Ferrando, Enrico Lipparini, and Vadim Malvone. Solvent: Liquidity verification of smart contracts. In Nikolai Kosmatov and Laura Kovács, editors, *Integrated Formal Methods*, pages 256–266, Cham, 2025. Springer Nature Switzerland.
- 6 Nathalie Bertrand, Pranav Ghorpade, Sasha Rubin, Bernhard Scholz, and Pavle Subotic. Reusable formal verification of dag-based consensus protocols. *CoRR*, abs/2407.02167, 2024. doi:10.48550/arXiv.2407.02167.
- 7 Bloomberg. Multi-chain future likely as ethereum’s defi dominance declines. <https://www.bloomberg.com/professional/insights/data/multi-chain-future-likely-as-ethereums-defi-dominance-declines/>, February 2022.
- 8 Daniel Britten. *Building trustworthy smart contracts using interactive theorem proving*. Doctor of philosophy (phd) thesis, The University of Waikato, 2024. Supervisors: Steve Reeves, Vimal Kumar. URL: <https://hdl.handle.net/10289/16566>.
- 9 Dominique Cansell. Foundations of the B method. *Computers and Informatics*, 22, January 2003.
- 10 Saksham Chand, Yanhong A. Liu, and Scott D. Stoller. Formal verification of multi-paxos for distributed consensus. In John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, volume 9995 of *Lecture Notes in Computer Science*, pages 119–136, 2016. doi:10.1007/978-3-319-48989-6_8.
- 11 CoinGecko. Top smart contract platform coins by market cap. <https://www.coingecko.com/en/categories/smart-contract-platform>, November 2024.
- 12 Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988. doi:10.1016/0890-5401(88)90005-3.
- 13 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Proceedings of SOSP 2017*, pages 51–68. ACM, 2017. doi:10.1145/3132747.3132757.
- 14 Yoichi Hirai. Defining the ethereum virtual machine for interactive theorem provers. In Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson, editors, *Financial Cryptography and Data Security*, pages 520–535, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-70278-0_33.
- 15 Kamil Jezek. Ethereum data structures. *CoRR*, abs/2108.05513, 2021. doi:10.48550/arXiv.2108.05513.
- 16 Defi Lama. Hacks. <https://defillama.com/hacks>, November 2024.
- 17 Sung-Shine Lee, Alexandr Murashkin, Martin Derka, and Jan Gorzny. Sok: Not quite water under the bridge: Review of cross-chain bridge hacks. *CoRR*, abs/2210.16209, 2022. doi:10.48550/arXiv.2210.16209.
- 18 Sung-Shine Lee, Alexandr Murashkin, Martin Derka, and Jan Gorzny. Sok: Not quite water under the bridge: Review of cross-chain bridge hacks. In *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2023. doi:10.1109/ICBC56567.2023.10174993.
- 19 Ningran Li, Minfeng Qi, Zhiyu Xu, Xiaogang Zhu, Wei Zhou, Sheng Wen, and Yang Xiang. Blockchain cross-chain bridge security: Challenges, solutions, and future outlook. *Distrib. Ledger Technol.*, October 2024. Just Accepted. doi:10.1145/3696429.
- 20 Marta Lohkava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafal Malinowsky, and Jed McCaleb. Fast and secure global payments with stellar. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 80–96. ACM, 2019. doi:10.1145/3341301.3359636.
- 21 Giuliano Losa and Mike Dodds. On the formal verification of the stellar consensus protocol. In Bruno Bernardo and Diego Marmosoler, editors, *2nd Workshop on Formal Methods for Blockchains, FMBC@CAV 2020, July 20-21, 2020, Los Angeles, California, USA (Virtual*

- Conference), volume 84 of *OASICS*, pages 9:1–9:9. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/OASICS.FMBC.2020.9.
- 22 Diego Marmosoler and Achim D. Brucker. Isabelle/solidity: A deep embedding of solidity in isabelle/hol. *Form. Asp. Comput.*, October 2024. Just Accepted. doi:10.1145/3700601.
 - 23 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, May 2009. URL: <http://www.bitcoin.org/bitcoin.pdf>.
 - 24 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. doi:10.1007/3-540-45949-9.
 - 25 Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg, 2002.
 - 26 M. Praveen, Raghavendra Ramesh, and Isaac Doidge. Formally verifying the safety of pipelined moonshot consensus protocol. In Bruno Bernardo and Diego Marmosoler, editors, *5th International Workshop on Formal Methods for Blockchains, FMBC 2024, April 7, 2024, Luxembourg City, Luxembourg*, volume 118 of *OASICS*, pages 3:1–3:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/OASICS.FMBC.2024.3.
 - 27 Maria Saraiva de Campos Mendes Ribeiro. Formal Verification of Ethereum Smart Contracts Using Isabelle/HOL. Master’s thesis, Instituto Superior Técnico, 2019. Supervisors: Paulo Alexandre Carreira Mateus, Pedro Miguel dos Santos Alves Madeira Adão. URL: https://fenix.tecnico.ulisboa.pt/downloadFile/1689244997260666/tese_maria_ribeiro.pdf.
 - 28 William Schultz, Ian Dardik, and Stavros Tripakis. Formal verification of a distributed dynamic reconfiguration protocol. In Andrei Popescu and Steve Zdancewic, editors, *CPP ’22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, pages 143–152. ACM, 2022. doi:10.1145/3497775.3503688.
 - 29 Rudrapatna K. Shyamasundar. Erc20: Correctness via linearizability and interference freedom of the underlying smart contract. In *Proceedings of the 20th International Conference on Security and Cryptography - Volume 1: SECRYPT*, pages 557–566. INSTICC, SciTePress, 2023. doi:10.5220/0012145800003555.
 - 30 Vilhelm Sjöberg, Kinnari Dave, Daniel Britten, Maria A Schett, Xinyuan Sun, Qinshi Wang, Sean Noble Anderson, Steve Reeves, and Zhong Shao. Foundational verification of smart contracts through verified compilation, 2024. doi:10.48550/arXiv.2405.08348.
 - 31 Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, April 1971. doi:10.1145/362575.362577.
 - 32 Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014. EIP-150 Revision. URL: <https://ethereum.github.io/yellowpaper/paper.pdf>.
 - 33 Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. zkbridge: Trustless cross-chain bridges made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS ’22*, pages 3003–3017, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3548606.3560652.
 - 34 Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. Security of cross-chain bridges: Attack surfaces, defenses, and open problems. In *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses, RAID ’24*, pages 298–316, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3678890.3678894.

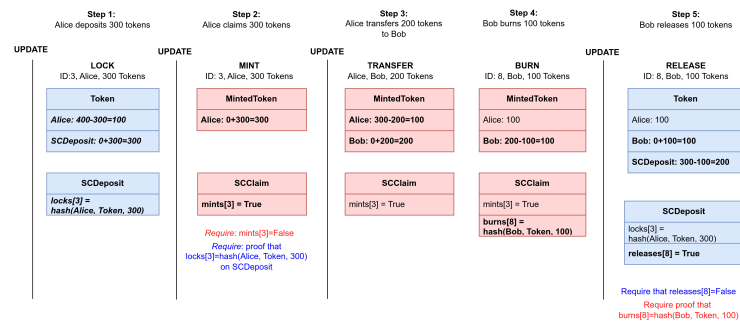


Figure 3 Example of lock/mint, burn/release operations. Sender blockchain contracts are depicted in blue, and receiver contracts in red.

A Detailed Bridge Example

An example showing basic bridge operations is shown in Figure 3. Bridge operations are supported by two smart contracts on the sender chain (our custom contract *SCDeposit* and an ERC20 contract *Token*), and two smart contracts on the receiver chain (our custom contract *SCClaim* and an ERC20 contract *MintedToken*).

We assume that Alice has 400 tokens on the sender chain and she wants to transfer 300 tokens to the receiver chain. She invokes a **LOCK** operation on the *SCDeposit* contract on the sender chain, that transfers 300 tokens from her balance to the *SCDeposit* contract balance. Information about this transaction is logged in the *SCDeposit* contract, in a special **locks** mapping, by assigning a hash value that combines information about the caller (Alice, i.e., her unique address), the currency (Token, i.e., its unique address), and the amount (300) to the unique transaction ID (it is 3, in this example). This information is later used to confirm that this transaction occurred.

In the next step, Alice invokes a **MINT** transaction on the *SCClaim* contract on the receiver chain. If that transaction succeeds, 300 tokens will be added to her balance on the *MintedToken* contract („minted out of thin air”). However, for this to be possible, (1) these 300 tokens should be justified by the 300 tokens deposited on the sender chain, and (2) it should be ensured that she can do this only once. For this, it is assumed that she sends the ID of the original **LOCK** transaction (the value 3) as a parameter of this claim. Requirement (2) is easily ensured by maintaining a special **mints** mapping in the *SCClaim* contract. Before the claim, the value **mints**[ID] must be False, and is set to True once the minting is done. Requirement (1) is harder to achieve, because it requires information about the data stored in the sender chain. From the value assigned to the key ID in the **locks** log of the *SCDeposit* contract, we can check that Alice was the one who deposited 300 tokens (assuming that there are no hash collisions, which is a rather common assumption). To be able to do this, we rely on Merkle-tree proofs and consensus on the root hash of the sender chain. During regular updates, a consensus of trusted verifiers agrees on the root hash value and writes it to a special place (for this, a special *StateOracle* contract is used). When Alice wants to claim the minted tokens, she provides a Merkle tree proof for the value assigned to **locks**[ID]. The *Bridge* contract then verifies if this value is equal to the hash value that combines her address, the token address, and the deposited amount, and if it is, the transaction succeeds.

Next, we assume that Alice transfers 200 minted to Bob. It is a regular ERC20 transfer operation, and no additional checks and loggings are performed.

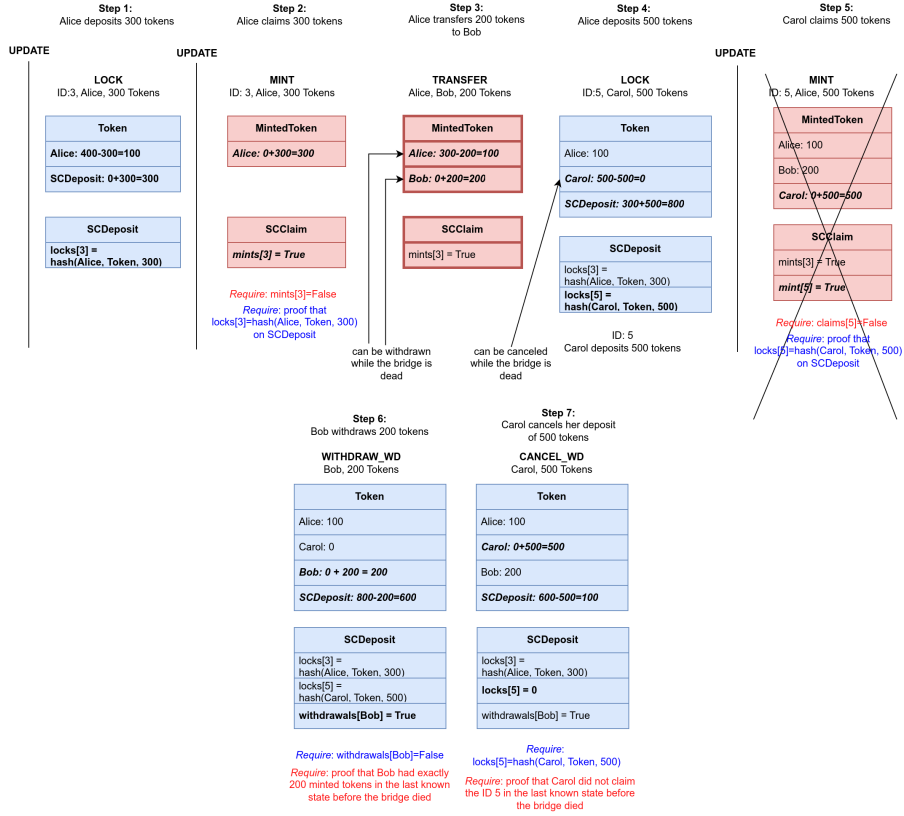


Figure 4 Example of the fail-safe mechanism used if the bridge dies (becomes unresponsive).

Next, we assume that Bob wants to transfer 100 of these minted tokens back to the sender chain. For this, he must first burn the tokens on the receiver chain. He invokes a **BURN** operation which reduces his balance (minted tokens are “burned and destroyed”), and then logs this information by assigning the hash that combines the information about the caller (Bob), the currency (Token) and the amount (100) to the unique transaction ID (it is 8 in this example) in a special **burns** mapping in the *SCClaim* contract.

Finally, Bob releases 100 tokens on the sender chain, by transferring them from the *SCDeposit* balance in the *Token* contract to his balance. Again, it must be ensured that (1) these are covered by the burned 100 tokens on the bridge, and that (2) this release is done exactly once. Requirement (2) is easily ensured by using a special **releases** mapping in the *SCDeposit* contract (the value **True** is assigned to the unique transaction ID once the release is done). To ensure (1), we again use Merkle-tree proofs and the state root set by the consensus of verifiers in the **UPDATE** operation (this time sender chain must be informed about the state root of the receiver chain at certain block height). Bob must supply a Merkle-proof that the value **burns**[ID] is equal to the hash value of the caller (Bob), the currency (Token) and the amount 100. If the Merkle-tree proof verification succeeds, we know that Bob really burned his 100 tokens on the receiver chain, so we can transfer him the required 100 tokens.

Our bridge has a fail-safety mechanism that enables users to retrieve their funds on the sender chain, even when the contracts on the receiver chain become unresponsive. An example of this feature in operation is shown in Fig. 4.

We assume that the first three steps are the same as in the previous example. Next, we assume that Carol makes a deposit and locks 500 tokens on the sender chain, and that after that she makes a successful claim and mints those 500 tokens on the receiver chain. For

this to succeed, there must have been an `UPDATE` step in between in which the *StateOracle* contract has been informed about the state roots of the chains. Just after the `MINT` step, the bridge dies and becomes unresponsive. If someone tries to release the funds on the sender chain, he would be able to do this only if the burned them prior to that last `UPDATE` (since, the effect of those `BURN` operations is encoded in the current state root). However, users that have funds on the failed receiver bridge that have not been burned before that update, cannot retrieve them. The operations that happened after that last `UPDATE` step are not visible from the sender chain, so it is assumed that the state of the receiver chain in the moment of that last `UPDATE` step is its *last valid state*, also called the *dead state* (it is the state after the step 3 in Fig. 4). After some predetermined time period in which there are no new updates, the sender chain realizes that the bridge is dead and switches to fail-safe mode. This is irreversible, since, even if the bridge becomes operational again, the contract on the sender chain will not take that into account. In the fail-safe mode, all users who had minted tokens in the last valid state can withdraw them as original tokens on the sender chain.

We assume that Bob wants to withdraw his 200 tokens that he got from Alice, so he invokes a special withdraw while dead (`WITHDRAW_WD`) operation. He can do this only once, and for this, he must provide a Merkle-tree proof that he had exactly 200 minted tokens in the last valid state. If this proof successfully verifies, he is given his 200 tokens from the *TokenDeposit* contract balance, and this is logged in a special `withdrawals` mapping (the value `withdrawals[Bob]` is set to `True`, guaranteeing that Bob has withdrawn all his funds and will not be able to do this again).


However, there is a problem with Carol. She has made a successful claim while the bridge was alive, but the bridge died before another `UPDATE` step, so from the sender chain, it is not visible that she has some minted tokens, and she cannot invoke `WITHDRAW_WD` operation to retrieve her funds. However, since the last valid state was prior to her claim, it is also not visible from the sender chain. Therefore, she has the same status as the users that have made a deposit, but did not claim them before the bridge died. They do not have minted tokens, and cannot withdraw them by proving their balance in the last valid state. Therefore another special operation is introduced.

We assume that Carol invokes the special cancel deposit while dead (`CANCEL_WD`) operation. She can do this only once and she must provide the unique ID of the original transaction (in this example, `ID=5`) along with the proof that she has not claimed the funds before the last valid state (i.e., that `mints[ID]=False` in the last valid state). The hash value assigned to `locks[5]` in the *SCDeposit* contract verifies that Carol was the one that really deposited 500 tokens. If these two verifications succeed, the deposit is canceled, she is given back 500 tokens, and it is recorded that the deposit with the `ID=5` has been canceled (by setting `locks[5]=0`, and assuming that 0 cannot be a valid hash value), so Carol cannot repeat this operation more than once.

Verifying Smart Contract Transformations Using Bisimulations

Kegan McIlwaine ✉ 

University of Wyoming, Laramie, WY, USA

James Caldwell ✉ 

University of Wyoming, Laramie, WY, USA

Abstract

Determining whether two computational artifacts share the same behavior is fundamental. In general, smart contracts and their interactions can be modeled as the concurrent composition of processes where: (i.) contracts are processes, (ii.) parties to the contract are processes, and (iii.) the blockchain itself is a process. In this paper we describe how we apply this view of smart contracts to the verification of an optimizing transformation in the Faustus smart contract programming language. Faustus compiles to the embedded domain specific language (eDSL) Marlowe. With Marlowe as the target compilation language, the operators and semantics of Milner's value passing Calculus of Communicating Systems (CCS) inspired the design of Faustus. In CCS, unobservable transitions (τ -transitions) arise from the parallel composition of processes that share a label and a co-label (e.g. a and \bar{a}). CCS also supports a *restriction* operator ($P \setminus \mathcal{A}$) which internalizes, within P , the labels in the set \mathcal{A} . From an observer's point of view, any number of τ -transitions followed by an observable transition, say a , looks like a single transition on a . In Faustus, similarly, unobservable actions arise by adding actions to be internalized to a set \mathcal{A} . A proof that two Faustus contracts are *weakly bisimilar* ($P \approx Q$) verifies that, with respect to their observable executions, they exhibit identical behaviors. This paper describes an application of observational equivalence, witnessed by the existence of a weak bisimulation relation, to verify that a smart contract and its transformed instance preserves observable behavior. More precisely, if $\lceil P \rceil$ is the result of applying our transformation to a contract P , we prove $\forall P. P \approx \lceil P \rceil$. The smart contract transformation verified here trades time for space in smart contracts running on the Cardano blockchain. The results of this paper have been formalized in the Isabelle theorem prover, and we have formalized the small-step semantics of Faustus contracts together with the labeled transition system induced by those semantics.

2012 ACM Subject Classification Software and its engineering \rightarrow Formal methods

Keywords and phrases Smart Contracts, Bisimulation, Program Transformation

Digital Object Identifier 10.4230/OASICS.FMBC.2025.9

Supplementary Material *Software (Source Code)*: <https://gitlab.com/UWyo-SSC/public/wabl/faustus-v2> [3], archived at `swb:1:dir:ba99bcd12d63cd2ece88fd95325e5032fcdd5637`

Funding The research presented in this paper was supported by a grant from IOG Singapore Pte. Ltd.

Acknowledgements The research presented in this paper was supported by a grant from IOG Singapore Pte. Ltd., and thanks especially to Charles Hoskinson.

1 Introduction

Smart contracts, initially proposed by Szabo [43], are self-enforcing, self-executing protocols governing interactions between several (potentially distrusting) parties. Smart contracts automate the execution of an agreement between participants. Trust is established between the participants by the use of a blockchain, which is an unforgeable distributed ledger where transactions are added in a cryptographically secure manner.



© Kegan McIlwaine and James Caldwell;

licensed under Creative Commons License CC-BY 4.0

6th International Workshop on Formal Methods for Blockchains (FMBC 2025).

Editors: Diego Marmsoler and Meng Xu; Article No. 9; pp. 9:1–9:19

OpenAccess Series in Informatics



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

Because blockchains are massive distributed systems, the cost of running and maintaining the blockchain must be paid in some way. Typically, this cost is a transaction fee, colloquially referred to as “gas”. The cost of running a smart contracts has led to a culture where programmers apply optimizations to minimize gas costs [17, 37, 4, 14]. These optimizations may introduce errors not in the original contract; there are no guarantees that they preserve the semantics of the original smart contract.

Reports of smart contracts being hacked are legion, and the losses from hacked contracts can be astronomically large [7, 23, 2]. The potential for, and the magnitude of, financial losses is a motivation for applying the most rigorous correctness guarantees to smart contracts, *i.e.* formal verification. The work reported here was funded by Input Output Global (IOG, formerly IOHK), the developers of the Cardano Blockchain¹. The IOG development methodology is based on application of *agile* formal methods for specification and verification of the system [22, 16] and the work reported here follows that methodology.

Faustus, the smart contract language used here, compiles to the Marlowe smart contract language [26, 24] developed at IOG. Prior to our work, the Marlowe developers formalized the evaluation semantics of Marlowe in Isabelle/HOL [38]. That formalization served as the basis for a verification of the correctness of the Faustus (to Marlowe) compiler [29, 30] and for the further developments described in this paper.

1.1 Related Works

Formal verification has previously been used to prove individual smart contracts satisfy some properties. A recent survey of these efforts can be found in Tolmach et al. [44]. These techniques verify the correctness of a smart contract with respect to a formal specification. For example, tools have been developed to verify unannotated common smart contracts against their canonical forms [13], automated tools have been developed to verify liquidity in BitML [11] and Solidity contracts [8, 12, 9], and verified super-optimization techniques have been applied to Ethereum smart contracts [4].

Nelaturu et al. [37] have developed a methodology for optimizing the cost of smart contracts, and synthesize bisimulation relations to verify the correctness of the optimizations. Their techniques are limited by the constraints on the inputs to the automated tools they use in their system, *e.g.* the programs must be deterministic and the kinds of loops their system can operate on are restricted.

Similar to the work described in this paper, Sergey and Hobor [42] aim to provide an analogy between smart contracts and concurrent processes. Differing from previous works of low-level verification, Sergey and Hobor argue that smart contract programmers and verifiers will work more efficiently and be less error prone by adopting the perspective of smart contracts as concurrent processes.

Building on the perspective of viewing smart contracts as concurrent processes, Qu et al. [41] presented a method for modeling and verifying smart contracts using Communicating Sequential Processes (CSP) [21]. Their technique is used to verify that a smart contract is safe against particular attacks. In their methodology, smart contracts that are written in Solidity are translated into CSP. This step introduces a gap between the actual code and the model, as they do not verify the correctness of the translation.

Additionally, a lot of work has already been done in the field of developing programming languages for specifying and implementing financial contracts. Peyton Jones et al. [40] developed a declarative language for specifying executable financial contracts in Haskell and

¹ <https://iohk.io/>

proved that two syntactically different contracts are equivalent using denotational semantics. Interestingly, this paper is one of the few that focuses on proving the equivalence of contracts. Andersen et al. [6] expanded on that work with their Contract Specification Language (CSL). Then Henglein et al. [19] provided a static analysis framework for verifying properties of CSL contracts, such as fairness and party participation. In a separate piece of work, Seijas and Thompson [26] developed Marlowe, a Domain Specific Language (DSL) for writing financial contracts on the Cardano blockchain. All Marlowe contracts are total - they are all guaranteed to terminate. They used the Isabelle/HOL theorem prover to prove that all Marlowe contracts terminate and pay out any locked funds. Seijas et al. [24, 25] also developed a static analysis tool that can verify all possible executions of a Marlowe contract never result in an error.

Another related field of work is the development of smart contract languages based on process algebra. Rholang, based on the ρ -calculus, was developed as the native smart contract language for the RChain blockchain [18]. Then Bartoletti et al. [11] developed the process calculus BitML as a DSL for writing Bitcoin smart contracts. Rholang executes natively on the RChain blockchain, while BitML is compiled to Bitcoin transactions. Finally, another process calculus, ILLUM, was recently developed by Bartoletti et al. [10] as an intermediate language for compiling higher level languages into transactions on the Bitcoin and Cardano blockchains.

1.2 Our Contribution

This paper describes the Faustus smart contract language and a methodology for verifying equivalence of Faustus contracts based on bisimulation [39, 34] from the perspective of parties interacting with the contract. Our work differs from the previous works in the following ways:

1. We have developed a methodology based on Milner's Calculus of Communicating Systems (CCS) [33, 34, 35] to prove the *observational equivalence* of contracts, *i.e.* to prove that Faustus contracts are semantically equivalent from the perspective of the parties interacting with the contract. Other work that we are aware of is: limited by the types of contracts that can be verified; verifies properties of individual contracts; or develops tools to analyze safety and liveness properties of specified contracts.
2. In Faustus, reasoning using CCS based techniques is done directly on well-formed code of the smart contract. There is no translation from the smart contract code to a separate model for verification. In many methodologies the first step is an unverified translation from a contract/program to a model suitable for formal analysis.
3. In our formal model, Faustus processes include a set of actions that, when executed, are hidden from users. Following Milner [34], we call this collection a restriction set. The behavior of a Faustus process with a nonempty restriction set corresponds to a parallel composition of processes. In the process defined by the parallel composition, actions in restriction set are unobservable.
4. Our methodology for verifying two processes are equivalent is to prove the two are observationally equivalent by exhibiting the appropriate bisimulation between the two. We describe a program (contract) transformation, and apply this methodology to formally prove that an arbitrary contract and the resulting transformed contract are observationally equivalent.
5. The work described here has been formalized in the Isabelle/HOL interactive theorem prover. We provide proofs for theorems and lemmas in this paper where possible, and give a more informal outline if the proof relies on theorems outside of the scope of this work.

1.3 Paper Organization

This paper briefly introduces the Faustus smart contract language, and then shows how to interpret Faustus contracts as CCS-style processes based on the transition relation given by the operational semantics of Faustus. Finally, we apply the methodology to verify a nontrivial transformation on Faustus contracts.

2 Introduction to Faustus

Faustus is based on an extension of Marlowe’s financial contract operators that makes it into a (more) fully featured programming language. The formal interpretation of a Faustus program is based on its operational semantics which have been formalized Isabelle/HOL [3]. Faustus contracts describe allowable interactions between contracts and the parties to the contract. Aside from blockchain addresses, which serve as parties, the details of the interaction between the contract and the blockchain are hidden from the Faustus programmer.

It is quite natural to use the operational semantics to define a standard kind of automata, a labeled transition system, where states of the automata are contract states, and labels are the interactions. Milner’s CCS [34] is defined over labeled transition systems. CCS processes are constructed with guarded commands, a choice operator, and parallel composition. We have implemented these operators in the Faustus language of guarded commands². Because our execution semantics prioritize certain transitions, our target is a variant of CCS with priority [27].

There are six basic things to note about the execution of Faustus contracts:

1. Faustus contracts run as a series of sequential statements that pause when waiting for external input (from an agent or the blockchain) or a specific timeout to occur.
2. Hidden (τ -actions) arise in contracts by the evaluation of: variable declarations; assignment statements; if-statements; **close** commands; **assert** statements; and **pay** statements. (See Section 3.1.)
3. The **when** contract denotes actions that parties are allowed to take within the contract, and a timeout contract in the case that no action is received before the specified time. **when** contracts have the form **when** $\{g_1 \rightarrow c_1, \dots, g_n \rightarrow c_n\}$ **after** $t \rightarrow c_t$. The list of guarded commands is tested in order until a guard evaluates to true in the current state with the current action. If no guard evaluates to true and the time $t < s$, where s is the start of the current time window, then the contract c_t is evaluated.
4. Faustus contracts contain internal accounts for each party. Currencies are moved between internal accounts and paid out to parties’ blockchain wallets according to the contract logic.
5. All Faustus contracts end with a **close** command or a contract variable. Recursion is not allowed to ensure termination. The **close** command ensures no funds are locked in the closed Faustus contract by paying out all internal account balances.
6. Variables can be bound for any syntactical element.

We can illustrate more interesting features of Faustus through the three-party escrow contract in Figure 1. At the beginning, two actions are declared corresponding to **yes** and **no** votes for moving the money along. Then a parameterized contract is declared, called

² Interestingly, we initially realized we could use choice and interleaving to compactly describe common combinatorial patterns used in many Marlowe programs. This led us to consider the more full-bodied implementation of CCS which this paper is based on.

```

action yes(party p) = p chooses "agree";
action no(party p) = p chooses not "agree";
contract escrow(party alice, party bob, party escrow, time start_time) {
  contract payout(int votes) {
    if votes >= 2
      then escrow pays 1000 ada to bob; close
    else escrow pays 1000 ada to alice; close
  };
  when {
    alice deposits 1000 ada into escrow -> {
      int votes = 0;
      when {
        (yes(alice) -> votes := votes + 1 <+> no(alice))
        <|> (yes(bob) -> votes := votes + 1 <+> no(bob))
        <|> (yes(escrow) -> votes := votes + 1 <+> no(escrow)) -> {
          payout(votes)
        }
      }
    } after start_time + (90 days) -> payout(votes)
  }
  } after start_time + (3 days) -> close
}; //   alice           bob           escrow           start_time
escrow(addr1q8zg...48g7, addr1qygn...9pl5, addr1qx2q...xlw7, 2025-05-04 00:00.00)

```

■ **Figure 1** Three party escrow contract in Faustus.

`three_party_escrow`, which takes wallet addresses for the participants and the time the contract will start. Inside `three_party_escrow` another `payout` contract is declared. Finally, a `when` clause allows defining the interactions between the participants and the contract.

First, Alice is expected to deposit 1000 ada into the escrow account. After the deposit, the 1000 ada will reside in an account that is internal to the contract for the escrow party. The contract logic will not allow the funds to be used until the `payout` contract is called.

Each participant is then allowed to take either a `yes` or `no` vote. The choice operator is `<+>`, and $P<+>Q$ only evaluates P or Q , but not both. Note that the `yes` action also increments `votes`. Then, those choices are allowed to be received in any order using the interleaving operator `<|>`³. Note that until the final continuation contract is written, guards may only be followed by variable reassignments, payments, or other guards. Both the `<+>` and `<|>` operators prioritize executing the left operand.

After all of the votes are received, or the contract times out, the `payout` contract is called. The `payout` contract checks if the votes meet a certain threshold to send the money to Bob. Otherwise, it sends the money back to Alice. In each case, the contract is then fully closed.

3 Equivalent Contracts

A central question that arises when thinking about computational systems is, when do two systems have equivalent behavior? Quoting Milner [36], “*Until we know what constitutes similarity or difference of behavior we cannot claim to know what ‘behavior’ means - and if that is the case then we have no precise way of explaining what our systems do.*”

³ In general, if there are k interleaved guarded commands there are $k!$ (factorial) possible execution paths induced by them.

Thinking from first principles to answer what it might mean for a pair of processes, P and Q , to be equivalent⁴, if we are very clever, we might come up with something like:

Processes P and Q are observationally equivalent if for every sequence of actions (say s), applying s to P results in state P' , then there is a state Q' that results from applying s to Q and P' and Q' are themselves equivalent; and vice versa.

This idea has served as the basis of a whole host of formalisms for reasoning about processes described as labeled transition systems.

It is convenient to use some notation for this relation. We write $P \approx Q$ to mean that P and Q are observationally equivalent. To make the idea of a sequence of actions (say s) being *applied* to a process P yielding P' , we write $P \xRightarrow{s} P'$. With this notation we can make the definition more precise as follows:

$P \approx Q$ if, and only if, for all sequences of actions s

- i.) Whenever $P \xRightarrow{s} P'$ then, for some Q' , $Q \xRightarrow{s} Q'$ and $P' \approx Q'$.
- ii.) Whenever $Q \xRightarrow{s} Q'$ then, for some P' , $P \xRightarrow{s} P'$ and $P' \approx Q'$.

In the rest of this section, we describe the formalization of this idea in a form suitable for proving equivalence of Faustus processes.

3.1 Faustus Processes

The operational semantics of Faustus is formalized in Isabelle/HOL [3]. The small step semantics of Faustus are given as an inductive relation over configurations which consist of a Faustus contract together with the Faustus state. The small step semantics define an evaluator for Faustus. Configurations correspond to CCS processes. Transitions in the small step semantics are either observable, and labeled with an action; or unobservable, and labeled with a τ . Additionally, Faustus processes contain a set of actions that become restricted and unobservable to the user. This formalization serves as the basis of our development of observational equivalence.

► **Definition 1** (Blockchain Primitives). ***String** is the set of all strings. **Party** is the set of all blockchain addresses. **Curr** is the set of all blockchain currencies or kinds of tokens.*

► **Definition 2** (Labeled Transition System). *A labeled transition system (LTS) is a triple $\langle S, \mathcal{L}, \Sigma \rangle$ where S is a set of states, \mathcal{L} is a set of labels, and $\Sigma \subseteq (S \times \mathcal{L} \times S)$ is a transition relation (see e.g. [34]).*

We now provide definitions with the end result of defining an LTS for Faustus processes. The states of the Faustus LTS will be the contract, its execution state, and a restriction set. The transition relation is defined by the operational semantics of Faustus. The labels will be actions paired with a time window when they are generated.

► **Definition 3** (Faustus Execution *State*).

$$\text{State} = \langle \text{Context}, \text{Env}, \text{Acct}, \text{Choice}, (\mathbb{Z}, \mathbb{Z}), [\text{Payment}], [\text{Assertion}] \rangle$$

Where **Context** is a typing context (a map of variable names to types); **Env** is an environment (a map of variable names to their meanings); **Acct** are account balances (a map of elements of $(\text{Party}, \text{Curr})$ pairs to integers); **Choice** are the choices that have been

⁴ This is exactly what Milner [31, 32], Hoare [20], and Park [39] did.

made (a map of choice name and party pairs to integers); the **start** and **end** times of the current transaction window are given by a pair of integers interpreted as POSIX Times; [**Payment**] is the history of payments; and [**Assertion**] is the history of failed assertions. We use variables $\{\sigma, \sigma_1, \sigma_2, \dots\}$ to denote arbitrary execution states.

In practice, we use a small step semantics to define the transition relation used to evaluate Faustus processes (see Definition 8). The small step semantics manage the actions in the restriction set to make them unobservable to parties observing the progress of a running contract.

There are three ways to interact with a contract running on the blockchain. They are the choice, deposit, and notify actions.

A choice action allows a party to provide an integer value for a specific choice, labeled by a string. For each choice action $\langle \text{choice}, p, c, x \rangle$ encountered during the evaluation of a contract, the choice-party pair is mapped to the integer value x in the execution state (see Definition 3). This data can be referred to later, both as part of the logic of the contract itself and by other parties, to guide future behavior. The party and choice in the user action must match a party and choice specified in a *choice guard* of the current **when**. In a choice guard, choice values are specified to be in a list of range values, $[[y_1, z_1], \dots [y_k, z_k]]$. A guard is triggered if, for the particular choice value x , if $y_i \leq x \leq z_i$ for some $i \in \{1..k\}$.

A deposit action $\langle \text{deposit}, p_1, x, c, p_2 \rangle$ specifies that party p_1 (recall, strictly speaking, parties are addresses on the blockchain⁵) deposits x tokens of currency c into party p_2 's account in the contract. During the execution of a contract, the contract itself holds the deposited funds. These funds may be paid back out of the contract to a party (an address on the blockchain) as specified by the logic of the contract. Deposit actions of the form $\langle \text{deposit}, p_1, x, c, p_2 \rangle$ are only successful if the parties, amount of the deposit and the currency match the deposit guard exactly. Funds internal to a contract are tracked in the **Acct** map of the execution state.

The notify action allows parties to *notify* a contract that it can proceed with execution. Notify guards are of the form $\text{notify } b \rightarrow \{C\}$ where b is a Boolean expression and C is the next contract to evaluate if the expression b evaluates to **true** in the current execution state. For a notify action to trigger a notify, the expression b must evaluate to true.

► **Definition 4** (User Actions). *There are three types of user actions; choice, deposit, and notify. We define the sets **Choice**, the set of all choice actions; **Dep**, the set of all deposit actions; and **Notif**, the set of all notify actions as follows:*

$$\begin{aligned} \mathbf{Choice} &\stackrel{\text{def}}{=} \{\langle \text{choice}, p, c, x \rangle \mid p \in \mathbf{Party}, c \in \mathbf{String}, x \in \mathbb{Z}\} \\ \mathbf{Dep} &\stackrel{\text{def}}{=} \{\langle \text{deposit}, p_1, x, c, p_2 \rangle \mid p_1 \in \mathbf{Party}, x \in \mathbb{Z}, c \in \mathbf{Curr}, p_2 \in \mathbf{Party}\} \\ \mathbf{Notif} &\stackrel{\text{def}}{=} \{\text{notify}\} \end{aligned}$$

We define $\mathbf{Act} \stackrel{\text{def}}{=} \mathbf{Choice} \cup \mathbf{Dep} \cup \mathbf{Notif}$.

Faustus processes are triples containing the code of the contract that is executing, the state of the executing contract, and a *restriction set*, a collection of actions that are *unobservable*⁶.

⁵ Cardano is based on an extended UTXO (Unspent Transaction Outputs) [15] model. Faustus uses an account based model built on top of the EUTXO model of the Cardano blockchain.

⁶ For readers familiar with Milner and Hoare style process algebras this is similar to restriction.

► **Definition 5** (Faustus Processes). Let **Contract** be the set of all well-formed Faustus contracts. Then we define the set of Faustus processes, **Proc**, to be the set of all pairs of Faustus contracts and execution states restricted by an action set \mathcal{A} :

$$\mathbf{Proc} \stackrel{\text{def}}{=} \{ \langle \mathcal{C}, \sigma \rangle \backslash \mathcal{A} \mid \mathcal{C} \in \mathbf{Contract}, \sigma \in \mathbf{State}, \mathcal{A} \subseteq \mathbf{Act} \}$$

Note that we write $\langle \mathcal{C}, \sigma \rangle$ when \mathcal{A} is empty instead of $\langle \mathcal{C}, \sigma \rangle \backslash \{\}$, also, we write P (or P_1, P', Q, \dots etc.) to denote arbitrary elements of $\mathbf{Contract} \times \mathbf{State}$.

Full transition labels include an action and a time window indicated by a start t_1 and end time t_2 , we use $\# = \langle t_1, t_2 \rangle$ to denote an arbitrary time window $\langle t_1, t_2 \rangle$. Due to the latency between a transaction being submitted and evaluated on the blockchain, actions are expected to be processed within the provided time window $\#$, and are rejected otherwise. The times in $\#$ are generated externally and provided in the stream of input to a running process. Typically, the time window provided allows more than enough time to be evaluated by the blockchain. These times are used to update the start and end time components of the **State** according to the Faustus semantics.

► **Definition 6** (Labels). When a Faustus process makes a transition, that transition is labeled to indicate the external/user action, a , and a time window, $\#$, during which the transaction containing the action is processed. The ϵ action allows a transition where time changes without an explicit user action. The τ label indicates an unobservable transition; a transition that does not require an external action. We define the set of all labels, **Label**, as follows:

$$\mathbf{Label} \stackrel{\text{def}}{=} \{ \langle a, \# \rangle \mid a \in \mathbf{Act} \cup \{\epsilon\}, \# \in \mathbb{Z} \times \mathbb{Z} \} \cup \{\tau\}$$

► **Definition 7** (Composition of Transition Relations). If $\mathcal{R} \subseteq A \times C$ and $\mathcal{S} \subseteq C \times B$ we write $\mathcal{R} \cdot \mathcal{S}$ to denote the relation on $A \times B$ constructed by (ordinary) composition of relations.

$$\mathcal{R} \cdot \mathcal{S} \stackrel{\text{def}}{=} \{ \langle x, y \rangle \in (A \times B) \mid \exists z \in C. x \mathcal{R} z \wedge z \mathcal{S} y \}$$

► **Definition 8** (Faustus Transition Relation). We write $\langle \mathcal{C}, \sigma \rangle \backslash \mathcal{A} \xrightarrow{\lambda} \langle \mathcal{C}', \sigma' \rangle \backslash \mathcal{A}'$ to denote transitions restricted by \mathcal{A} where $\mathcal{A}, \mathcal{A}' \subseteq \mathbf{Act}$. Under the small step semantics of Faustus⁷, contract \mathcal{C} in state σ with restricted actions \mathcal{A} transitions on input (label) λ to contract \mathcal{C}' , with state σ' , and restriction set \mathcal{A}' . Labels $\langle a, \# \rangle \in \mathbf{Act} \times \mathbb{Z}^2$ are hidden in the sense that transitions made on labels containing actions in \mathcal{A} are not observable⁸. When $\mathcal{A} = \mathcal{A}'$ (and it always will be in this paper) we write $\langle \mathcal{C}, \sigma \rangle \backslash \mathcal{A} \xrightarrow{\lambda} \langle \mathcal{C}', \sigma' \rangle \backslash \mathcal{A}$. In the case where $P \backslash \mathcal{A} \xrightarrow{\lambda_1} R_1 \backslash \mathcal{A}$, $R_1 \backslash \mathcal{A} \xrightarrow{\lambda_2} R_2 \backslash \mathcal{A}$, and so on, up to $R_{k-1} \backslash \mathcal{A} \xrightarrow{\lambda_k} Q \backslash \mathcal{A}$, we write $P \backslash \mathcal{A} \xrightarrow{\lambda_1 \dots \lambda_k} Q \backslash \mathcal{A}$ (instead of $P \backslash \mathcal{A} (\xrightarrow{\lambda_1} \dots \xrightarrow{\lambda_k}) Q \backslash \mathcal{A}$) to denote a transition in the relation formed by the composition of $\xrightarrow{\lambda_1} \dots \xrightarrow{\lambda_k}$. In the case where there is a vector of actions, $\vec{\alpha} = \langle \alpha_1, \dots, \alpha_k \rangle$, occurring sequentially with the same time window, we write $P \backslash \mathcal{A} \xrightarrow{\langle \vec{\alpha}, \# \rangle} Q \backslash \mathcal{A}$ instead of $P \backslash \mathcal{A} \xrightarrow{\langle \alpha_1, \# \rangle \dots \langle \alpha_k, \# \rangle} Q \backslash \mathcal{A}$. The full transition relation is defined by the following equations:

- 1.) $P \backslash \mathcal{A} \xrightarrow{\langle b, \# \rangle}_0 Q \backslash \mathcal{A} \stackrel{\text{def}}{=} \{ \langle P \backslash \mathcal{A}, \langle b, \# \rangle, Q \backslash \mathcal{A} \rangle \mid b \notin \mathcal{A} \wedge P \xrightarrow{\lambda} Q \}$
- 2.) $P \backslash \mathcal{A} \xrightarrow{\langle b, \# \rangle}_k Q \backslash \mathcal{A} \stackrel{\text{def}}{=} \{ \langle P \backslash \mathcal{A}, \langle b, \# \rangle, Q \backslash \mathcal{A} \rangle \mid k > 0 \wedge b \neq \epsilon \wedge b \notin \mathcal{A} \wedge \exists \vec{\alpha} \in \mathcal{A}^k. P \xrightarrow{\langle \vec{\alpha}, b, \# \rangle} Q \}$
- 3.) $P \backslash \mathcal{A} \xrightarrow{\langle b, \# \rangle} Q \backslash \mathcal{A} \stackrel{\text{def}}{=} \bigcup_{k \in \mathbb{N}} \{ P \backslash \mathcal{A} \xrightarrow{\langle b, \# \rangle}_k Q \backslash \mathcal{A} \}$
- 4.) $P \backslash \mathcal{A} \xrightarrow{\tau} Q \backslash \mathcal{A} \stackrel{\text{def}}{=} \{ \langle P \backslash \mathcal{A}, \tau, Q \backslash \mathcal{A} \rangle \mid P \xrightarrow{\tau} Q \}$

⁷ Formally, in the Faustus semantics, $\langle \mathcal{C}, \sigma \rangle \xrightarrow{\lambda} \langle \mathcal{C}', \sigma' \rangle$ is defined by $\llbracket \Gamma, \mathcal{A} \vdash \mathcal{C} : \text{contract} \rrbracket \sigma \lambda = \langle \mathcal{C}', \sigma' \rangle$.

⁸ In CCS [34] unobservable τ -transitions arise by the synchronization of a name (a label) say λ and its co-name $\bar{\lambda}$.

Note that $\xrightarrow{\lambda}^*$ denotes the reflexive transitive closure of $\xrightarrow{\lambda}$, i.e. it is the relation defined by the composition of any number of $\xrightarrow{\lambda}$ relations including 0.

The Faustus transition relation allows an arbitrary number of hidden actions from the restriction set to occur before an observable transition. In this way, the restriction set not only prevents the users from sending actions in the set, but it also corresponds to a process running concurrently with the Faustus smart contract that can send any of the actions in the restriction set. Regular τ -transitions and timeout transitions remain unchanged by the restriction set.

► **Definition 9** (Faustus Labeled Transition System). *The Faustus LTS is defined as the triple $\langle \mathbf{Proc}, \mathbf{Label}, \rightarrow \rangle$. The states of the Faustus transition relation are elements of \mathbf{Proc} . The labels of the Faustus transition system are elements of \mathbf{Label} . The transition relation \rightarrow is defined in Definition 8, and given by the small step semantics of Faustus; the details of which have been formalized in Isabelle/HOL [3].*

Typically, users interact with contracts using an empty set of restricted actions. Non-empty sets of restricted actions allow for traversing different contract structures through multiple transactions without requiring additional user interaction. This will be explained in more detail in Section 4.

With the definition of the Faustus transition relation, we also define the experiment relation between Faustus processes on sequences of labels.

► **Definition 10** (Experiment Relation). *Let $s = \lambda_1 \cdots \lambda_n \in \mathbf{Label}^*$. We define the experiment relation \xRightarrow{s} as follows:*

$$\begin{aligned} \xRightarrow{s} &\stackrel{\text{def}}{=} \xrightarrow{\tau}^* \\ \xRightarrow{s} &\stackrel{\text{def}}{=} \Rightarrow \cdot \xrightarrow{\lambda_1} \cdot \Rightarrow \cdots \Rightarrow \cdot \xrightarrow{\lambda_n} \cdot \Rightarrow \end{aligned}$$

Thus, \xRightarrow{s} is the behavior that allows an arbitrary number of τ actions before, between, and after the observable actions in s .

3.2 Observational Equivalence

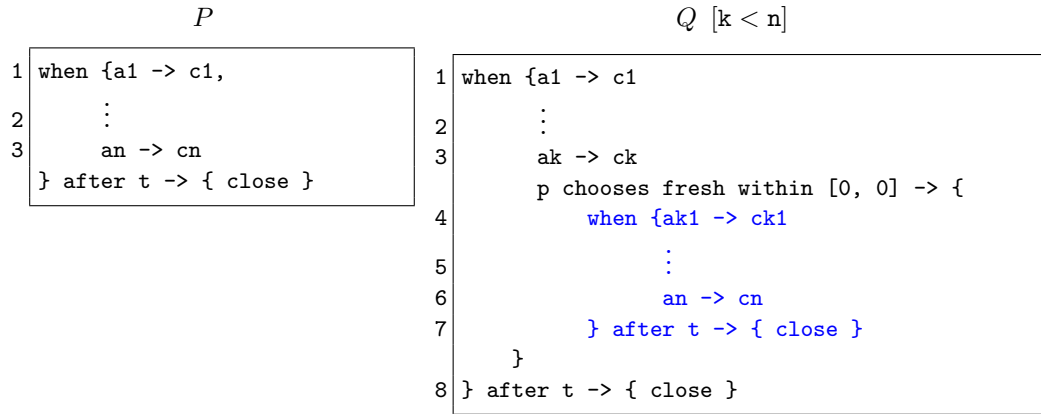
A *bisimulation* is a binary relation between states of processes. It identifies states that cannot be distinguished from one another by any sequence of observable actions. From the perspective of a user interacting with two bisimilar processes, the same inputs result in the same outputs for both processes. The user cannot tell the difference between the two processes by interacting with them. If two processes, P and Q , are bisimilar, we write $P \approx Q$. It turns out that \approx is an equivalence relation [35].

► **Definition 11** (Weak Simulation). *Let $\mathcal{F} = \langle \mathbf{Proc}, \mathbf{Label}, \rightarrow \rangle$ be the Faustus LTS, and let $\mathcal{R} \subseteq (\mathbf{Proc} \times \mathbf{Proc})$ be a binary relation, and $s \in \mathbf{Label}^*$. \mathcal{R} is a weak simulation over \mathcal{F} when the following property holds:*

If $(P_1, Q_1) \in \mathcal{R}$ and $P_1 \xRightarrow{s} P_2$ then there exists $Q_2 \in \mathcal{S}$ such that $Q_1 \xRightarrow{s} Q_2$ and $(P_2, Q_2) \in \mathcal{R}$

This definition leads to the definitions of weak bisimulation and observational equivalence.

► **Definition 12** (Weak Bisimulation, Observational Equivalence). *Let $\mathcal{F} = \langle \mathbf{Proc}, \mathbf{Label}, \rightarrow \rangle$ be the Faustus LTS, and let $\mathcal{R} \subseteq (\mathbf{Proc} \times \mathbf{Proc})$ be a binary relation. Then \mathcal{R} is a weak bisimulation over \mathcal{F} , if it and its converse are both weak simulations. We say P and Q are observationally equivalent, written $P \approx Q$, if there exists a weak bisimulation \mathcal{R} such that $(P, Q) \in \mathcal{R}$.*



■ **Figure 2** A contract (P) and the result (Q) of recursively applying the transformation.

Processes are called observationally equivalent when there is a bisimulation between them. The number of unobservable transitions may differ when evaluating two observationally equivalent processes on the same observable transitions. We have proved in Isabelle/HOL that bisimulations on Faustus processes are equivalence relations, and that simulations are closed under union. The proofs of these properties are well understood, and can be found in multiple works by Milner [33, 34, 35, 36].

With the definition of observational equivalence, we can now prove transformations of Faustus contracts do not modify their observed behaviors. The next section describes a common transformation that is applied to Faustus contracts, and proves the result of the transformation is observationally equivalent to the original.

4 Contract Transformations

In each round of computation on the blockchain, Faustus programs run until they pause for external input at a `when` contract, or are closed. The computation time between each pause may take longer than the maximum computation time allowed per transaction on the blockchain, leading to out of gas errors [1]. If an out of gas error occurs, the contract will revert back to its state prior to that round of computation. Reverting to the prior state can result in code that never progresses while still generating gas costs.

Consider the contract on the left side of Figure 2. Assume that in each round of computation there is only enough time to check $k + 1$ guards before an out of gas error occurs. Then there are $n - (k + 1)$ guard continuations that are unreachable when running the contract on the blockchain. On the right side of Figure 2, the sequence of n guards has been replaced by a length $k + 1$ sequence of guards, the last of which is a fresh choice guard that cascades to a `when` containing the rest of the transformed contract. We have colored this section blue to indicate it will be recursively transformed.

To allow every guarded command to be reachable, programmers apply this transformation where the list of guarded commands is truncated, and moved into a sub-contract that runs after a “fresh” choice action is taken. The key aspect is selecting a choice name that does not appear in the original contract, *i.e.* it is fresh, hence $(p \text{ chooses } \textit{fresh} \text{ within } [0, 0])^9$ is a choice guard that does not occur in the original contract. The result of the transformation

⁹ The range $[0, 0]$ simply means p must choose 0.

```

(* sg - split guards *)
function sg :: "ChoiceName ⇒ FContract ⇒ FContract" where
"sg cn (When guards t cont) = (
  let
    newGuards = map (λ(Case g c) ⇒ (Case g (sg cn c))) guards;
    newCont = sg cn cont
  in
    if (k > 1 ∧ length newGuards > k)
    then (When (take k newGuards @
      [Case (ActionGuard (Choice (FChoiceId cn p) [(0, 0)]))
        (sg cn (When (drop k guards) t cont))]))
      t newCont)
    else (When newGuards t newCont))" |
"sg cn (Close) = Close" |
"sg cn (StatementCont s c) = (StatementCont s (sg cn c))" |
"sg cn (If e c1 c2) = (If e (sg cn c1) (sg cn c2))" |
"sg cn (Let i v c) = (Let i v (sg cn c))" |
"sg cn (LetObservation i v c) = (LetObservation i v (sg cn c))" |
"sg cn (LetPubKey i v c) = (LetPubKey i v (sg cn c))" |
"sg cn (LetC i p b c) = (LetC i p (sg cn b) (sg cn c))" |
"sg cn (UseC i a) = (UseC i a)"

```

■ **Figure 3** Isabelle implementation of the transformation algorithm.

applied to P in Figure 2 is Q in the same figure. As we will see below, the “fresh” choice allows for a transition that leads directly to a **when** contract. Since **when** contracts pause execution until the next round of blockchain computation, the out of gas error is avoided, and the rest of the guard continuations can be reached in the next round of computation.

This type of transformation has been informally described by Bush [14] as a best practice for Marlowe contracts with complex logic. The Isabelle function in Figure 3, which acts on the Faustus abstract syntax tree from Appendix A, formally describes the transformation.

► **Definition 13** (Transformation Constants). *The transformation function in Figure 3 has two constants associated with it, p and k . The first one, p , is an arbitrary **Party** that will be allowed to send signals containing the $\langle \text{choice}, p, c, 0 \rangle$ action that traverses the cascading **when** contracts; and k is a natural number representing the maximum number of guarded commands before a split into cascading **when** contracts should be performed. These constants can take on any values of the appropriate type.*

Evaluation of different guard expressions may take different amounts of time; thus, experimentation may be required to determine the safe value for k for a specific contract.

► **Definition 14** (Transformed Contracts). *Given a contract C , and choice name c , we use the notation $\lceil C \rceil_c$ to denote the result of applying the transformation $(\text{sg } c)$ to C . We also overload the notation for $\lceil \langle C, \sigma \rangle \rceil_c$ to denote application of the transformation to C as well as all contracts mentioned in the environment in σ . See Definition 3 for the definition of the environment in σ .*

In the example from Figure 2, $Q = \lceil P \rceil_{\text{fresh}}$. To verify this transformation, we must show that there is a weak bisimulation between them for an arbitrary restriction set \mathcal{A} , i.e. that $\langle P, \sigma \rangle \setminus \mathcal{A} \approx \langle \lceil P \rceil_{\text{fresh}}, \sigma \setminus (\mathcal{A} \cup \{\langle \text{choice}, p, \text{fresh}, 0 \rangle\})$.

<pre> 1 when { 2 a -> c1, 3 a -> c2 4 } after t -> { close } </pre>	<pre> 1 when { 2 a -> c1, 3 p chooses fresh within [0, 0] -> { 4 when { 5 a -> c2 6 } after t -> { close } 7 } 8 } after t -> { close } </pre>
--	---

■ **Figure 4** A contract without disjoint guards before and after applying the transformation.

4.1 Restriction - Making Observable Actions Unobservable

The strategy of the transformation is to transform a **when** contract having a long list of guarded commands into a new contract having a structure of cascading **when** contracts whose guarded command lists are length less than or equal to $k + 1$. To accomplish this, we introduce a choice action with a choice name that does not occur in the original contract called c . The guard in the contract takes the form $(p \text{ chooses } c \text{ within } [0, 0])$, so that inputs of the form $\langle \langle \text{choice}, p, c, 0 \rangle, \# \rangle$ induce a transition.

► **Definition 15** (Restricted Choice Action). *Given a choice name $c \in \mathbf{String}$ we define the choice action, $\alpha_c \stackrel{\text{def}}{=} \langle \text{choice}, p, c, 0 \rangle$.*

The action α_c is used to traverse the cascading **when** structure introduced by the transformation. As long as c does not occur as choice name in P , α_c can safely be used to internally transition along the chain of cascading **whens** introduced by **sg** in $\lceil P_c \rceil$ (see Figure 3). Adding α_c to the restriction set hides those transitions when evaluating $\lceil P_c \rceil$. This allows us to show that there is a bisimulation between $P \setminus \mathcal{A}$ and $\lceil P_c \rceil \setminus (\mathcal{A} \cup \{\alpha_c\})$.

4.2 Verifying the Transformation

To be a candidate for the transformation requires a kind of predictability in the behavior of the contract to be transformed. Specifically, the guards in every **when** having more than k guarded commands must be disjoint, in the sense that at most one guard evaluates to true in any state-action pair.

The example contracts in Figure 4 demonstrate the issue that occurs if the guards are not disjoint. In the original contract on the left there is no way to get to the **c2** continuation. This is because the guards in a **when** are evaluated in the order they appear in the list. If the guarded command $(a \rightarrow c1)$ is triggered, then $(a \rightarrow c2)$ would have been triggered as well. $(a \rightarrow c1)$ will always continue to **c1** before $(a \rightarrow c2)$ is ever evaluated. On the other hand, the transformed contract on the right can reach both **c1** and **c2**. If the label $\langle a, \# \rangle$ reaches **c1**, then the sequence $\langle \alpha_{\text{fresh}} \cdot a, \# \rangle$ reaches **c2**, which was previously unreachable. Also note that α_{fresh} is hidden, so to any observer, the labels $\langle \alpha_{\text{fresh}} \cdot a, \# \rangle$ and $\langle a, \# \rangle$ appear to be identical. As this example shows, the transformed contract may have more behaviors than the original contract, and there is no evaluation of the original contract that will simulate those behaviors. By requiring the guards to be disjoint, we guarantee that there is no unreachable code uncovered by the transformation. In practice, most contracts are written to have disjoint guards. For example, the three-party escrow contract in Figure 1 has disjoint guards since all choice guards in the **when** are for different parties or allow different choices of values for the vote, and there is only one deposit guard.

While outside the scope of this work, Appendix B describes a transformation that will make all guards in the lists of guarded commands disjoint for contracts where programmers find the need to write non-disjoint guards. The transformation would maintain the behavior of the contract by prioritizing the guard that appears earlier in the **when**. By modifying the logic of the later guards, it is possible to remove the logic that overlaps with the earlier guard while maintaining the non-overlapping logic. Since bisimulations are equivalence relations, once a bisimulation is shown to exist using the disjoint transformation, the disjoint transformation can be composed with the **sg** transformation, and the result would be observationally equivalent to the original contract.

► **Definition 16** (Disjoint Guards). *A contract-state pair $\langle C, \sigma \rangle$ is disjoint if, for each contract in $\langle C, \sigma \rangle$ of the form (when gs after $t \rightarrow cont$), the list of guarded commands gs is disjoint. gs is disjoint when, for all l and r such that $gs = l \mathrel{++} r$, the following two properties hold in for all actions in all execution states:*

- i.) *If l contains a guard that evaluates to true then all guards in r evaluate to false.*
- ii.) *If r contains a guard that evaluates to true then all guards in l evaluate to false.*

Next, we define a binary relation between original processes and their transformed counterparts, and prove it is a bisimulation.

► **Definition 17** (Split Guards Relation). *For all $c \in \mathbf{String}$ let \mathbb{SG}_c be the following binary relation over \mathbf{Proc} :*

$$\begin{aligned} \mathbb{SG}_c \stackrel{\text{def}}{=} \{ & \langle \langle C, \sigma_1 \rangle \backslash \mathcal{A}, \lceil C, \sigma_2 \rceil_c \backslash (\mathcal{A} \cup \{\alpha_c\}) \rangle \\ & \mid c \text{ fresh in } \langle C, \sigma_1 \rangle, \text{ disjoint } \langle C, \sigma_1 \rangle, \\ & \sigma_1 = \langle \Gamma, e, accts, choices_1, t, ps, asrt \rangle, \\ & \sigma_2 = \langle \Gamma, e, accts, choices_2, t, ps, asrt \rangle, \\ & \forall x \in \mathbf{String}, p \in \mathbf{Party}. x \neq c \text{ implies } choices_1[(x, p)] = choices_2[(x, p)] \} \end{aligned}$$

The binary relation \mathbb{SG}_c is a relation in $\mathbf{Proc} \times \mathbf{Proc}$, where the second element is the result of applying the transformation function (**sg** c) to the first element. Also note, the original contract must have disjoint guarded command lists; and the Faustus execution states in the two elements of the relation are equivalent up to the choices for the new choice name c . We will use \mathbb{SG}_c to show a bisimulation between an original contract and a transformed contract under the condition that the choice name used in the transformation is not in the original contract.

► **Lemma 18.** *For all $c \in \mathbf{String}$, \mathbb{SG}_c is a weak simulation over $(\mathbf{Proc}, \mathbf{Label}, \rightarrow)$.*

► **Lemma 19.** *For all $c \in \mathbf{String}$, \mathbb{SG}_c^{-1} is a weak simulation over $(\mathbf{Proc}, \mathbf{Label}, \rightarrow)$.*

Proof. The proofs of Lemmas 18 and 19 are done in Isabelle/HOL by induction on the structure of the experiment relation (\Rightarrow), and by cases on the structure of the transition relation (\rightarrow). Assume membership in the relation (\mathbb{SG}_c or \mathbb{SG}_c^{-1}) and an arbitrary transition for the first member of the relation. Then for each case of the transition relation, we show the second member of the relation will transition on an experiment containing the same label to another element of \mathbf{Proc} that maintains membership in the relation (\mathbb{SG}_c or \mathbb{SG}_c^{-1}). ◀

► **Theorem 20** (Transformation Verification). *Let $\langle C, \sigma_1 \rangle \backslash \mathcal{A}, \lceil C, \sigma_2 \rceil_c \backslash (\mathcal{A} \cup \{\alpha_c\}) \in \mathbf{Proc}$ be states in the Faustus LTS $(\mathbf{Proc}, \mathbf{Label}, \rightarrow)$, for some arbitrary $c \in \mathbf{String}$ such that:*

- i.) *c is fresh for C and σ_1 , i.e. c does not occur as a choice name in either.*
- ii.) *Guarded command lists in $\langle C, \sigma_1 \rangle$ are disjoint.*

iii.) $\sigma_1 = \langle \Gamma, e, accts, chs_1, t, ps, asrt \rangle$
 iv.) $\sigma_2 = \langle \Gamma, e, accts, chs_2, t, ps, asrt \rangle$,
 v.) $\forall x \in \mathbf{String}, r \in \mathbf{Party}. x \neq c \text{ implies } chs_1[(x, r)] = chs_2[(x, r)]$.
 Then $\langle C, \sigma_1 \rangle \backslash \mathcal{A} \approx \langle C, \sigma_2 \rangle_c \backslash (\mathcal{A} \cup \{\alpha_c\})$.

Proof. Assume an arbitrary $c \in \mathbf{String}$, $\langle C, \sigma_1 \rangle \backslash \mathcal{A}, \langle C, \sigma_2 \rangle_c \backslash (\mathcal{A} \cup \{\alpha_c\}) \in \mathbf{Proc}$, and that (i.) through (v.) hold. From Lemma 18, Lemma 19, and Definition 12 of bisimulation, we can conclude that $\mathbb{S}\mathbb{G}_c$ is a bisimulation. Then by the definition of $\mathbb{S}\mathbb{G}_c$, we know that $\langle \langle C, \sigma_1 \rangle \backslash \mathcal{A}, \langle C, \sigma_2 \rangle_c \backslash (\mathcal{A} \cup \{\alpha_c\}) \rangle \in \mathbb{S}\mathbb{G}_c$. Thus $\langle C, \sigma_1 \rangle \backslash \mathcal{A} \approx \langle C, \sigma_2 \rangle_c \backslash (\mathcal{A} \cup \{\alpha_c\})$. ◀

With Theorem 20 we have verified contracts have the same observable behavior before and after the transformation is applied. Programmers can safely apply the transformation to their contracts using the initial execution state and empty restriction set before they are loaded onto the blockchain. The conditions required to guarantee the bisimulation is valid are: i.) the choice name used in the transformation is fresh, and ii.) the guarded command lists in the original contract are all disjoint.

5 Conclusions and Future Work

In this paper we have described an Isabelle formalization of a bisimulation verifying that smart contracts written in Faustus and their transformed instances are observationally equivalent. The formalization introduces a notion of hidden actions in a way different from Milner's CCS [34]. Also, Hoare's CSP [21] has a restriction operator, but it differs in a number of ways; perhaps most significantly, his restriction sets enumerate the allowable actions.

There are a few clear ways this work can be continued. The first is that we plan to apply this methodology to many more transformations. First among them will be to formalize, in Isabelle, one that transforms an arbitrary guarded command list into one that is disjoint. A strategy for this transformation is described in Appendix B. Since the verification of the transformation in this paper requires the guarded command lists in the contract to be disjoint, it would be useful for programmers to have more contracts that the transformation can be applied to.

Also, we plan to apply these techniques to other smart contract programming languages (perhaps Solidity [8]). In Faustus, labels for the transition relation are clearly given in the syntax of the contract as guards in the guarded commands of a **when** contract. In Solidity, the **require** statements act like guards. Marmosler and Brucker [28] have formalized the semantics of Solidity in Isabelle/HOL, which would serve as a good starting point.

Another area of investigation is applying the additional methodologies used for Timed CCS. We include time information as part of the labels in the Faustus LTS and use that information to update the **State** time information according to the Faustus semantics. Timed CCS variants use clock signals as labels in processes transitions [5, 27]. In order to show a bisimulation between Faustus processes, the **State** information must be included in the relation. By moving the time information out of the **State** and using the clock labels that timed CCS variants use, it should reduce the complexity of the relations required to verify smart contract transformations.

References

- 1 Cardano protocol parameters reference guide | Cardano Docs. URL: <https://docs.cardano.org/about-cardano/explore-more/parameter-guide>.

- 2 Internet Crime Complaint Center (IC3) | Cyber Criminals Increasingly Exploit Vulnerabilities in Decentralized Finance Platforms to Obtain Cryptocurrency, Causing Investors to Lose Money. URL: <https://www.ic3.gov/PSA/2022/PSA220829>.
- 3 isabelle · master · Secure Systems Collaborative / Public / WABL / Faustus V2 · GitLab, February 2025. URL: <https://gitlab.com/UWyo-SSC/public/wabl/faustus-v2/-/tree/master/isabelle>.
- 4 Elvira Albert, Pablo Gordillo, Alejandro Hernández-Cerezo, Albert Rubio, and Maria A. Schett. Super-optimization of smart contracts. *ACM Trans. Softw. Eng. Methodol.*, 31(4):70:1–70:29, July 2022. doi:10.1145/3506800.
- 5 Henrik Reif Andersen and Michael Mendler. An asynchronous process algebra with multiple clocks. In Donald Sannella, editor, *Programming Languages and Systems — ESOP ’94*, pages 58–73, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.
- 6 Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Grue Simonsen, and Christian Stefansen. Compositional specification of commercial contracts. *International Journal on Software Tools for Technology Transfer*, 8(6):485–516, November 2006. doi:10.1007/s10009-006-0010-1.
- 7 Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust*, pages 164–186, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg. doi:10.1007/978-3-662-54455-6_8.
- 8 The Solidity Authors. Solidity — Solidity 0.8.29 documentation, 2024. URL: <https://docs.soliditylang.org/en/latest/>.
- 9 Massimo Bartoletti, Angelo Ferrando, Enrico Lipparini, and Vadim Malvone. Solvent: Liquidity Verification of Smart Contracts. In Nikolai Kosmatov and Laura Kovács, editors, *Integrated Formal Methods*, pages 256–266, Cham, 2025. Springer Nature Switzerland. doi:10.1007/978-3-031-76554-4_14.
- 10 Massimo Bartoletti, Riccardo Marchesin, and Roberto Zunino. Secure compilation of rich smart contracts on poor utxo blockchains. In *2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P)*, pages 235–267, 2024. doi:10.1109/EuroSP60621.2024.00021.
- 11 Massimo Bartoletti and Roberto Zunino. BitML: A Calculus for Bitcoin Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS ’18*, pages 83–100, New York, NY, USA, October 2018. Association for Computing Machinery. doi:10.1145/3243734.3243795.
- 12 Massimo Bartoletti and Roberto Zunino. Verifying liquidity of bitcoin contracts. In Flemming Nielson and David Sands, editors, *Principles of Security and Trust*, pages 222–247, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-17138-4_10.
- 13 Sidi Mohamed Beillahi, Gabriela Ciocarlie, Michael Emmi, and Constantin Enea. Behavioral simulation for smart contracts. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020*, pages 470–486, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3385412.3386022.
- 14 Brian Bush. marlowe-cardano/marlowe/best-practices.md at main · input-output-hk/marlowe-cardano · GitHub, January 2023. URL: <https://github.com/input-output-hk/marlowe-cardano/blob/main/marlowe/best-practices.md#avoid-or-break-up-complex-logic-in-the-contract>.
- 15 Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The extended utxo model. In *Financial Cryptography and Data Security: FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers*, pages 525–539, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-54455-3_37.
- 16 James Chapman, Arnaud Bailly, and Polina Vinogradova. Applying Continuous Formal Methods to Cardano (Experience Report). In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Software Architecture*, pages 18–24, Milan Italy, August 2024. ACM. doi:10.1145/3677998.3678222.

- 17 Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 442–446, 2017. doi:10.1109/SANER.2017.7884650.
- 18 RChain Cooperative. Contract Design — RChain Architecture 0.9.0 documentation, 2017. URL: <https://architecture-docs.readthedocs.io/contracts/contract-design.html>.
- 19 Fritz Henglein, Christian Kjær Larsen, and Agata Murawska. A formally verified static analysis framework for compositional contracts. In Matthew Bernhard, Andrea Bracciali, L. Jean Camp, Shin'ichiro Matsuo, Alana Maurushat, Peter B. Rønne, and Massimiliano Sala, editors, *Financial Cryptography and Data Security*, pages 599–619, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-54455-3_42.
- 20 C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. doi:10.1145/359576.359585.
- 21 C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall International series in computer science. Prentice/Hall International, Englewood Cliffs, N.J, 1985.
- 22 Philipp Kant, Kevin Hammond, Duncan Coutts, James Chapman, Nicholas Clarke, Jared Corduan, Neil Davies, Javier Díaz, Matthias Güdemann, Wolfgang Jeltsch, Marcin Szamotulski, and Polina Vinogradova. Flexible formality practical experience with agile formal methods. In Aleksander Byrski and John Hughes, editors, *Trends in Functional Programming*, pages 94–120, Berlin, Heidelberg, 2020. Springer International Publishing. doi:10.1007/978-3-030-57761-2_5.
- 23 Olga Kharif. Crypto-Bridge Hacks Reach Over \$1 Billion in Little Over a Year. *Bloomberg.com*, March 2022. URL: <https://www.bloomberg.com/news/articles/2022-03-30/crypto-bridge-hacks-reach-over-1-billion-in-little-over-a-year>.
- 24 Pablo Lamela Seijas, Alexander Nemish, David Smith, and Simon Thompson. Marlowe: Implementing and analysing financial contracts on blockchain. In Matthew Bernhard, Andrea Bracciali, L. Jean Camp, Shin'ichiro Matsuo, Alana Maurushat, Peter B. Rønne, and Massimiliano Sala, editors, *Financial Cryptography and Data Security*, pages 496–511, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-54455-3_35.
- 25 Pablo Lamela Seijas, David Smith, and Simon Thompson. Efficient static analysis of marlowe contracts. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, pages 161–177, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-61467-6_11.
- 26 Pablo Lamela Seijas and Simon Thompson. Marlowe: Financial contracts on blockchain. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, pages 356–375, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-030-03427-6_27.
- 27 Luigi Liquori and Michael Mendler. Strong Priority and Determinacy in Timed CCS. Technical report, Inria ; University of Bamberg, December 2023. URL: <https://inria.hal.science/hal-04367635>.
- 28 Diego Marmosoler and Achim D. Brucker. A Denotational Semantics of Solidity in Isabelle/HOL. In Radu Calinescu and Corina S. Păsăreanu, editors, *Software Engineering and Formal Methods*, volume 13085, pages 403–422. Springer International Publishing, Cham, 2021. Series Title: Lecture Notes in Computer Science. doi:10.1007/978-3-030-92124-8_23.
- 29 Kegan McIlwaine, Stone Olguin, and James Caldwell. Faustus: Adding formally verified parameterized abstractions to the smart contract language marlowe, 2022. Unpublished Manuscript. URL: https://gitlab.com/UWyo-SSC/public/wabl/faustus-v2/-/blob/master/papers/WTSC22_Authored.pdf.
- 30 Kegan McIlwaine, Stone Olguin, and James Caldwell. Developing faustus: A formally verified smart contract programming language, 2023. Unpublished Manuscript. URL: https://gitlab.com/UWyo-SSC/public/wabl/faustus-v2/-/blob/master/papers/iFM_2023.pdf.

- 31 Robin Milner. An algebraic definition of simulation between programs. In D. C. Cooper, editor, *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*. London, UK, September 1-3, 1971, pages 481–489. William Kaufmann, 1971. URL: <http://ijcai.org/Proceedings/71/Papers/044.pdf>.
- 32 Robin Milner. Processes: A mathematical model of computing agents. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 157–173. Elsevier, 1975. doi:10.1016/S0049-237X(08)71948-7.
- 33 Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 1980. doi:10.1007/3-540-10235-3.
- 34 Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., USA, 1989.
- 35 Robin Milner. Operational and algebraic semantics of concurrent processes. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 1201–1242. Elsevier and MIT Press, 1990. doi:10.1016/B978-0-444-88074-1.50024-X.
- 36 Robin Milner. *Communicating and mobile systems: the π -calculus*. Cambridge University Press, USA, 1999.
- 37 Keerthi Nelaturu, Sidi Mohamed Beillahi, Fan Long, and Andreas Veneris. Smart contracts refinement for gas optimization. In *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, pages 229–236, 2021. doi:10.1109/BRAINS52497.2021.9569819.
- 38 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. doi:10.1007/3-540-45949-9.
- 39 David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science*, pages 167–183, Berlin, Heidelberg, 1981. Springer. doi:10.1007/BFb0017309.
- 40 Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 280–292, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/351240.351267.
- 41 Meixun Qu, Xin Huang, Xu Chen, Yi Wang, Xiaofeng Ma, and Dawei Liu. Formal Verification of Smart Contracts from the Perspective of Concurrency. In Meikang Qiu, editor, *Smart Blockchain*, pages 32–43, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-030-05764-0_4.
- 42 Ilya Sergey and Aquinas Hobor. A concurrent perspective on smart contracts. In Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson, editors, *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 478–493, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-70278-0_30.
- 43 Nick Szabo. Smart Contracts, 1994. URL: <https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/L0Twinterschool2006/szabo.best.vwh.net/smart.contracts.html>.
- 44 Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. A survey of smart contract formal specification and verification. *ACM Comput. Surv.*, 54(7), July 2021. doi:10.1145/3464421.

```

datatype FAction =
  Deposit FParty FParty Token FValue
| Choice FChoiceId "Bound list"
| Notify FObservation

datatype FStatement =
  Pay FAccountId FPayee Token FValue
| Assert FObservation
| ReassignVal Identifier FValue
| ReassignObservation Identifier FObservation
| ReassignPubKey Identifier FParty

datatype FGuardExpression =
  ActionGuard FAction
| GuardThenGuard FGuardExpression FGuardExpression
| GuardStmtsGuard FGuardExpression "FStatement list"
| DisjointGuard FGuardExpression FGuardExpression
| InterleavedGuard FGuardExpression FGuardExpression

datatype FContract =
  Close
| StatementCont FStatement FContract
| If FObservation FContract FContract
| When "FCase list" Timeout FContract
| Let Identifier FValue FContract
| LetObservation Identifier FObservation FContract
| LetPubKey Identifier FParty FContract
| LetC Identifier "FParameter list" FContract FContract
| UseC Identifier "FArgument list"
and FCase = Case FGuardExpression FContract

```

■ **Figure 5** Isabelle: constructors for Statements, Guard Expressions, and Contracts.

A Faustus Abstract Syntax Tree

The Isabelle data type definitions in Figure 5 are the result of parsing a Faustus program. Each constructor represents a different type of the same syntactical object. The constructors of `FContract` each define a type of Faustus contract. Similarly, the constructors of `FGuardExpression` and `FStatement` each define a type of guard expression or statement that can occur in a Faustus contract, respectively. The constructors of `FAction` give the types of basic action guards for the actions that users can take while interacting with Faustus processes. The list inside the `When` contract is a list of `Cases`, the full constructor for a guarded command.

B The Disjoint Guards Transformation

The verification of the `sg` transformation requires guarded command lists in the original contract to be disjoint. In Figure 6 we provide an algorithm for creating disjoint guards when given two that may match on the same user action. Since guarded command lists and

```

fun left_prioritize_action :: "FState  $\Rightarrow$  FAction  $\Rightarrow$  FAction  $\Rightarrow$  FAction" where
"left_prioritize_action  $\sigma$ 
  (Deposit p11 p12 t1 v1)
  (Deposit p21 p22 t2 v2) =
  (if (evalFParty  $\sigma$  p11 = evalFParty  $\sigma$  p21)  $\wedge$ 
    (evalFParty  $\sigma$  p12 = evalFParty  $\sigma$  p22)  $\wedge$ 
    t1 = t2
  then Deposit p21 p22 t2 (Cond (ValueEQ v1 v2) (Constant (-1000)) v2)
  else Deposit p21 p22 t2 v2)" |
"left_prioritize_action  $\sigma$ 
  (Notify obs1)
  (Notify obs2) =
  (Notify (AndObs (NotObs obs1) (obs2)))" |
"left_prioritize_action  $\sigma$ 
  (Choice (FChoiceId cn1 p1) bounds1)
  (Choice (FChoiceId cn2 p2) bounds2) =
  (if (evalFParty  $\sigma$  p1 = evalFParty  $\sigma$  p2)  $\wedge$  cn1 = cn2
  then Choice (FChoiceId cn2 p2) (left_prioritize_bounds bounds1 bounds2)
  else Choice (FChoiceId cn2 p2) bounds2)" |
"left_prioritize_action  $\sigma$  _ a = a"

```

■ **Figure 6** Isabelle: constructors for Statements, Guard Expressions, and Contracts.

guard expressions prioritize the top/left, we can make contract guards disjoint by modifying the guard that has lower priority. A full Isabelle/HOL verification of this algorithm has not been completed yet, but it will use the bisimulation method described in this paper.

In Faustus, a negative deposit is not an action that a user can make. Thus, if two **deposits** are identical, to avoid a situation where the **deposit** guards overlap, we disable the second deposit guard by making its value negative. In this way, we make two **deposit** guards disjoint while prioritizing the first. Note that disabling the second will have no effect, because it is identical to the earlier one.

The algorithm handles **choice** guards much like **deposit** guards. Checking the equality of the choice names and parties, and then performing an interval difference operation gives disjoint choice guards.

In a simpler case, two **notify** guards can be made disjoint by taking the Boolean formula of the left guard $b1$ and combining it with the Boolean formula in the right guard $b2$ in the formula $\neg b1 \wedge b2$.


Finally, `left_prioritize_action σ _ a = a` handles all other cases, where the guards being prioritized are not the same, *e.g.* the case where the first is a **deposit** guard and the second is a **notify**

Figure 6 shows how to perform these operations on the Faustus AST.

Program Logics for Ledgers

Orestis Melkonian 

Input Output, London, UK

Wouter Swierstra 

Utrecht University, The Netherlands

James Chapman 

Input Output, London, UK

Abstract

Distributed ledgers nowadays manage substantial monetary funds in the form of cryptocurrencies such as Bitcoin, Ethereum, and Cardano. For such ledgers to be safe, operations that add new entries must be cryptographically sound – but it is less clear how to reason effectively about such ever-growing linear data structures. This paper demonstrates how distributed ledgers may be viewed as *computer programs*, that, when executed, transfer funds between various parties. As a result, familiar program logics, such as Hoare logic, are applied in a novel setting. Borrowing ideas from concurrent separation logic, this enables modular reasoning principles over arbitrary fragments of any ledger. All of our results have been mechanised in the Agda proof assistant.

2012 ACM Subject Classification Theory of computation → Program semantics; Theory of computation → Separation logic

Keywords and phrases blockchain, distributed ledgers, UTxO separation logic, program semantics, formal verification, Agda

Digital Object Identifier 10.4230/OASICS.FMBC.2025.10

Supplementary Material

Software (Zenodo archive): <https://doi.org/10.5281/zenodo.15097592> [22]

Software (Source Code): <https://github.com/omelkonian/hoare-ledgers> [21]

archived at `swh:1:dir:fe2bce9b8779645c5a992156ea43604432ccc496`

Acknowledgements We would like to thank Philip Wadler, Marko Doko, Murdoch J. Gabbay, Manuel M.T. Chakravarty, Brian Campbell, and James Wood for their valuable feedback.

1 Introduction

Ledger-based cryptocurrencies manage large amounts of money and record monetary transfers. On the Cardano blockchain alone, transactions valuing over 300M USD are recorded every day. The underlying blockchain that records transactions, gigabytes in size, is an ever growing linear data structure. How could we ever hope to reason about such colossal and monolithic data structures?

To illustrate this point, consider the following simplified example, giving two simple lists of transactions, also known as *ledgers*:

Alice pays Bob 5;
Carroll pays Dana 3;
Dana pays Alice 2;

Dana pays Bob 5;
Alice pays Dana 3;
Carroll pays Dana 3;

Are these transactions “the same”? Although a simple calculation shows that they have the same net effect, one of the two might fail: for instance, if Dana has less than five funds available these two behave differently. Now suppose that these transactions are interleaved with an arbitrary number of other transactions, some even involving the same accounts. Can



© Orestis Melkonian, Wouter Swierstra, and James Chapman;
licensed under Creative Commons License CC-BY 4.0

6th International Workshop on Formal Methods for Blockchains (FMBC 2025).

Editors: Diego Marmosoler and Meng Xu; Article No. 10; pp. 10:1–10:22

OpenAccess Series in Informatics



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

we still say anything about how they might behave? For many financial applications, such as fraud detection or ledger compression, it is crucial to study fragments of the ledger in isolation to ensure analyses remain computationally tractable.

This paper explores the application of program language semantics to the domain of financial ledgers, such as those underlying cryptocurrencies. Starting from simple ledger based accounts, we extend our study to cover the Unspent Transaction Outputs model (UTxO), underlying modern cryptocurrencies including Bitcoin [27] and Cardano [9]. Crucially, we explore how to employ *separation logic* to enable *effective* and *modular* reasoning about ledger-based financial transactions. Just as imperative programs mutate computer memory, financial transactions mutate bank accounts. Hoare logic and separation logic enable us to rigorously prove the correctness of computer programs. Surprisingly – as this paper demonstrates – these logics can be adapted to reason about the financial transactions stored on a ledger with the same degree of confidence. To this end, this paper makes the following novel contributions:

- First and foremost, we demonstrate how *the financial transactions stored in a ledger form a simple programming language*. We present denotational and axiomatic semantics of account-based ledgers, together with a *separation logic* that enables modular reasoning over ledger fragments (Section 2). The separation logic that arises in this context, however, turns out to be subtly different, yet strictly more general than the typical logics used to reason about computer programs.
- We show how these same semantics can be given for blockchain ledgers (Section 3), in particular ones based on the UTxO model. Separation logic, however, poses more of a challenge as the hash-based nature of UTxO adds new side conditions to the frame rule that were not necessary for account-based ledgers.
- To address this problem, we propose a novel variant of UTxO, dubbed *Abstract UTxO*. In contrast to regular UTxO, our Abstract UTxO model supports compositional reasoning using separation logic without further side conditions (Section 4). The resulting logic enables us to reason locally and safely about a limited number of transactions, sprinkled arbitrarily throughout a larger ledger.

It is important to emphasise that we do *not* study individual smart contracts or other such programs that might manipulate the ledger; the focus of this paper is the meaning of the ledger *as a whole*.

All the definitions and theorems presented in this paper have been mechanised in Agda [22]:

<https://omelkonian.github.io/hoare-ledgers/>

We use mathematical notation rather than “literate programming” style, but still provide hyperlinks to the actual mechanisation indicated by the Agda logo (🔗). The proofs themselves are typically quite simple – the hard work is in finding the definitions that make them so.

While the semantics and logics may be unsurprising to program language experts, we feel their application in a novel setting is ample reason for excitement. Just as previous work has shown how complicated financial contracts are built from simple functional combinators [30], this work aims to discover mathematical structure where none is apparent.

2 Account-based ledgers

[🔗 ValueSepExact.Main]

To start things off, we give a formal definition of the syntax and semantics of a simple account-based ledger. This illustrates one of the key ideas underlying our work: applying programming language theory in a novel domain. For the sake of simplicity, we assume a

fixed set of participants \mathcal{P} . Each participant may spend or receive *funds*. At any given point in time, we model the state of all the participants' accounts as a (finite) map, mapping each participant to their current balance:

$$S := \mathcal{P} \mapsto \mathbb{N}$$


Note that this model does now allow negative account balances; one could generalise this to any fixed bound other than zero to model overdraft.

We will treat a finite map σ as a function from keys to values for simplicity, retrieving a key k with $\sigma(k)$ and constructing a new map with anonymous λ -functions. As we saw in the introduction, a *ledger* records the history of transfers between accounts. We view such a ledger as a *program*, describing updates to the state of the accounts modelled by S . The abstract syntax of our ledger is defined as:

$$\begin{aligned} T &:= \mathcal{P} \xrightarrow{n} \mathcal{P} \\ L &:= \epsilon \mid T; L \end{aligned}$$

Each transaction T describes the transfer of funds n from one person to another; the ledger consists of a *list* of such transactions, with the most recent transaction last. Now that we have the syntax in place, we present the semantics of L in three different styles.

2.1 Denotational semantics

 ValueSepExact.Ledger

We give the denotational semantics of a ledger by mapping L to a function of type $S \rightarrow \text{Maybe } S$, executing all the transactions in the ledger starting from a given state with given account balances. The optional return type is used to model the case where a transaction fails due to insufficient funds: the result is **just** a new state after successful execution or **nothing** to signal an error.

This semantics for ledgers is straightforward to define by iterating its transactions:

$$\begin{aligned} \llbracket _ \rrbracket : L \rightarrow S \rightarrow \text{Maybe } S \\ \llbracket \epsilon \rrbracket &= \text{just} \\ \llbracket t; l \rrbracket &= d(t) \gg \llbracket l \rrbracket \end{aligned} \quad (f \gg g)(s) = \begin{cases} g(s') & \text{if } f(s) = \text{just } s' \\ \text{nothing} & \text{if } f(s) = \text{nothing} \end{cases}$$

$$d : T \rightarrow S \rightarrow \text{Maybe } S$$

$$d(p_1 \xrightarrow{n} p_2)(\sigma) = \begin{cases} \text{just } \lambda p. \begin{cases} \sigma(p) - n & \text{if } p = p_1 \neq p_2 \\ \sigma(p) + n & \text{if } p = p_2 \neq p_1 \\ \sigma(p) & \text{otherwise} \end{cases} & \text{if } \sigma(p_1) \geq n \\ \text{nothing} & \text{otherwise} \end{cases}$$


The Kleisli arrow (\gg) composes partial functions by collapsing to **nothing** when the first function fails. A transaction's semantics (d) checks the validity of each transfer and fails if not enough funds are available, otherwise updates the state accordingly. We will write “ t is valid in σ ” as a uniform way to express the validity of a transaction t with respect to a given state σ , which will become more intricate when we consider blockchain ledgers in the next section.

We formulate and prove a simple compositionality result, stating that the appending of ledgers ($++$) is mapped to the composition of their denotations.

► **Theorem 1.** *For any ledgers l_1 and l_2 , we have $\llbracket l_1 ++ l_2 \rrbracket = \llbracket l_1 \rrbracket \gg \llbracket l_2 \rrbracket$.*

This result, however, gives us only limited modularity – we still need to break a ledger into sequential pieces that we consider individually. To handle large ledgers, however, we would like to reason about *arbitrary* ledger fragments; in particular, we may be interested in an arbitrary subset of the transactions that are related to a specific smart contract in the blockchain setting.

2.2 Axiomatic semantics

 `ValueSepExact.HoareLogic`

We also define an *axiomatic semantics* for L . To do so, we define inference rules for Hoare triples of the form $\{P\} l \{Q\}$, where P and Q are predicates on our state space S .

$$\frac{}{\{P\} \epsilon \{P\}} \text{STOP} \qquad \frac{\{P\} l \{Q\}}{\{\uparrow P \circ d(t)\} t; l \{Q\}} \text{STEP}$$

The base rule dictates that executing an empty ledger leaves the state unchanged, while the inductive step rule provides the *weakest pre-condition* by viewing the transaction as a *predicate transformer* [14].

Also note the necessary operation \uparrow , lifting a predicate over S to a predicate over *Maybe* S . There are two canonical ways to achieve this lifting: the **weak** lifting that collapses to **true** when a transaction fails, and the **strong** lifting that collapses to **false** upon failure. Since we wish to observe failing transactions, we opt for the *strong* version, which we prove *sound* with respect to the denotational semantics:

► **Theorem 2.** $\{P\} l \{Q\}$ holds iff $(P(\sigma) \wedge \llbracket l \rrbracket(\sigma) = \text{just } \tau)$ implies $Q(\tau)$ for all σ and τ .

We add the typical rule for weakening/strengthening pre-/post-conditions:

$$\frac{P' \Rightarrow P \quad \{P\} l \{Q\} \quad Q \Rightarrow Q'}{\{P'\} l \{Q'\}} \text{CONSEQ}$$

The soundness theorem also allows us to derive a sequencing rule as a corollary of the equivalent statement about ledgers in the denotational semantics:

$$\frac{\{P\} l_1 \{Q\} \quad \{Q\} l_2 \{R\}}{\{P\} l_1 ++ l_2 \{R\}} \text{APP}$$

► **Remark 3.** For the rest of the paper, whenever we **axiomatize** inference rules (e.g. STOP, STEP, CONSEQ) we imply that they are at the same time proven *sound* with respect to the denotational semantics. Moreover, any subsequent **derived** inference rules (e.g. APP above) are implicitly proven using either the axioms or directly appealing to their denotational counterparts.

Example specification

Equipped with a program logic for transactions, we now formulate properties using Hoare triples and prove them in a sequential fashion akin to *equational reasoning*:

$$\{\lambda\sigma. \sigma(A) = 2\} \quad A \xrightarrow{1} B \quad \{\lambda\sigma. \sigma(A) = 1\} \quad A \xrightarrow{1} C \quad \{\lambda\sigma. \sigma(A) = 0\}$$

The above reads as follows: we start from a state where A holds 2 units of currency; then execute a transfer of one of those from A to B resulting in a state where only a single unit remains in A 's account; and we subsequently transfer the other unit to C reaching a final state where A holds no funds.

However, to prove such statements amounts to providing evidence for each Hoare triple at each step, which involves predicates over the whole state, although each transaction can only refer to two distinct participants. In the case of a more complicated state space than just a single participant, this approach is *non-compositional*, since you would need to talk about the whole state you care about in one go. This is precisely the reason we now turn our attention to *separation logic* [32].

2.3 Separation logic

 ValueSepExact.SL

Both of our denotational and axiomatic semantics rely on having the *complete* ledger at our disposal – we cannot yet use these semantics to reason about arbitrary subsets of transactions, independent of the others. To this end, we define a *separating conjunction* combining two predicates, P and Q , on our state space S . Before we do so, however, we need to consider how to combine states S . In most program language semantics, this is done by splitting the memory state (i.e. the *heap* mapping variable addresses to their value) into two disjoint parts. The separating conjunction, $P * Q$, is then defined as follows:

$$(P * Q)(\sigma) := \exists \sigma_1. \exists \sigma_2. P(\sigma_1) \wedge Q(\sigma_2) \wedge \sigma = \sigma_1 \uplus \sigma_2$$

Here, σ is the resulting heap of combining two smaller heaps σ_1 and σ_2 using the *disjoint union* operation (\uplus).

When considering financial ledgers, however, we can do better. As each transaction preserves the overall funds, we do *not* require the maps to be disjoint; instead, we divide the *funds* from both maps into two distinct parts! To do so, we begin by defining the following operation of combining ledger states by pointwise addition of their funds:

$$(\sigma_1 \oplus \sigma_2)(p) := \sigma_1(p) + \sigma_2(p)$$

Using this operation, we now define the separating conjunction of predicates as follows:

$$(P * Q)(\sigma) := \exists \sigma_1. \exists \sigma_2. P(\sigma_1) \wedge Q(\sigma_2) \wedge \sigma = \sigma_1 \oplus \sigma_2$$

The frame rule, used to introduce the separating conjunction, now becomes:

$$\frac{\{P\} l \{Q\}}{\{P * R\} l \{Q * R\}} \text{ FRAME}$$

Crucially, this version of the frame rule does not have the usual side conditions required to reason about imperative languages, namely, that the set of variables modified by l must be disjoint from the free variables mentioned by R . Intuitively, this rule is valid since transactions preserve the total amount of funds in circulation: we split off some of these funds (leaving funds that satisfy R left over), move these funds in accordance with l , and then recombine the result with the funds satisfying R .

To complete this semantics, however, we need to add a few basic rules that are currently missing. The rule for handling a single transaction is very simple indeed:


$$\frac{}{\{p_1 \mapsto n\} p_1 \xrightarrow{n} p_2 \{p_2 \mapsto n\}} \text{ SEND}$$

The precondition, $p_1 \mapsto n$, states that participant p_1 has a total of n funds (and all other participants have none). After executing this transaction, p_2 has received these n funds (and all other participants, including p_1 , have none). By itself, this rule does not seem useful – but in combination with the frame rule above, it can be used to execute a single transaction in any larger state – leaving all other funds untouched. The final two rules describe the behaviour of an entire ledger:

$$\frac{}{\{emp\} \in \{emp\}} \text{EMPTY} \qquad \frac{\{P\} l_1 \{Q\} \quad \{Q\} l_2 \{R\}}{\{P\} l_1 ++ l_2 \{R\}} \text{APP}$$

The first rule states that the empty ledger leaves the empty state unchanged; the second describes how transactions from two non-empty ledgers are run sequentially.

2.4 Concurrent separation logic


 ValueSepExact.CSL

Furthermore, we can define a (non-deterministic) interleaving operation on ledgers, $l_1 \parallel l_2$. One of the more promising observations we can make is that the familiar rule for concurrent separation logic also holds for the interleaving of two ledgers:

$$\frac{\{P_1\} l_1 \{Q_1\} \quad \{P_2\} l_2 \{Q_2\}}{\{P_1 * P_2\} l_1 \parallel l_2 \{Q_1 * Q_2\}} \text{PAR}$$

This provides a modular reasoning principle for ledgers: it allows us to focus on an arbitrary subset of the ledger’s transactions and reason about this subset in isolation. Whenever we interleave its transactions with the remainder of the ledger, any properties we have established still hold of the composite ledger. We refer the reader to Appendix A for example derivations.

3 UTxO

 UTxOErr.Main

In the coming sections, we will explore how to define similar semantics for UTxO-based blockchains. To do so, requires abandoning our previous assumption that there is a fixed set of participants, each with their own account. The UTxO model is quite different: rather than develop a model based on accounts and transactions between them, the UTxO model focuses on “unspent funds”. Such unspent funds are locked by a *validator script*. These funds can be spent by anyone, provided they can provide the *redeemer data*, that is, data mapped to **true** by the associated validator script:

$$\text{Output} := \{\text{validator} : \text{DATA} \rightarrow \mathbb{B}, \text{value} : \mathbb{N}\}$$

Typically, such a validator script might use public-private key pairs to allow access only to the person holding the private key. This model is more general than the account-based model we have studied so far: funds might be shared by different parties that each must provide their private key to unlock the corresponding funds. The overall state of the ledger is a collection of unspent transaction outputs (UTxOs) – we will make this more precise shortly, but first need to describe how transactions work in the UTxO model.

If all unspent funds are locked by a validator script – how can we possibly move funds? In the UTxO model, each transaction consumes unspent funds from *inputs*, producing new unspent *outputs*:

$$T := \{\text{inputs} : [\text{Input}], \text{outputs} : [\text{Output}]\}$$

Each input needs to refer to the unspent output being consumed *and* provide the redeemer data required to unlock the corresponding funds:

$$\begin{aligned} \text{Ref} &:= \{\text{tx} : \text{HASH}, \text{index} : \mathbb{N}\} \\ \text{Input} &:= \{\text{ref} : \text{Ref}, \text{redeemer} : \text{DATA}\} \end{aligned}$$

There is a subtle point to consider here: how should an input refer to an unspent transaction output? This is usually done by referring to the transaction's *hash*. As each transaction produces a list of unspent transaction outputs, we also require an index into this list to refer to a specific transaction output. We write $t_k^\#$ to refer to the k -th output of the transaction t . Finally, the ledger itself consists of a list of transactions – as we have seen before.

$$L := \epsilon \mid T; L$$

For the sake of clarity, we have elided some additional fields and validator arguments that do not play a significant role in our semantics:

- adding a single transaction field **forge** : \mathbb{N} to create new currency immediately gets us to Bitcoin's UTXO model [3];

$$T := \{\dots, \text{forge} : \mathbb{N}\}$$

- an additional field **datum** : $DATA$ in outputs and extending validators with additional context further brings us to the Extended UTXO model employed by Cardano [9] that supports fully expressive smart contracts;

$$Output := \{\text{validator} : Context \rightarrow DATA \rightarrow DATA \rightarrow \mathbb{B}, \text{value} : \mathbb{N}, \text{datum} : DATA\}$$

- generalising output values from \mathbb{N} to *token bundles* and including policies to control how to mint these tokens enables native tokens and multi-currency support [11, 10].

$$TokenBundle := HASH \mapsto HASH \mapsto \mathbb{N}$$

$$T := \{\dots, \text{forge} : TokenBundle, \text{mintingPolicies} : \dots\}$$

$$Output := \{\dots, \text{value} : TokenBundle\}$$

At any given point, the state records the currently unspent transaction outputs:

$$S := Ref \mapsto Output$$

We again treat finite maps as functions from keys to values; we write $k \in \sigma$ to check for membership in the map; $\sigma \setminus ks$ to remove a set of keys; $\sigma \uplus \sigma'$ for the *disjoint union* on maps.

For some examples of transactions, see Appendix B.

3.1 Denotational semantics

$[\![\bigcup^{\#} \text{UTxOErr} . \text{Ledger}]\!]$

In the previous section, a transaction could fail due to insufficient funds. Similarly in the UTXO setting, transactions are only valid under certain conditions. Given transaction t and state σ , t is valid in σ iff *all* the following criteria are met:

- **referenced outputs are unspent:**

$$\forall (i \in t.\text{inputs}). i.\text{ref} \in \sigma$$

- **there is no double spending:**

$$\forall (i, j \in t.\text{inputs}). i \neq j \rightarrow i.\text{ref} \neq j.\text{ref}$$

- **value is preserved:**

$$\sum_{i \in t.\text{inputs}} \sigma(i.\text{ref}).\text{value} = \sum_{o \in t.\text{outputs}} o.\text{value}$$

■ all inputs validate:

$$\forall(i \in t.\text{inputs}). \sigma(i.\text{ref}).\text{validator}(i.\text{redeemer}) = \text{true}$$

All other parts remain identical to the previous semantics, Hoare logic rules included, except the denotation of a single transaction: instead of updating account balances, it instead removes all previous UTxOs consumed by its inputs and then inserts new UTxOs for each output:

$$d : T \rightarrow S \rightarrow S$$

$$d(t)(\sigma) = \sigma \setminus \{i.\text{ref} \mid i \in t.\text{inputs}\} \uplus \{t_k^\# \mapsto o \mid t.\text{outputs}[k] = o\}$$

3.2 Separation logic

$[\text{UTxOErr.SL}]$

So far it has been straightforward to extend our results from the previous sections to UTxO-based blockchains: once we have the denotation of a single transaction, the semantics of a ledger is simply the composition of its constituent transactions. When we attempt to define a separation logic for the UTxO model, however, we encounter a new problem.

The UTxO model refers to existing outputs *by name* (i.e. the hash of the enclosing transaction), while the previous model for account-based ledger transferred funds directly *by value*. This allowed us to split and combine the finite maps, $\sigma_1 \oplus \sigma_2$, that associate each participant with their available funds. In the UTxO situation, however, funds are locked by a validator script and must be consumed as a whole: we cannot readily split and combine funds in the same way as we saw previously. Therefore, predicates such as $t_3^\# \mapsto v * t_3^\# \mapsto v'$ no longer make sense, since the third output of transaction t can only be spent once. Thus our separating conjunction has to be restricted only to *disjoint* fragments of the state:

$$(P * Q)(\sigma) := \exists \sigma_1. \exists \sigma_2. P(\sigma_1) \wedge Q(\sigma_2) \wedge \sigma = \sigma_1 \uplus \sigma_2$$

As a result, we have to extend the frame rule with a *disjointness* side-condition, familiar from the semantics of imperative programs that mutate memory:

$$\frac{\{P\} l \{Q\} \quad l \# R}{\{P * R\} l \{Q * R\}} \text{FRAME}$$

The condition $l \# R$ ensures all references in l are disjoint from the *support* of R , i.e. the validity of the predicate does not depend on parts of the state that the ledger mutates:

$$l \# R := \forall s. R(s) \leftrightarrow R(s \setminus \{i.\text{ref} \mid i \in l.\text{inputs}\})$$

3.3 Concurrent separation logic

$[\text{UTxOErr.CSL}]$

Similarly, the parallel rule also needs to be restricted to only *disjoint interleavings*:

$$\frac{\{P_1\} l_1 \{Q_1\} \quad \{P_2\} l_2 \{Q_2\} \quad l_1 \# P_2 \quad l_2 \# P_1}{\{P_1 * P_2\} l_1 || l_2 \{Q_1 * Q_2\}} \text{PAR}$$

This is the point in our development where we have lost the stronger *compositionality* properties that the previous semantics enjoyed. To use the frame rule to reason about UTxO-based blockchains, this side condition requires checking disjointness (see Appendix B for examples) – where previously we were free to split off arbitrary funds from the account state.

4 Abstract UTxO

 ValueSepUTxO.Main

Another way to approach the problems with a separation logic for UTxO ledgers identified in the previous section would be to tweak the UTxO model itself to make it easy to accommodate compositional reasoning techniques.

Rather than give up on UTxO entirely, we instead define a variation of UTxO where we abstract away from hash-based references and refer to unspent outputs by *value*:

$$Ref := Output$$

It is important to emphasise what changes here: rather than refer to an unspent transaction output using the transaction hash (and the index in the list of outputs it produces), each input refers directly to the *values* of the unspent outputs it consumes.

The rest of the basic definitions remain intact, except that the state of the ledger can no longer be represented by a map from references to outputs, but rather as a *bag* of outputs, since we need to keep track of duplicates which are now perfectly fine (there can be multiple outputs with the same exact value).

$$S := Bag(Output)$$

These bags, also known as *multi-sets*, can again be viewed as functions mapping outputs to quantities (\mathbb{N}), so we will reuse the notation from the previous sections; now $\sigma(k)$ returns how many times an element k occurs in bag σ . If we furthermore exploit the monoidal nature of the number of occurrences, we get access to an *overlapping union* operator that performs pointwise addition, as well as a notion of *bag inclusion*:

$$(\sigma_1 \oplus \sigma_2)(p) := \sigma_1(p) + \sigma_2(p) \quad \sigma \subseteq \tau := \forall x. \sigma(x) \leq \tau(x)$$

We call the resulting ledger model *Abstract UTxO* (AUTxO), given that it abstracts away the ordering on transaction outputs imposed by the UTxO model.

4.1 Denotational semantics

 ValueSepUTxO.Ledger

To define a denotational semantics for AUTxO, we need to revise the validity conditions that check a transaction t given a current ledger state σ , and redefine the state transition function, d . Validity of abstract transactions closely follows the criteria we set previously in Section 3.1, except that inputs now only contain a monetary value locked by a validator (i.e. they are no longer represented as unspent outputs attached to previous transactions), so we need only check that the current bag of unspent values contains at least the consumed amount, and there is no longer a requirement to check for duplicate references, since it is now perfectly sensible to have two inputs that carry the same value. Formally:

- **there are sufficient funds in σ :**

$$t.inputs \subseteq \sigma$$

- **all inputs validate:**

$$\forall (i \in t.inputs). i.ref.validator(i.redeemer) = true$$

- **value is preserved:**

$$\sum_{i \in t.inputs} i.ref.value = \sum_{o \in t.outputs} o.value$$

10:10 Program Logics for Ledgers

Notice that value preservation has become significantly simpler to formulate in this more abstract model, since we no longer need to query the value of a referenced output from the current state σ : the reference *is* the value!

The denotational semantics of a single transaction removes previously unspent transaction outputs, replacing them with the outputs of the new transaction:

$$\begin{aligned} d : T &\rightarrow S \rightarrow S \\ d(t)(\sigma) &= \sigma \setminus \{i.\text{ref} \mid i \in t.\text{inputs}\} \oplus t.\text{outputs} \end{aligned}$$

We derive the rest of the scaffolding to sequentially derive the denotation of a whole ledger exactly as before. The axiomatic semantics do not change in any way, except that they work on predicates over bags of outputs instead of maps from references to outputs.

4.2 Separation logic

 ValueSepUTxO.SL

We can finally regain modularity for our separation logic, thanks to transaction inputs in AUTxO referring to existing outputs by value. In particular, we can define separating conjunction:

$$(P * Q)(\sigma) := \exists \sigma_1. \exists \sigma_2. P(\sigma_1) \wedge Q(\sigma_2) \wedge \sigma = \sigma_1 \oplus \sigma_2$$

Notice how we utilise the monoidal composition of two bags that may overlap, regardless of whether they are disjoint or not.


The resulting inference rules are identical to the ones presented previously for account-based ledgers in Section 2, where we now use the monoidal actions on bags of values instead of the pointwise sum on finite maps.

$$\frac{\{P\} l \{Q\}}{\{P * R\} l \{Q * R\}} \text{FRAME} \qquad \frac{\{P_1\} l_1 \{Q_1\} \quad \{P_2\} l_2 \{Q_2\}}{\{P_1 * P_2\} l_1 || l_2 \{Q_1 * Q_2\}} \text{PAR}$$

In particular, the PAR rule enables us to reason about separate parts of the ledger independently. We can now prove properties at the AUTxO level in a modular fashion (see Appendix C for an example), and have confidence that they also hold in an equivalent UTxO ledger with hash references and ordered outputs.

Note that the elements of each bag are pairs of a validator function and available funds. While previously in the account-based ledger model, we used the monoidal action on the monetary funds – adding their monetary value when splitting the state into smaller parts. In the UTxO model, however, we cannot split locked funds: if the same validator locks two values v and v' , we cannot deduce that it locks $v + v'$ – a property that the simple account-based ledgers did support. We will discuss this limitation in more detail later (Section 7), but leave its resolution for future work.

4.3 Sound abstraction

 ConcreteToAbstract

The relation between AUTxO and UTxO is not yet satisfying, as we need some kind of *full abstraction* [24] result that lets us conduct compositional proofs at the *abstract* (\mathbb{A}) level which then translate to properties about an actual *concrete* (\mathbb{C}) ledger. One can informally see that all properties that do not observe the implementation details of the concrete model should be derivable from their abstract counterparts. To formalise this intuition, we first

define the abstraction of a concrete state as viewing its *range* as a bag:

$$\begin{aligned} \text{abs}^S : \mathbb{C}.S &\rightarrow \mathbb{A}.S \\ \text{abs}^S(\sigma) &= \{\sigma(k) \mid k \in \sigma\} \end{aligned}$$

We can then build up abstraction functions for *valid* transactions (abs^T) and ledgers (abs^L), where we resolve the actual outputs that references consume. Most importantly, UTxO validity is transformed into AUTxO validity, making it possible to then relate their respective denotational semantics.

► **Lemma 4.** *Given a UTxO ledger l valid in σ , applying the UTxO semantics and then abstracting the resulting state is the same as first abstracting the state and then running the AUTxO semantics on the abstracted ledger:*

$$\frac{l \text{ valid in } \sigma \quad \mathbb{C} \llbracket l \rrbracket(\sigma) = \text{just } \tau}{\mathbb{A} \llbracket \text{abs}^L(l) \rrbracket(\text{abs}^S(\sigma)) = \text{just } \text{abs}^S(\tau)}$$

Finally, we can prove soundness of our abstract model with respect to the UTxO model, at least for properties that do not observe implementation details.

► **Theorem 5.** *Given a UTxO ledger l valid in some initial concrete state σ , we can discharge a concrete Hoare triple with abstract pre-/post-conditions by proving its abstract counterpart:*

$$\frac{\mathbb{A}\{P\} \text{abs}^L(l) \{Q\} \quad l \text{ valid in } \sigma}{\mathbb{C}\{P \circ \text{abs}^S\} l \{Q \circ \text{abs}^S\}} \text{SOUNDNESS}$$

where both Hoare triples have been implicitly instantiated to the state σ that is universally quantified at the outermost level.

This means it is *sound* to conduct modular proofs on the abstract level; the equivalent statement on concrete ledgers will also hold. Note that our abstract model is not *complete*, since we can only cover abstract state predicates of the form $P \circ \text{abs}^S$, thus we cannot hope to prove a *full abstraction* result. We feel that this will not be problematic in practice: these predicates mention implementation details that arguably *should* be kept abstract.

► **Remark 6.** While making this formal connection to UTxO is important to make sure our results readily transfer to existing blockchains, there is still something to be said about AUTxO in isolation, as an alternative underlying model for new blockchains. From the pragmatic lens of blockchain validation, AUTxO seems to allow far more liberal transaction sequences than UTxO, where you would need to re-submit transactions to resolve conflicts. This contention bottleneck heavily influences how many transactions can be validated in parallel, hence a blockchain built on AUTxO might allow higher transaction throughput. Although an experimental validation of this claim still remains to be done, we note that there have been some initial experiments that explore similar relaxations of the UTxO model [25], as employed in the IOTA distributed ledger [26].

5 Perspectives

Why care about semantics? Oftentimes blockchains are designed with cryptographic security guarantees in mind; the intended ledger semantics is usually clear, but as is often the case, the devil is in the details. This shows up when considering separation logic for ledgers built on the UTxO model, where cryptographic details about hashing and the model's implementation

leak into the logic. Exposing the underlying mathematics, as our AUTxO model does, nails down the exact behaviour once and for all. We believe there are numerous applications that would be difficult, if not impossible, to reason about without a clearly specified formal semantics. This section sketches several such directions.

Formal verification of smart contracts

So far we have only tackled the verification of individual transactions, but there is nothing preventing us from reasoning about resources in the more elaborate setting of *smart contracts*.

Given the extensions to the UTxO model outlined in Section 3 to allow for fully-fledged smart contracts to be expressed in validator scripts, it is crucial to observe that such contracts will again manifest as transaction outputs holding a certain amount of funds. In other words, smart contracts would appear as another kind of participant in Hoare triples, and we can reason about its resources in the usual manner. Moreover, smart contracts would now appear in a *sequence* of related transactions, while possibly also interacting with other non-contract resources; the modularity of our Hoare-style framework gives us the ability to focus on exactly the subset of transactions we are interested in.

One immediate application of this method to EUTxO smart contracts would be to prevent the common issue of **double satisfaction** [38], arising when a *single* input resource is used to satisfy *multiple* constraints coming from different validators/scripts. Within our framework, the resource in question would be precisely characterised in a Hoare triple, either to rule out double satisfaction or exhibit that it occurs (in case it was deliberate).

Ledger compression

Another application of this work would be in *ledger compression*: denotation semantics give us a natural algorithm for minimising the ledger, while concurrent separation logic lets us reason under the substitution of the original ledger with the compressed one.

Inspired by the technique of *normalisation by evaluation* [5], we proceed as follows for a given ledger l that we wish to compress:

1. Compute the denotational semantics $\llbracket l \rrbracket$, i.e. a function that transforms states.
2. Read back a *minimal* ledger by observing the net change to participant accounts. The resulting compressed ledger is guaranteed to have the same semantics as the ledger we started with, therefore we can instead prove a Hoare triple for the smaller ledger.
3. Complete our reasoning by utilising the PAR rule to embed our results from step (2) in the context of the whole ledger L , which is expected to be way larger than l .

While the above works for the account-based ledger of Section 2, things become trickier in the UTxO case: the hash references appearing in the resulting state will always necessarily differ after compression. However, if we ignore the hashes and instead consider the values that they refer to, we would be able to see that the associated values indeed remain identical. In other words, this application requires reasoning *modulo hashes*, which is exactly what the AUTxO model of Section 4 provides.

It is also worth mentioning that the same technique can be used to detect fraud when “mixers” obfuscate the provenance of certain funds by moving them around between different accounts. Since the net effect of such “rings” would be zero, ledger compression would then optimise them away, enabling us to read off the fraudulent addresses that were involved.

6 Related work

Blockchain theory

The entire line of research on UTxO-based ledgers starts from Bitcoin [27, 3, 4], later extended in the Cardano blockchain to *Extended UTxO* (EUTxO) [39] so as to enable the full expressivity of smart contracts. Thankfully, there are mechanised formalisations for the meta-theory of both Bitcoin [35] and EUTxO [9, 10], all of which however suffer from a monolithic approach, where the only reasoning provided is based on induction over the whole history of the ledger. We believe that the approach present here does not contradict in any way with the basic assumptions in these formulations; we expect it can be readily deployed in each respective setting. One experiment for ledger modularity in the EUTxO setting [23] led to the inevitable non-compositional notion of separation we addressed here.

On the Bitcoin side, there is a mechanised program logic for reasoning about Bitcoin’s script language [1] based on *predicate transformer* semantics [14]; the striking similarity with our work lies in the use of weakest preconditions to model access control, which is essentially what we use to define the STEP rule for our Hoare logic in Section 2.

Alternative approaches to solving the modularity problem include the algebraic model of *Idealised UTxO* [16] where ledgers are generalised to *ledger chunks* with open-ended inputs rather than an inductive structure and naming is handled using *nominal techniques* [15], as well as the categorical treatment of Nester’s material history [28, 29] where one reasons about resources and ownership in the intuitive graphical language of *symmetric monoidal categories* [33, 12].

In the non-UTxO setting, where the underlying ledger follows the account-based variant of models led by Ethereum, an approach based on ownership influenced by the program logic literature is used for implementing *sharding* – a technique for scaling up transaction validation across multiple nodes – for the Zilliqa blockchain [31].

Concurrency theory

Analogies between the study of blockchains and classic concurrent or distributed computing have already been noted by experts in the latter that subsequently became involved in blockchain research [18, 34].

One particular separation logic in existing work bears close resemblance to the one developed in this paper, namely that of *fractional permissions* [6, 13] for handling partial ownership of resources. Similarly to our work, separating conjunction does not enforce disjointness but admits some level of overlap, in this case used to model scenarios in parallel programming with many readers and a single writer, for instance.

Last but not least, we note our initial inspiration from previous work that applied the idea of separation logic on something other than computer programs mutating memory, namely in the domain of version control systems [36].

Type Theory

The resource-oriented nature of our logics also echoes efforts in type theories that track resource usage in some way or another, for instance *Quantitative Type Theory* [20, 2], currently supported by the Idris programming language [7].

Blockchains have to do with monetary resources above everything else, thus provide a natural setting to apply these resource-oriented frameworks, and indeed there is already research supporting this in the context of the Tezos blockchain [17].

7 Future work

Decompositionality

One aspect that fails to translate to the UTxO setting is the treatment of separated conjunctions as arithmetic formulas, where equivalences such as $A \mapsto 2 \approx A \mapsto 1 * A \mapsto 1$ hold by definition. We can refer to this property as *decompositionality*, since it lets us automatically decompose a large resource into its constituent parts.

This is simply not true in the UTxO model, as noted in Section 4.2, since we still need to consume previous outputs as a whole, whose funds are predetermined by the enclosing transaction. However, we could get around this by *silently* inserting transactions that perform the necessary split/merge operations, thus allowing us to reason at an even more abstract level *modulo* transactions that merely redistribute funds. Accounting for such silent steps in the (A)UTxO model is a topic for further work.

Connection with existing separation logics

Although our approach draws heavily from the rich literature of separation logic in programming languages, we have not yet made a formal connection with our definitions and various notions of separation. One way to accomplish that is to instantiate an existing framework that supports various kinds of separation logics. A suitable candidate for that would be *Abstract Separation Logic* [8], where we could prove that the various ledger states across our development obey the interface and corresponding laws of *separation algebras*.

A more practically oriented course of action would be to directly implement our proposal in the Iris framework [19] which supports a wide variety of separation logics in the Rocq proof assistant [37]. Given how extensible Iris is and the relative simplicity of our program logics, the transliteration of our Agda formalisation to Iris should be straightforward and quickly give us a practical verification tool.

8 Conclusion

We have presented a compositional approach to reasoning about UTxO ledgers, made possible by exploiting the analogy between programs mutating memory and transactions transferring funds between accounts. The key methodological insight is that the ledger can be viewed as a programming language, thus opening up the possibility of developing program logics to reason about (sequences of) transactions. We have demonstrated how ideas from separation logic in particular provide the modularity principle to reason about ledger fragments independently of one another.

In the future, this work may lay the foundations for scaling up verification of complex UTxO-based smart contracts, or even offer multiple program logics depending on the desired level of modularity and detail. Reasoning about monolithic ledgers cannot scale without modular reasoning principles – this paper presents a first step in that direction.

References


- 1 Fahad F. Alhabardi, Arnold Beckmann, Bogdan Lazar, and Anton Setzer. Verification of Bitcoin script in Agda using weakest preconditions for access control. In Henning Basold, Jesper Cockx, and Silvia Ghilezan, editors, *27th International Conference on Types for Proofs and Programs, TYPES 2021, June 14-18, 2021, Leiden, The Netherlands (Virtual Conference)*, volume 239 of *LIPICs*, pages 1:1–1:25. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.TYPES.2021.1.

- 2 Robert Atkey. Syntax and semantics of Quantitative Type Theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 56–65. ACM, 2018. doi:10.1145/3209108.3209189.
- 3 Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. A formal model of Bitcoin transactions. In Sarah Meiklejohn and Kazue Sako, editors, *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers*, volume 10957 of *Lecture Notes in Computer Science*, pages 541–560. Springer, 2018. doi:10.1007/978-3-662-58387-6_29.
- 4 Massimo Bartoletti and Roberto Zunino. Formal models of Bitcoin contracts: A survey. *Frontiers Blockchain*, 2:8, 2019. doi:10.3389/fbloc.2019.00008.
- 5 Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*, pages 203–211. IEEE Computer Society, 1991. doi:10.1109/LICS.1991.151645.
- 6 John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003. doi:10.1007/3-540-44898-5_4.
- 7 Edwin C. Brady. Idris 2: Quantitative Type Theory in practice. In Anders Möller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 9:1–9:26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ECOOP.2021.9.
- 8 Cristiano Calcagno, Peter W. O’Hearn, and Hongseok Yang. Local action and Abstract Separation Logic. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, pages 366–378. IEEE Computer Society, 2007. doi:10.1109/LICS.2007.30.
- 9 Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The Extended UTXO model. In Matthew Bernhard, Andrea Bracciali, L. Jean Camp, Shin’ichiro Matsuo, Alana Maurushat, Peter B. Rønne, and Massimiliano Sala, editors, *Financial Cryptography and Data Security - FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers*, volume 12063 of *Lecture Notes in Computer Science*, pages 525–539. Springer, 2020. doi:10.1007/978-3-030-54455-3_37.
- 10 Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann Müller, Michael Peyton Jones, Polina Vinogradova, and Philip Wadler. Native custom tokens in the Extended UTXO model. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III*, volume 12478 of *Lecture Notes in Computer Science*, pages 89–111. Springer, 2020. doi:10.1007/978-3-030-61467-6_7.
- 11 Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann Müller, Michael Peyton Jones, Polina Vinogradova, Philip Wadler, and Joachim Zahnentferner. UTXO_{ma}: UTXO with multi-asset support. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III*, volume 12478 of *Lecture Notes in Computer Science*, pages 112–130. Springer, 2020. doi:10.1007/978-3-030-61467-6_8.
- 12 Bob Coecke, Tobias Fritz, and Robert W. Spekkens. A mathematical theory of resources. *Inf. Comput.*, 250:59–86, 2016. doi:10.1016/j.ic.2016.02.008.

- 13 Thibault Dardinier, Peter Müller, and Alexander J. Summers. Fractional resources in unbounded separation logic. *Proc. ACM Program. Lang.*, 6(OOPSLA2):1066–1092, 2022. doi:10.1145/3563326.
- 14 Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975. doi:10.1145/360933.360975.
- 15 Murdoch Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects Comput.*, 13(3-5):341–363, 2002. doi:10.1007/s001650200016.
- 16 Murdoch James Gabbay. Algebras of UTxO blockchains. *Math. Struct. Comput. Sci.*, 31(9):1034–1089, 2021. doi:10.1017/S0960129521000438.
- 17 Christopher Goes. Compiling Quantitative Type Theory to Michelson for compile-time verification and run-time efficiency in juvix. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III*, volume 12478 of *Lecture Notes in Computer Science*, pages 146–160. Springer, 2020. doi:10.1007/978-3-030-61467-6_10.
- 18 Maurice Herlihy. Blockchains from a distributed computing perspective. *Commun. ACM*, 62(2):78–85, 2019. doi:10.1145/3209623.
- 19 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order Concurrent Separation Logic. *J. Funct. Program.*, 28:e20, 2018. doi:10.1017/S0956796818000151.
- 20 Conor McBride. I got plenty o’ nuttin’. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 207–233. Springer, 2016. doi:10.1007/978-3-319-30936-1_12.
- 21 Orestis Melkonian. omelkonian/hoare-ledgers. Software, swhId: swh:1:dir:fe2bce9b8779645c5a992156ea43604432ccc496 (visited on 2025-04-14). URL: <https://github.com/omelkonian/hoare-ledgers>, doi:10.4230/artifacts.23004.
- 22 Orestis Melkonian. Agda formalisation for “Program logics for ledgers”. <https://github.com/omelkonian/hoare-ledgers>, March 2025. doi:10.5281/zenodo.15097592.
- 23 Orestis Melkonian, Wouter Swierstra, and Manuel M. T. Chakravarty. Formal investigation of the Extended UTxO model. In *4th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe)*, 2019. URL: <https://omelkonian.github.io/data/publications/formal-utxo.pdf>.
- 24 Robin Milner. Fully abstract models of typed λ -calculi. *Theor. Comput. Sci.*, 4(1):1–22, 1977. doi:10.1016/0304-3975(77)90053-6.
- 25 Sebastian Müller, Andreas Penzkofer, Nikita Polyanskii, Jonas Theis, William Sanders, and Hans Moog. Reality-based UTXO ledger. *CoRR*, abs/2205.01345, 2022. doi:10.48550/arXiv.2205.01345.
- 26 Sebastian Müller, Andreas Penzkofer, Nikita Polyanskii, Jonas Theis, William Sanders, and Hans Moog. Tangle 2.0 leaderless nakamoto consensus on the heaviest DAG. *IEEE Access*, 10:105807–105842, 2022. doi:10.1109/ACCESS.2022.3211422.
- 27 S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. <https://bitcoin.org/en/bitcoin-paper>, October 2008.
- 28 Chad Nester. A foundation for ledger structures. In Emmanuelle Anceaume, Christophe Bisière, Matthieu Bouvard, Quentin Bramas, and Catherine Casamatta, editors, *2nd International Conference on Blockchain Economics, Security and Protocols, Tokenomics 2020, October 26-27, 2020, Toulouse, France*, volume 82 of *OASICS*, pages 7:1–7:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/OASICS.Tokenomics.2020.7.
- 29 Chad Nester. The structure of concurrent process histories. In Ferruccio Damiani and Ornella Dardha, editors, *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021*,

- Proceedings*, volume 12717 of *Lecture Notes in Computer Science*, pages 209–224. Springer, 2021. doi:10.1007/978-3-030-78142-2_13.
- 30 Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering (functional pearl). *ACM SIGPLAN Notices*, 35(9):280–292, 2000. doi:10.1145/351240.351267.
 - 31 George Pirlea, Amrit Kumar, and Ilya Sergey. Practical smart contract sharding with ownership and commutativity analysis. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 1327–1341. ACM, 2021. doi:10.1145/3453483.3454112.
 - 32 John C. Reynolds. Separation Logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002. doi:10.1109/LICS.2002.1029817.
 - 33 Peter Selinger. A survey of graphical languages for monoidal categories. *New structures for physics*, pages 289–355, 2011.
 - 34 Ilya Sergey and Aquinas Hobor. A concurrent perspective on smart contracts. In Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y. A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson, editors, *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers*, volume 10323 of *Lecture Notes in Computer Science*, pages 478–493. Springer, 2017. doi:10.1007/978-3-319-70278-0_30.
 - 35 Anton Setzer. Modelling Bitcoin in Agda. *CoRR*, abs/1804.06398, 2018. arXiv:1804.06398.
 - 36 Wouter Swierstra and Andres Löb. The semantics of version control. In Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz, editors, *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014*, pages 43–54. ACM, 2014. doi:10.1145/2661136.2661137.
 - 37 The Coq Development Team. The Coq proof assistant (*renamed to Rocq*), September 2024. doi:10.5281/zenodo.14542673.
 - 38 Polina Vinogradova and Orestis Melkonian. Message-passing in the Extended UTxO ledger. In *Financial Cryptography and Data Security. FC 2024 International Workshops: Voting, DeFI, WTSC, CoDecFin, Willemstad, Curaçao, March 4–8, 2024, Revised Selected Papers*, pages 150–169, Berlin, Heidelberg, 2024. Springer-Verlag. doi:10.1007/978-3-031-69231-4_11.
 - 39 Joachim Zahnentferner. An abstract model of UTxO-based cryptocurrencies with scripts. *IACR Cryptol. ePrint Arch.*, page 469, 2018. URL: <https://eprint.iacr.org/2018/469>.

A Account-based examples

 `ValueSepExact.Example`

Example derivation using FRAME

We now introduce an example derivation that will act as a running example across the various logics we will develop throughout the paper, in order to demonstrate the relative strengths and weaknesses of each approach.

We will have two transactions between participants *A* and *B* exchanging a single unit of currency back and forth, interleaved with another two transactions of the same form but now between different participants *C* and *D*. Overall, this set of transaction leave the state of account balances unchanged after execution.

Apart from the aforementioned basic rules, we will also make use of the fact that separating conjunction is *symmetric*, in the form of the following derived rule:

10:18 Program Logics for Ledgers

$$\frac{}{\{P * Q\} \approx \{Q * P\}} \text{ SWAP}$$

The FRAME rule lets us focus on a small part of a larger separating conjunction and apply the rule locally only on the part of the state that concerns the two participants of the transaction at hand:

$$\begin{array}{l}
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} \\
\quad A \xrightarrow{1} B \quad \quad \quad \vdash \text{FRAME}(C \mapsto 0 * D \mapsto 1, \text{SEND}) \\
\{A \mapsto 0 * B \mapsto 1 * C \mapsto 0 * D \mapsto 1\} \\
\quad \approx \\
\{C \mapsto 0 * D \mapsto 1 * A \mapsto 0 * B \mapsto 1\} \\
\quad D \xrightarrow{1} C \quad \quad \quad \vdash \text{FRAME}(A \mapsto 0 * B \mapsto 1, \text{SEND} \circ \text{SWAP}) \\
\{C \mapsto 1 * D \mapsto 0 * A \mapsto 0 * B \mapsto 1\} \\
\quad \approx \\
\{A \mapsto 0 * B \mapsto 1 * C \mapsto 1 * D \mapsto 0\} \\
\quad B \xrightarrow{1} A \quad \quad \quad \vdash \text{FRAME}(C \mapsto 1 * D \mapsto 0, \text{SEND} \circ \text{SWAP}) \\
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 1 * D \mapsto 0\} \\
\quad \approx \\
\{C \mapsto 1 * D \mapsto 0 * A \mapsto 1 * B \mapsto 0\} \\
\quad C \xrightarrow{1} D \quad \quad \quad \vdash \text{FRAME}(A \mapsto 1 * B \mapsto 0, \text{SEND}) \\
\{C \mapsto 0 * D \mapsto 1 * A \mapsto 1 * B \mapsto 0\} \\
\quad \approx \\
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} \quad \blacktriangleleft
\end{array}$$


Example derivation using PAR

Notice how in the previous example the first and third transaction only involve A and B , while the other two only involve C and D . That is why we can do better using the PAR rule, where we assemble a compositional proof from smaller proofs:

$$\begin{array}{l}
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} \\
(A \xrightarrow{1} B; B \xrightarrow{1} A) \parallel (D \xrightarrow{1} C; C \xrightarrow{1} D) \\
\quad \ni (A \xrightarrow{1} B; D \xrightarrow{1} C; B \xrightarrow{1} A; C \xrightarrow{1} D) \quad \quad \quad \vdash \text{PAR}(H^{AB}, H^{CD}) \\
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} \quad \blacktriangleleft
\end{array}$$

where

$ \begin{array}{l} H^{AB} := \\ \{A \mapsto 1 * B \mapsto 0\} \\ \quad A \xrightarrow{1} B \quad \quad \quad \vdash \text{SEND} \\ \{A \mapsto 0 * B \mapsto 1\} \\ \quad B \xrightarrow{1} A \quad \quad \quad \vdash \text{SEND} \circ \text{SWAP} \\ \{A \mapsto 1 * B \mapsto 0\} \quad \blacktriangleleft \end{array} $	$ \begin{array}{l} H^{CD} := \\ \{C \mapsto 0 * D \mapsto 1\} \\ \quad D \xrightarrow{1} C \quad \quad \quad \vdash \text{SEND} \circ \text{SWAP} \\ \{C \mapsto 1 * D \mapsto 0\} \\ \quad C \xrightarrow{1} D \quad \quad \quad \vdash \text{SEND} \\ \{C \mapsto 0 * D \mapsto 1\} \quad \blacktriangleleft \end{array} $
--	--

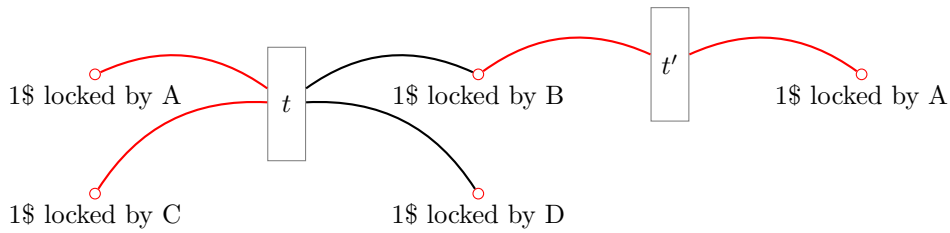
B UTxO examples[ UTxOErr.Example]**Example transaction**

Let us define an example transaction in the UTxO model. First, we associate the notion of a participant – which does not inherently exist in the UTxO model – with a validator script that restricts access to the funds solely to said participant.¹

Therefore, an output of the form “1\$ locked by A” carries a single unit of currency that only A can unlock. Moreover, a single transaction can pack together multiple inputs and outputs, thus immediately performing exchange between multiple “participants”.

Given a starting state where participants A and C hold a single unit of currency (in the form of two unspent outputs assumed to already exist in previous transactions), a transaction t can transfer these funds to another set of participants B and D, and another transaction t' give B's funds back to A.

This can be compactly depicted as a *directed acyclic graph*, whose *left fringe* contains dangling outputs that comprise the current state of unspent outputs and the *right fringe* corresponds to the resulting state, commonly referred to as the *UTxO set*:



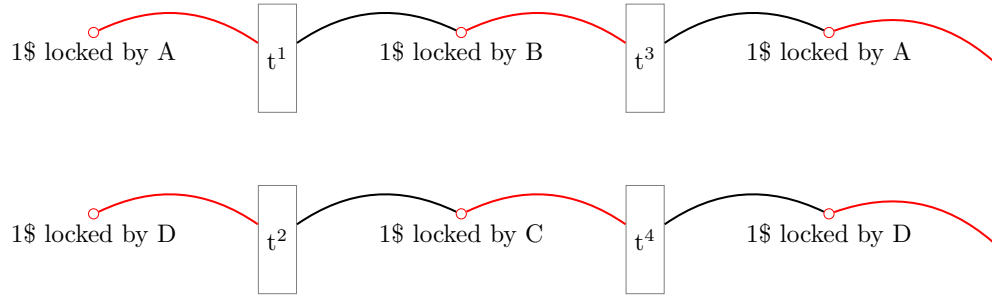
Note that the actual transaction fields are omitted in the node in the graph, since these can be easily deduced from the incoming and outgoing edges. Therefore, the above transactions denote a transition from the source state $\{ta_i^\# \mapsto 1\$ \text{ locked by A}, tc_j^\# \mapsto 1\$ \text{ locked by C}\}$ (left fringe) to the resulting state $\{t'_0^\# \mapsto 1\$ \text{ locked by A}, t'_1^\# \mapsto 1\$ \text{ locked by D}\}$ (right fringe), assuming previous outputs in ta and tc holding the initial funds for A and C.

Example derivation using FRAME

Back to our running example, we can prove similar derivations for UTxO-based ledgers, although our predicates now have to also include references to previous transactions. We denote singleton predicates by $t_i \mapsto v \text{ at } p$, where we require a single UTxO to be unspent in the i -th output of transaction t , holding a value v locked by validator function p .

¹ This could be achieved by naively having the redeemer be a password only known to the participant, or via a public-key approach where the validator script verifies a signature with respect to the participant's private key.

10:20 Program Logics for Ledgers



$$\begin{array}{l}
 \{t_0^0 \mapsto 1 \text{ at } A * t_1^0 \mapsto 1 \text{ at } D\} \\
 \quad t^1 \\
 \{t_0^1 \mapsto 1 \text{ at } B * t_1^0 \mapsto 1 \text{ at } D\} \\
 \quad \approx \\
 \{t_1^0 \mapsto 1 \text{ at } D * t_0^1 \mapsto 1 \text{ at } B\} \\
 \quad t^2 \\
 \{t_0^2 \mapsto 1 \text{ at } C * t_0^1 \mapsto 1 \text{ at } B\} \\
 \quad \approx \\
 \{t_0^1 \mapsto 1 \text{ at } B * t_0^2 \mapsto 1 \text{ at } C\} \\
 \quad t^3 \\
 \{t_0^3 \mapsto 1 \text{ at } A * t_0^2 \mapsto 1 \text{ at } C\} \\
 \quad \approx \\
 \{t_0^2 \mapsto 1 \text{ at } C * t_0^3 \mapsto 1 \text{ at } A\} \\
 \quad t^4 \\
 \{t_0^4 \mapsto 1 \text{ at } D * t_0^3 \mapsto 1 \text{ at } A\} \\
 \quad \approx \\
 \{t_0^3 \mapsto 1 \text{ at } A * t_0^4 \mapsto 1 \text{ at } D\}
 \end{array}
 \quad
 \begin{array}{l}
 \vdash \text{FRAME}(t_1^0 \mapsto 1 \text{ at } D, \dots, \text{SEND}) \\
 \\
 \vdash \text{FRAME}(t_0^1 \mapsto 1 \text{ at } B, \dots, \text{SEND}) \\
 \\
 \vdash \text{FRAME}(t_0^2 \mapsto 1 \text{ at } C, \dots, \text{SEND}) \\
 \\
 \vdash \text{FRAME}(t_0^3 \mapsto 1 \text{ at } A, \dots, \text{SEND})
 \end{array}
 \quad \blacktriangleleft$$

Note that this derivation now requires additional proof obligations, marked with \dots , ensuring the disjointness of relevant transactions.

Example derivation using PAR


The PAR can slightly improve the situation by composing smaller proofs, but is no longer a scalable solution since we still need to provide evidence that the interleaved ledgers are disjoint:

$$\begin{array}{l}
 \{t_0^0 \mapsto 1 \text{ at } A * t_1^0 \mapsto 1 \text{ at } D\} \\
 \quad t^1 \dots t^4 \\
 \{t_0^3 \mapsto 1 \text{ at } A * t_0^4 \mapsto 1 \text{ at } D\}
 \end{array}
 \quad
 \vdash \text{PAR}(\dots, H^{AB}, H^{CD})
 \quad \blacktriangleleft$$

where

$$\begin{array}{c|c}
H^{AB} := & H^{CD} := \\
\begin{array}{l}
\{t_0^0 \mapsto 1 \text{ at } A\} \\
t^1 \quad \quad \quad \vdash \text{SEND} \\
\{t_0^1 \mapsto 1 \text{ at } B\} \\
t^3 \quad \quad \quad \vdash \text{SEND} \\
\{t_0^3 \mapsto 1 \text{ at } A\} \quad \blacktriangleleft
\end{array} &
\begin{array}{l}
\{t_1^0 \mapsto 1 \text{ at } D\} \\
t^2 \quad \quad \quad \vdash \text{SEND} \\
\{t_0^2 \mapsto 1 \text{ at } C\} \\
t^4 \quad \quad \quad \vdash \text{SEND} \\
\{t_0^4 \mapsto 1 \text{ at } D\} \quad \blacktriangleleft
\end{array}
\end{array}$$

C AUTxO examples

 ValueSepUTxO.Example

As was the case for UTxO, we consider validator scripts as a replacement for participant identifiers A, B, C, D , assuming transactions $t_1 \dots t_4$ that have the corresponding structure that enacts the transfers we defined in the initial non-blockchain example.

Unsurprisingly, the Hoare conditions remain identical and only the enclosed transactions change from the initial proof on account-based ledgers (Appendix A):

$$\begin{array}{l}
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} \\
t^1 \quad \quad \quad \vdash \text{FRAME}(C \mapsto 0 * D \mapsto 1, \text{SEND}) \\
\{A \mapsto 0 * B \mapsto 1 * C \mapsto 0 * D \mapsto 1\} \\
\approx \\
\{C \mapsto 0 * D \mapsto 1 * A \mapsto 0 * B \mapsto 1\} \\
t^2 \quad \quad \quad \vdash \text{FRAME}(A \mapsto 0 * B \mapsto 1, \text{SEND}) \\
\{C \mapsto 1 * D \mapsto 0 * A \mapsto 0 * B \mapsto 1\} \\
\approx \\
\{A \mapsto 0 * B \mapsto 1 * C \mapsto 1 * D \mapsto 0\} \\
t^3 \quad \quad \quad \vdash \text{FRAME}(C \mapsto 1 * D \mapsto 0, \text{SEND}) \\
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 1 * D \mapsto 0\} \\
\approx \\
\{C \mapsto 1 * D \mapsto 0 * A \mapsto 1 * B \mapsto 0\} \\
t^4 \quad \quad \quad \vdash \text{FRAME}(A \mapsto 1 * B \mapsto 0, \text{SEND}) \\
\{C \mapsto 0 * D \mapsto 1 * A \mapsto 1 * B \mapsto 0\} \\
\approx \\
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} \quad \blacktriangleleft
\end{array}$$

Most importantly, we no longer need to provide disjointness proofs as in the UTxO case.

We finally demonstrate how we have regained compositionality in the AUTxO setting:

$$\begin{array}{l}
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} \\
t^1 \dots t^4 \quad \quad \quad \vdash \text{PAR}(H^{AB}, H^{CD}) \\
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} \quad \blacktriangleleft
\end{array}$$

where

10:22 Program Logics for Ledgers

$H^{AB} :=$

$$\begin{array}{l} \{A \mapsto 1 * B \mapsto 0\} \\ \quad t^1 \quad \quad \quad \dashv \text{SEND} \\ \{A \mapsto 0 * B \mapsto 1\} \\ \quad t^3 \quad \quad \quad \dashv \text{SEND} \\ \{A \mapsto 1 * B \mapsto 0\} \quad \blacktriangleleft \end{array}$$

$H^{CD} :=$

$$\begin{array}{l} \{C \mapsto 0 * D \mapsto 1\} \\ \quad t^2 \quad \quad \quad \dashv \text{SEND} \\ \{C \mapsto 1 * D \mapsto 0\} \\ \quad t^4 \quad \quad \quad \dashv \text{SEND} \\ \{C \mapsto 0 * D \mapsto 1\} \quad \blacktriangleleft \end{array}$$

Formally Specifying Contract Optimizations with Bisimulations in Coq

Derek Sorensen  

Department of Computer Science and Technology, University of Cambridge, UK

Abstract

The efficacy of formal verification of smart contracts depends on being able to correctly specify and carry out the verification of optimized code. However, code optimized for performance is rarely optimized for intelligibility, which can make formally verifying optimized code difficult and costly. To address this issue, we present a formal tool for reasoning about an optimized contract in terms of its reference implementation. This tool reduces the work of formally verifying an optimized contract to proving behavioral equivalence to the reference implementation.

2012 ACM Subject Classification Theory of computation → Program verification

Keywords and phrases smart contract verification, formal methods, interactive theorem prover, smart contract upgradeability

Digital Object Identifier 10.4230/OASICS.FMBC.2025.11

Supplementary Material *Software (Source code)*: <https://github.com/dhsorens/FinCert/tree/FMBC-25> [13]; archived at `swb:1:dir:4373edebe9d83da3ce0c5452513515f60c7173aa`

1 Introduction

The efficacy of formal verification to prevent actual, critical contract vulnerabilities depends on the feasibility of applying formal verification to deployable contract code. However, deployable code is typically optimized for performance, which typically makes it more difficult to reason about formally [3]. Code highly optimized for performance thus risks vulnerability due to the difficulty of formal reasoning, while code written for ease of formal reasoning may not be efficient enough for the resource-scarce environment of smart contracts. Ideally, we would reason about contracts in a state optimized for formal reasoning while still deploying them in a state optimized for efficiency and gas consumption.

What is needed is a formal tool that enables us to reason about code in a format optimized for intelligibility, design and formal reasoning, whose results can be applied to a highly optimized and equivalent version of that code. As it stands no such framework exists for smart contracts. To mitigate this we introduce a formal framework of extensional equivalence between smart contracts in Coq, called *contract isomorphisms*. These equivalences will allow us to use a reference implementation as a specification of an optimized contract, as well as to port proofs between contracts that can be proved to be bisimilar.

We proceed as follows. In Section 2 we discuss related work. In Section 3 we define contract isomorphisms, our notion of formal, extensional equivalence which implements a bisimulation of contracts. In Section 4 we show that our definition of contract isomorphisms induces a strong form of equivalence between contracts, a *trace equivalence*. In Section 5, we give an example of a contract formally specified by equivalence to an existing contract and port proofs over a bisimulation. In Section 6 we conclude.

2 Related Work

Bisimulations are a core component of theoretical computer science. They primarily denote an equivalence of state transition systems [9, 12]. They are, for example, central in the study of process algebras, which rely on a notion of equivalence between processes in order to reason algebraically about the behavior of concurrent systems.



© Derek Sorensen;

licensed under Creative Commons License CC-BY 4.0

6th International Workshop on Formal Methods for Blockchains (FMBC 2025).

Editors: Diego Marmosier and Meng Xu; Article No. 11; pp. 11:1–11:13

OpenAccess Series in Informatics



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

One critical role that bisimulations play is in equivalence checking [4, 5]. Equivalence checking is an approach to formal verification that consists in proving that two programs or models are related modulo some equivalence relation, or that one is included in the other modulo some preorder relation [4]. In this case, one uses a bisimulation to prove that a particular program meets its specification, where the specification is defined not in prose but as a program. Areas of formal reasoning and logic, including Hennessy-Milner logic, treat bisimulations as full equality and cannot distinguish between bisimilar processes [8, 9].

To our knowledge none of these techniques have been applied to smart contracts, but one can imagine that with a sufficient notion of contract bisimulation, we can mimic this process and use a contract formally verified and optimized for formal reasoning as a specification for a contract optimized for deployment. Proving the optimized contract correct then consists of producing a contract bisimulation.

One could also conceive of porting proofs over such an equivalence of contracts, *e.g.* in [2, 3, 15]. The strategy of porting proofs over equivalences is used in formal verification elsewhere. For example, work by Ringer *et al.* uses type equivalences to efficiently reuse proofs when updating a formally verified program [10]; work by Cohen *et al.* [2, 3] uses refinement types to optimize code in a proof-invariant way. Our work here is in a similar spirit, and may be applicable to future version of this work, but our equivalence in question is a contract bisimulation instead of a type equivalence.

Previous work has used bisimulations to encode the notion of correct implementation on a UTXO-based blockchain [6], but to our knowledge the work here is the first application of bisimulations as a tool for formally verifying optimized contracts. We build off of previous work in Coq which introduced the notion of a *contract morphism*, the key tool used to construct contract bisimulations [14]. Our work here is a special use case of that theoretical tool. All of our work is built in ConCert [1], a Coq-based formal verification tool with verified extraction to high-level smart contract languages including Tezos’s LIGO and Concordium’s Oak.

3 Contract Isomorphisms

The fundamental contribution of this paper is a formal mechanism for proving equivalence (bisimilarity) between smart contracts in Coq, to be used in formal specification and verification. To present this mechanism, we first give a theoretical definition of contract isomorphisms as bisimulations between contracts (Section 3.1), moving onto the details of an implementation in Coq (Section 3.2).

3.1 Bisimilarity

Bisimilarity is a stable and natural concept that describes equivalence between processes [6, 11, 16]. A standard definition of bisimulation for labelled transition systems is as follows.

► **Definition 1 (Bisimulation).** Consider a labelled transition system $(S, \Lambda, \rightarrow)$, where S is a set of states, Λ is a set of labels, and \rightarrow is a set of labelled transitions (a subset of $S \times \Lambda \times S$). A bisimulation is a binary relation $R \subseteq S \times S$ such that for every pair of states $(p, q) \in R$ and labels $\alpha, \beta \in \Lambda$,

- if $p \xrightarrow{\alpha} p'$, then there is $q \xrightarrow{\beta} q'$ such that $(p', q') \in R$
- if $q \xrightarrow{\beta} q'$, then there is $p \xrightarrow{\alpha} p'$ such that $(p', q') \in R$.

A bisimulation defines equivalence between transition systems by defining a correspondence between states that is stable under transition: given two equivalent states and a transition on the first, there is a corresponding transition such that the output states are also equivalent.

As we will see in the following section, ConCert models smart contracts as pure functions. Since we wish to capture the notion of bisimulations of contracts by defining equivalences of states that are stable under transitions, our specialized definition of bisimulation is a *natural isomorphism* of pure functions. A natural isomorphism of two pure functions defines a correspondence of function inputs and outputs that is stable under application of the function. In the following definition, we consider the category of contracts defined in [14].

► **Definition 2** (Natural Isomorphism of Pure Functions). *Consider functions $F : A \rightarrow B$ and $G : A' \rightarrow B'$. A natural isomorphism between F and G is a pair of isomorphisms, $\iota_A : A \cong A'$ and $\iota_B : B \cong B'$ such that the following square commutes:*

$$\begin{array}{ccc} A & \xleftarrow[\sim]{\iota_A} & A' \\ F \downarrow & & \downarrow G \\ B & \xleftarrow[\sim]{\iota_B} & B' \end{array}$$

Smart contracts modeled as pure functions get their state and entrypoint calls as inputs and as output an updated state with the resulting transactions. Because contract calls result in transitions between contract states, we can consider contracts as state transition systems. The fact that the square commutes is precisely what makes it a bisimulation in this particular interpretation of a state transition system.

3.2 Bisimulations in ConCert

We now move on to give details of a specific implementation of contract bisimulations in ConCert.¹ In ConCert, smart contracts are formalized with a **Contract** type as a pair of pure functions **init** and **receive**. The **init** function governs contract initialization and the **receive** function governs contract calls. The **Contract** type is polymorphic, parameterized by four types: **Setup**, **Msg**, **State**, and **Error** which, respectively, govern the data necessary for contract initialization, contract calls, contract storage, and contract errors. For a contract

C : **Contract** **Setup** **Msg** **State** **Error**

the type signatures of each component function **C**.(**init**) and **C**.(**receive**) are given in Listing 1, where the types **Chain** and **ContractCallContext** are ConCert-specific types used to model the underlying blockchain and context.

■ **Listing 1** Type signatures in ConCert of the **init** and **receive** functions of a contract.

C.(**init**) : **Chain** → **ContractCallContext** → **Setup** → **result** **State** **Error**.

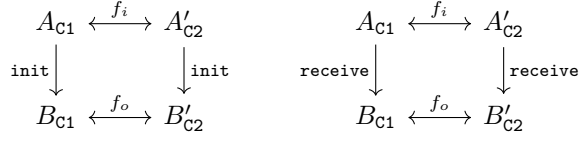
C.(**receive**) : **Chain** → **ContractCallContext** → **State** → **option** **Msg** → **result** (**State** * **list** **ActionBody**) **Error**.

Following the theory in Section 3.1, we formalize bisimulations in ConCert between a pair of contracts **C1** and **C2**,

C1 : **Contract** **Setup1** **Msg1** **State1** **Error1**

C2 : **Contract** **Setup2** **Msg2** **State2** **Error2**,

¹ The definitions and results of this section are available at [theories/ContractMorphisms.v](#)



■ **Figure 1** A bisimulation of contracts in ConCert is a natural isomorphism of each of the component functions `init` and `receive`, which inductively constructs a contract bisimulation.

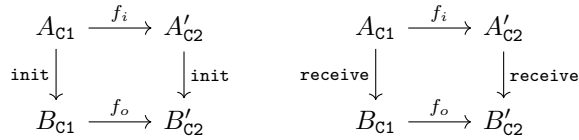
by constructing natural isomorphisms between the `init` and `receive` functions, respectively, of $C1$ and $C2$. This is made by defining a correspondence of inputs and outputs to each of `init` and `receive` which is stable under contract initialization and contract calls. Constructing this equivalence consists of proving that the respective `init` functions are equivalent (the base case) and then show that each of the steps are also equivalent (the inductive step).

As we will see in Section 4, these natural isomorphisms induce a trace equivalence of contracts, or a bisimulation of contracts when considering them as state transition systems. This is an *extensional* equivalence of contracts.

3.2.1 Constructing Bisimulations via Contract Isomorphisms

We can encode these extensional equivalences in ConCert using *contract morphisms*, a category theoretic tool defined in ConCert for reasoning formally about one contract in terms of another [14].

A contract morphism is a formal, structural relationship between two contracts which formally relate the inputs, outputs, and state of both contracts in question. Much like the natural isomorphism from before, contract morphisms are encoded in ConCert as a *natural transformation* of contracts, which in diagram form differ from natural isomorphisms only in that the horizontal arrows only point in one direction (*i.e.* they are not necessarily invertible).



■ **Figure 2** A contract morphism between contracts $C1$ and $C1$ is a natural transformation of `init` and `receive` functions, respectively.

The natural transformation depicted above in Figure 2 corresponds to a morphism f from contracts $C1$ to $C2$, written either $f : C1 \rightarrow C2$ or, more formally,

$$f : \text{ContractMorphism}(C1, C2).$$

Morphisms compose, and there is a canonical identity morphism [14]. These two facts give us everything we need to define an invertible pair of contract morphisms, or a contract *isomorphism*, which will take as our notion of contract bisimulation.

► **Definition 3** (Contract Isomorphism). *A contract isomorphism between contracts $C1$ and $C2$ is a pair of morphisms,*

$$\begin{aligned} f &: \text{ContractMorphism } C1 \ C2 \\ g &: \text{ContractMorphism } C2 \ C1, \end{aligned}$$

such that f and g compose each way to the identity morphism.

To state this as a formal proposition, we summarize this definition as a proposition in Coq.

■ **Listing 2** Contract isomorphisms are defined as a pair of morphisms that compose each way to the identity morphism under the morphism composition function `compose_cm`.

```
Definition is_iso_cm
  (f : ContractMorphism C1 C2) (g : ContractMorphism C2 C1) : Prop :=
  compose_cm g f = id_cm C1 ∧
  compose_cm f g = id_cm C2.
```

4 Contract Bisimulations Induce Trace Equivalences in ConCert

Our task now is to prove that contract isomorphisms actually induce the desired, strong notion of equivalence between state transition systems. In fact, they induce an isomorphism of the generated trace graphs of the contracts in question [16]. In this section we give a formal proof in Coq that a contract isomorphism produces a trace equivalence of contracts. We define trace equivalence in 4.1. We formalize a trace equivalence between contracts in ConCert in Section 4.2, and then show that contract isomorphisms imply trace equivalence in Section 4.3.

4.1 Trace Equivalences

A trace equivalence between contracts modeled as pure functions is an equivalence of all possible execution trace graphs, or graphs where nodes are states and edges are state transitions labelled with emitted transactions. This is proved inductively with a mapping of nodes and edges which maps initial states to initial states (the base case), and that respects state transitions (the inductive step à la Definition 1).

4.2 Trace Equivalences in ConCert

To codify trace equivalences in ConCert, we formally define contract traces and morphisms between contract traces. With morphisms we can formalize equivalence via *trace isomorphisms*. Similar to contract morphisms [14], contract trace morphisms are a formal, structural relationship between the traces of two contracts. As we will see, an equivalence of contract traces is the strong form of extensional equivalence that we are looking for.²

4.2.1 Contract Traces

We begin by defining some key data types. First, a contract's trace is a chained list of contract states, connected by contract steps.

² The definitions and results of this section are available at `theories/Bisimulation.v`

Definition `ContractTrace` ($C : \text{Contract Setup Msg State Error}$) :=
`ChainedList State (ContractStep C).`

Contract steps are a record of the data for a successful contract call, or a call to the `receive` function, which links two contract states. The record contains data for a successful contract call such as the contract call context, the incoming message, the resulting actions, as well as a proof that the call to `receive` succeeds.

■ **Listing 3** Contract steps are successful calls to the `receive` function.

```
Record ContractStep (C : Contract Setup Msg State Error)
  (prev_st : State) (next_st : State) := {
  (* data for a successful contract call *)
  seq_chain : Chain ;
  seq_ctx : ContractCallContext ;
  seq_msg : option Msg ;
  seq_new_acts : list ActionBody ;
  (* we can call receive successfully *)
  recv_some_step :
    receive C seq_chain seq_ctx prev_st seq_msg =
      Ok (next_st, seq_new_acts) ;
  }.
```

Contract traces codify the trace of contracts as state transition systems.

4.2.2 Contract Trace Morphism

A *contract trace morphism*, analogous to a contract morphism of [14], encodes a formal, structural relationship between the traces of two contracts. For contracts

$C_1 : \text{Contract Setup1 Msg1 State1 Error1}$

$C_2 : \text{Contract Setup2 Msg2 State2 Error2},$

a morphism of contract traces includes the following data:

- A function between contract state types, `ct_state_morph : State1 -> State2`.
- A proof that `ct_state_morph` sends valid initial states of C_1 to valid initial states of C_2 .
- A function `cstep_morph` that, for states `state1` and `state2` of C_1 , sends a contract step

`step1 : ContractStep C1 state1 state2,`

to a corresponding contract step of C_2 between the corresponding states

`step2 : ContractStep C2 (ct_state_morph state1) (ct_state_morph state2).`

Inductively, this data gives us a relationship between all reachable states: initial states of each contract are related via the function between state types, and from there, any contract step of C_1 is related to a contract step of C_2 that respects the function on states. We codify this with a type `f : ContractTraceMorphism C1 C2`.

► **Example 4** (The Identity Contract Trace Morphism). For any contract C we can define the identity morphism `id_ctm`, whose component functions are the identity and respective proofs are trivial, and which inhabits the type `ContractTraceMorphism C C`.

► **Example 5** (Contract Trace Morphism Composition). We can define composition of contract trace morphisms similar to composition of contract morphisms in [14], via a function `compose_ctm`, which takes morphisms

`f : ContractTraceMorphism C1 C2 and g : ContractTraceMorphism C2 C3`

and returns a morphism

```
compose_ctm g f : ContractTraceMorphism C1 C3.
```

To compose contract morphisms, we simply compose their component functions. That composition is associative comes trivially. Similarly, it comes immediately that composition on either side with the identity is a trivial operation, and so composition and identity behave as we might expect in a well-defined category.

4.2.3 Contract Trace Isomorphisms

Contract trace isomorphisms are then defined analogously to contract isomorphisms (3.2.1).

► **Definition 6** (Contract Trace Isomorphism). *A contract trace isomorphism between contracts $C1$ and $C2$ is a pair of trace morphisms,*

$$f : \text{ContractTraceMorphism } C1 \ C2$$

$$g : \text{ContractTraceMorphism } C2 \ C1,$$

such that f and g compose each way to the identity morphism id_ctm .

To state this as a formal proposition, we summarize this definition in a type in Coq.

■ **Listing 4** Contract trace isomorphisms are defined as a pair of morphisms that compose each way to the identity morphism.

```
Definition is_iso_ctm
  (m1 : ContractTraceMorphism C1 C2) (m2 : ContractTraceMorphism C2 C1) :=
  compose_ctm m2 m1 = id_ctm C1 ∧
  compose_ctm m1 m2 = id_ctm C2.
```

By definition, if two contracts are related by a contract trace isomorphism, then there is a one-to-one correspondence between all possible contract states; furthermore, this correspondence respects initial states. Thus extensionally, contracts which are trace isomorphic have identical behavior up to their state isomorphisms. When considered as a labelled transition system, their execution graphs are necessarily isomorphic. The behavior of a contract is fully defined by its initial state and the steps it can take from there, and so contract trace isomorphisms give us the strong form of extensional equivalence we are looking for.

4.3 Contract Morphisms to Contract Trace Morphisms

The final result of this section is that contract bisimulations induce contract trace isomorphisms. We prove this result by defining a function `cm_to_ctm`, which takes a contract morphism

$$f : \text{ContractMorphism } C1 \ C2$$

and returns a contract trace morphism

$$\text{cm_to_ctm } f : \text{ContractTraceMorphism } C1 \ C2,$$

which respects identity and compositions. Contract morphisms and contract trace morphisms define a category whose objects are contracts in ConCert, so `cm_to_ctm` is a functor.

11:8 Formal Specification via Bisimulation

To define `cm_to_ctm` for a contract morphism $f : C1 \rightarrow C2$, we need a function between the state types of `C1` and `C2` which respects initial states and state transitions. The obvious candidate is, of course, the component function of f of contract states, `f.state_morph`, which respects initial states state transitions by the coherence conditions of its definition [14].

Furthermore, the identity contract morphism induces the identity contract trace morphism, and compositions of contract morphisms induce compositions of contract trace morphisms.

■ **Listing 5** Identity induces the identity.

```
Theorem cm_to_ctm_id : cm_to_ctm (id_cm C1) = id_ctm C1.
```

■ **Listing 6** Compositions induce compositions.

```
Theorem cm_to_ctm_compose (g : ContractMorphism C2 C3) (f : ContractMorphism C1 C2) :  
  (* the image of the composition = ... *)  
  cm_to_ctm (compose_cm g f) =  
  (* composing the image morphisms *)  
  compose_ctm (cm_to_ctm g) (cm_to_ctm f).
```

Since contract isomorphisms and contract trace isomorphisms are both defined as respective morphism pairs which compose each way to the identity, a bisimulation of contracts induces a trace equivalence. We have our desired result.

5 Using Bisimulation as a Tool for Formal Specification

Contract isomorphisms (bisimulations) could be considered as a tool in at least two ways: first, to reuse proofs on a different contract version by porting them over the isomorphism and achieve those results on the target contract, *e.g.* as in [14]; and second, to use a contract *as a specification*. To show this, in this section we give an example of a contract whose specification is another contract, *e.g.* a reference implementation, and explore the ways in which proofs transport over a contract bisimulation. This is an example where a common optimization makes a contract more difficult to reason about, and we use a contract bisimulation to formally specify the optimized contract with the intelligible contract.³

5.1 Linked Lists and Dynamic Arrays

Consider a simple contract `C_arr` that manages an array of owners, *e.g.* for access control, each identified by a natural number. It has functionality to add owners, remove owners, and swap owners. Consider also a second implementation `C_ll` that does the same, except that it stores owner IDs as a linked list instead of a dynamic array, a common contract optimization strategy over arrays in Solidity which introduces nontrivial challenges to verification [7]. The correctness criteria for the second, optimized implementation are that it behave identically to the reference implementation from an extensional standpoint, precisely because the linked list is supposed to emulate a dynamic array (though more efficiently at the bytecode level).

The two contracts `C_arr` and `C_ll` share setup and entrypoint types, but differ in their storage types and the implementation of their entrypoint functions. Both contracts must maintain a set of owners with no duplicates.

³ The contracts and bisimulations of this section are available at `optimization2.v`

■ **Listing 7** The entrypoint type shared by both `C_arr` and `C_ll`.

```
Inductive entrypoint :=
| addOwner (a : N) (* to add a as an owner ID *)
| removeOwner (a : N) (* to remove a as an owner ID *)
| swapOwners (a_fst a_snd : N). (* to swap a_fst for a_snd as owners *)
```

The first contract, `C_arr`, keeps track of owners in an array in its storage type `storage_arr`.

```
Record storage_arr := { owners_arr : list N }.
```

The optimized contract `C_ll` keeps track of owners in a linked list implemented (somewhat unconventionally in Coq) via a finite mapping.

```
Record storage_ll := { owners_ll : FMap N N }.
```

The mapping emulates an array as follows: Using a global constant `SENTINEL : N`, the empty list is emulated as the mapping which points `SENTINEL` to `SENTINEL`.

```
arr_to_ll := [] => { SENTINEL : SENTINEL }.
```

From here, to insert an element `a` into the mapping, we point `SENTINEL` to `a`, and `a` to whatever `SENTINEL` used to point to (`SENTINEL` if `a` is the first element of the list).

```
arr_to_ll := [a] => { SENTINEL : a ; a : SENTINEL }.
```

This pattern continues such that in the mapping `SENTINEL` always points to the most recently-added element, and elements form a chain until the last points back to `SENTINEL`. So for list of the form `[a, b, c]`, the corresponding mapping points `SENTINEL` to `a`, and `a` to `b`, `b` to `c`, and `c` back to `SENTINEL`.

```
arr_to_ll := [a, b, c] => { SENTINEL : a ; a : b ; b : c ; c : SENTINEL }
```

The three entrypoints behave analogously for their respective data structures. For our array contract, `C_arr`, calling `(addOwner a)` simply appends `a` to the list of owners (provided `a` is not already an owner).

```
addOwner a := { | owners_arr := l | } => { | owners_arr := a :: l | }.
```

For our linked list contract, `C_ll`, calling `(addOwner a)` inserts the owner into the linked list.

```
addOwner a := { | owners_ll := { SENTINEL : a' ; ... } | } =>
{ | owners_ll := SENTINEL : a ; a : a' ; ... | }.
```

Removing an owner behaves similarly: for `C_arr`, `(removeOwner a)` removes `a` from the array,

```
removeOwner a := { | owners_arr := [ ..., b, a, b', ... ] | } =>
{ | owners_arr := [ ..., b, b', ... ] | }.
```

while for `C_ll`, `(removeOwner a)` updates the pointers in the mapping to excise `a`.

```
removeOwner a := { | owners_ll := { ... ; b : a ; a : b' ; ... } | } =>
{ | owners_ll := ... ; b : b' ; ... | }.
```

Finally, to swap owners in `C_arr`, `(swapOwners a a')` replaces `a` with `a'`,

```
swapOwners a a' := { | owners_arr := [ ..., b, a, b', ... ] | } =>
{ | owners_arr := [ ..., b, a', b', ... ] | }.
```

and `C_ll`, does the analogous operation by updating its pointers.

```
swapOwners a a' := { | owners_11 := { ... ; b : a ; a : b' ; ... } | } ⇒
  { | owners_11 := ... ; b : a' ; a' : b' ; ... | }.
```

5.2 The Bisimulation

We now explore the consequences of a bisimulation, or a contract isomorphism, between our reference implementation `C_arr` and its counterpart `C_11`.

Theorem `bisim_arr_11 : contracts_isomorphic C_arr C_11`.

A witness of the proposition `contracts_isomorphic C_arr C_11` is a contract isomorphism between `C_arr` and `C_11`. We first explore how a bisimulation between `C_arr` and `C_11` lets us use code as a specification (5.2.1), and then explore how the specification of each ports over the bisimulation (5.2.2). Note that in the following example we assume some key properties about array and map operations and their properties.

5.2.1 Contract as a Specification

The purpose of any contract optimization is to improve the performance of the code without changing its behavior within some semantic domain. That domain is, at least in principle, the domain of a formal specification. This almost always means that changes can be made *intentionally*, affecting the inner workings of the contract, but *extensional* behavior – behavior from an outside or semantic perspective – should remain the same. In the case of our contracts `C_arr` and `C_11`, we expect `C_11` to behave identically to `C_arr` up to an equivalence of data structures. That precise equivalence, of expected behavior of data structures and contract entypoints, is exactly the data held in the contract isomorphism.

To illustrate this point, we construct the contract morphism. To do so we need functions between entypoint, state, error, and setup types [14]. Because `C_arr` and `C_11` differ only in their entypoint type, these functions are the identity on all but the entypoint type; and for the entypoint type, these are the functions `arr_to_11` and `11_to_arr` specified above in Section 5.1.

■ **Listing 8** The component functions of morphisms between `C_arr` and `C_11`.

```
(* msg, setup, and error morphisms are all identity *)
Definition msg_morph : entypoint → entypoint := id.
Definition setup_morph : setup → setup := id.
Definition error_morph : error → error := id.

(* storage morphisms *)
Definition state_morph : owners_arr → owners_11 := arr_to_11.
Definition state_morph_inv : owners_11 → owners_arr := 11_to_arr.
```

With these component functions we can prove the corresponding coherence conditions, and we get morphisms:

```
f : ContractMorphism C_arr C_11 and f_inv : ContractMorphism C_11 C_arr
```

The key point of data held in this pair of functions, which form a bisimulation, is in the way that they codify the relationship in functionality between storage and entypoints in each contract. This is precisely the data of the argument we made in Section 5.1 that `C_11` was indeed an alternative representation of `C_arr`.

Consider in particular the behavior of calling `(addOwner a)`. We know from Section 5.1 that in `C_arr` this appends `a` to the list of owners, while in `C_ll` this inserts `a` into the implemented linked list. We have a formal proof of this correspondence in the following two lemmas. The functions `add_owner_arr` and `add_owner_ll` are, respectively, the functions that implement the `addOwner` entrypoint in each of `C_arr` and `C_ll`.

■ **Listing 9** Two coherence results which show the correspondence of the `addOwner` entrypoint between `C_arr` and `C_ll`.

```
Lemma add_owner_coh : forall a st st' acts,
  add_owner_arr a st = Ok (st', acts) →
  add_owner_ll a (state_morph st) = Ok (state_morph st', acts).
```

```
Lemma add_owner_coh' : forall a st e,
  add_owner_arr a st = Err e →
  add_owner_ll a (state_morph st) = Err e.
```

These are coherence results à la Figure 2: adding `a` to the state of `C_arr` and then transforming the state to a linked list is the same as transforming the state to a linked list first and then adding `a` to the state of `C_ll`, and vice versa. They constitute a formal proof that the behavior of the two contracts is the same up to the equivalence of their data structures for the `addOwner` entrypoint.

We have analogous proofs for each of the remaining two entrypoints of `C_arr` and `C_ll`. That they give us a bisimulation of contracts tells us that the behavior of the two contracts is the same up to the equivalence of their data structures for each entrypoint – and the equivalence of their data structures is precisely a formal description of how arrays are emulated as linked lists in the state of `C_ll`. How could you possibly be more precise in formally specifying `C_ll` as an optimization of `C_arr` than by a formal proof like this that the two contracts are extensionally equivalent?

5.2.2 Porting Properties Over the Bisimulation

Standard practice for comparing an optimized contract to its reference implementation would be to apply the same test suites or formal specification to the new contract and ensure that it passes all tests and still conforms to the formal specification. If the formal specification includes details of the inner workings of the contract, then relevant alterations are made to the formal specification to accommodate the new setting. This is a translation effort, which can be prone to mistranslation and resulting errors by underspecification, so instead we would rather see if we can port previously-proved results over a bisimulation.

Indeed, we can and we will do so here with a key property for both contracts: that there be no duplicate owner IDs in storage. This property is important not only because of the intended contract functionality of `C_arr`, but also in the optimization of `C_arr` into `C_ll`. Due to the implementation of `C_ll` as a linked list via a mapping, being able to add a “duplicate” would actually compromise the integrity of the linked list as a model of an array: the mapping only allows for an owner ID to point to one other owner, so adding a “duplicate” would mean altering the pointers and unlinking the data structure. That there not be duplicates is thus an important property both from the perspective of high-level functionality (with respect to contract permissions and control flow) as well as from the perspective of low-level implementation correctness (linked list implementation emulating an array).

We first formally verify the reference implementation, `C_arr`, by proving that all reachable contract states are free of duplicates, codified in the following result.

■ **Listing 10** All reachable states of `C_arr` have no duplicate owners in storage.

```
Theorem no_dup_arr (st : owners_arr) :  
  reachable C_arr st → no_duplicates_arr st.
```

Using the bisimulation, we can now prove the analogous result about `C_ll` using `morphism_induction`, a proof technique that leverages contract morphisms to compare the reachable states of contracts related by contract morphisms [14].

► **Lemma 7** (Morphism Induction). *Consider contracts `C1` and `C2` and a contract morphism `f` : `ContractMorphism C1 C2`. Then every reachable state `st_1` of `C1` corresponds to a reachable state `st_2` of `C2`, related by the state morphism component of `f` such that*

$$\text{st_2} == f.(\text{state_morph}) \text{ st_1}.$$

This lemma is codified as `left_cm_induction` in `FinCert`.⁴

Because we have a bisimulation, not only do we know that the states of `C_arr` and `C_ll` are related by `state_morph` described in Section 5.2.1, but we know that `state_morph` has an inverse. Thus using the details of that morphism we can prove that `C_arr` has duplicates in storage if and only if `C_ll` has been unlinked, the analogous property for duplicates in a linked list. By morphism induction, then, we have the analogous result on `C_ll`.

■ **Listing 11** The desired result that all reachable states of `C_ll` have no duplicate owners in storage.

```
Theorem no_dup_ll (st : owners_ll) :  
  reachable C_ll st → no_duplicates_ll st.
```

6 Conclusion

The efficacy of formal verification on smart contracts depends on being able to correctly specify and carry out the verification of optimized code. However, code optimized for performance is rarely optimized for intelligibility, which can make formally verifying optimized code difficult and costly. To remedy this, we introduced contract isomorphisms, a formal tool that establishes a structural equivalence between smart contracts, and we proved that contract isomorphisms give us full trace equivalences of contracts. We then demonstrated how contract isomorphisms can be used to formally specify and verify an optimized smart contract by proving it extensionally equivalent to its reference implementation. Our example illustrates the practical application of this framework to a common optimization technique in smart contract development. It shows how formal proofs of correctness can be ported over a bisimulation and how a bisimulation enables the use of a contract as a specification. We hope that this work paves the way for more robust and reliable smart contract verification, enabling practitioners to more easily reason about optimized contracts in terms of their more intelligible reference implementations.



References

- 1 Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. ConCert: A smart contract certification framework in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 215–228, New York, NY, USA, January 2020. Association for Computing Machinery. doi:10.1145/3372885.3373829.




⁴ theories/ContractMorphisms.v

- 2 Cyril Cohen, Enzo Crance, and Assia Mahboubi. TrocQ: proof transfer for free, with or without univalence. In *European Symposium on Programming*, pages 239–268. Springer, 2024. doi:10.1007/978-3-031-57262-3_10.
- 3 Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In *International Conference on Certified Programs and Proofs*, pages 147–162. Springer, 2013. doi:10.1007/978-3-319-03545-1_10.
- 4 Hubert Garavel and Frédéric Lang. Equivalence checking 40 years after: A review of bisimulation tools. *A Journey from Process Algebra via Timed Automata to Model Learning: Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday*, pages 213–265, 2022. doi:10.1007/978-3-031-15629-8_13.
- 5 Osman Hasan and Sofiene Tahar. Formal verification methods. In *Encyclopedia of Information Science and Technology, Third Edition*, pages 7162–7170. IGI global, 2015.
- 6 Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, January 1985. doi:10.1145/2455.2460.
- 7 Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Aleksandar Nanovski, and Mooly Sagiv. Effectively-propositional reasoning about reachability in linked data structures. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 756–772, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-39799-8_53.
- 8 Kim G Larsen. Proof systems for satisfiability in Hennessy-Milner logic with recursion. *Theoretical Computer Science*, 72(2-3):265–288, 1990. doi:10.1016/0304-3975(90)90038-J.
- 9 Robin Milner. Communication and concurrency. *Prentice Hall International*, 13, 1989.
- 10 Talia Ringer, RanDair Porter, Nathaniel Yazdani, John Leo, and Dan Grossman. Proof repair across type equivalences. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 112–127, 2021. doi:10.1145/3453483.3454033.
- 11 Davide Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8(5):447–479, October 1998. doi:10.1017/S0960129598002527.
- 12 Davide Sangiorgi. *Introduction to bisimulation and coinduction*. Cambridge University Press, 2011.
- 13 Derek Sorensen. FinCert. Software (visited on 2025-05-08). URL: <https://github.com/dhsorens/FinCert/tree/FMBC-25>, doi:10.4230/artifacts.23081.
- 14 Derek Sorensen. Towards Formally Specifying and Verifying Smart Contract Upgrades in Coq. *5th International Workshop on Formal Methods for Blockchains (FMBC 2024)*, 2024.
- 15 Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. Equivalences for free: univalent parametricity for effective transport. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–29, 2018. doi:10.1145/3236787.
- 16 Johan Van Benthem and Jan Bergstra. Logic of transition systems. *Journal of Logic, Language and Information*, 3:247–283, 1994. doi:10.1007/BF01160018.

Isabelle/Solidity: A Tool for the Verification of Solidity Smart Contracts

Asad Ahmed   

University of Exeter, UK

Diego Marmsoler   

University of Exeter, UK

Abstract

Smart contracts are an important innovation in Blockchain which allow to automate financial transactions. Every day, hundreds of thousands of new contracts are deployed managing millions of dollars' worth of transactions. Thus, bugs in smart contracts may lead to high financial losses and it is important to get them right before deploying them to the Blockchain. To address this problem we developed Isabelle/Solidity, a tool for the verification of smart contracts in Isabelle. The tool is implemented as a definitional extension for the Isabelle proof assistant and thus complements existing tools in this area which are mostly based on axiomatic approaches. In this paper we describe Isabelle/Solidity and demonstrate it by verifying a casino contract from the VerifyThis long term verification challenge.

2012 ACM Subject Classification Security and privacy → Logic and verification

Keywords and phrases Program Verification, Smart Contracts, Isabelle, Solidity

Digital Object Identifier 10.4230/OASICS.FMBC.2025.12

Category Tool Paper

Supplementary Material *Software*: <https://doi.org/10.5281/zenodo.15121526>

Funding This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/X027619/1].

1 Introduction

One important innovation which comes with Blockchain are so-called *smart contracts*. These are digital contracts which are automatically executed once certain conditions are met and which are used to automate transactions on the Blockchain. For instance, a payment for an item might be released instantly once the buyer and seller have met all specified parameters for a deal. Every day, hundreds of thousands of new contracts are deployed [9] managing millions of dollars' worth of transactions [18].

Technically, a smart contract is *code which is deployed to a Blockchain* and which can be executed by sending special transactions to it. Thus, as for every computer program, smart contracts may contain bugs which can be exploited. However, since smart contracts are often used to automate financial transactions, such exploits may result in huge economic losses. In general, it is estimated that since 2019, more than \$5B was stolen due to vulnerabilities in smart contracts [5].

The high impact of vulnerabilities in smart contracts together with the fact that once deployed to the Blockchain, they cannot be updated or removed easily, makes it important to “*get them right*” before they are deployed. To address this problem, we developed Isabelle/Solidity, a tool for the deductive verification of Solidity smart contracts implemented as a definitional extension for the Isabelle [16] proof assistant.



© Asad Ahmed and Diego Marmsoler;

licensed under Creative Commons License CC-BY 4.0

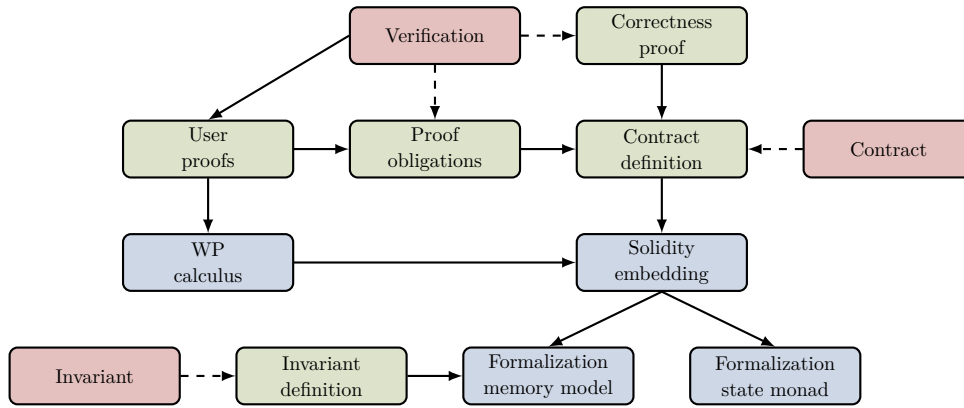
6th International Workshop on Formal Methods for Blockchains (FMBC 2025).

Editors: Diego Marmsoler and Meng Xu; Article No. 12; pp. 12:1–12:9

OpenAccess Series in Informatics



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



■ **Figure 1** Isabelle/Solidity: theories (blue), commands (red), and generated artefacts (green).

The general approach of Isabelle/Solidity is already described in [12]. With this paper we describe the tool itself in more detail as well as some extensions to improve its usability. In particular the contributions of this paper are as follows:

- We extended Isabelle/Solidity with new features to support the specification of invariants as well as the specification and verification of postconditions for functions.
- We describe the architecture and implementation of Isabelle/Solidity as a definitional package for Isabelle.
- We demonstrate the tool by describing how it can be used to verify the casino contract from the VerifyThis long term verification challenge.

2 Isabelle/Solidity

Isabelle/Solidity is based on the formal semantics of Solidity described in [13, 14] and the verification methodology described in [2] and verified in [15]. Its architecture is depicted in Figure 1. Isabelle/Solidity is based on four Isabelle theories which are represented by the blue rectangles:

Formalization state monad We model Solidity programs as functions which manipulate states. Such functions are usually described in terms of state monads [17] and this theory contains our formalization of it based on [6].

Formalization memory model Solidity has a quite particular memory model with different types of stores which support different types of data structures (See Table 1). This theory contains our formalization of the Solidity memory model. It also defines the notion of a Solidity state as a collection of different stores.

Solidity embedding A Solidity statement is defined as a particular state monad manipulating a Solidity state. This theory contains definitions for all of our Solidity statements.

WP calculus To support a user in the verification of Solidity programs, Isabelle/Solidity comes with a verification condition generator (VCG). The VCG is based on a weakest precondition calculus [8] for our Solidity statements and which is formalized in this theory.

■ **Table 1** Types of stores and corresponding properties.

	Stack	Storage	Memory	Calldata
Persistence	Temporary	Permanent	Temporary	Temporary
Mutable	No	Yes	No	No
Scope	Local to function	Global	Local to contract	Local to contract

In addition to the theories described above, Isabelle/Solidity extends the Isabelle proof assistant by means of three new definitional commands represented by the **red rectangles** in Figure 1. Each of the commands generates lower-level definitions and theorems for Isabelle/HOL which are represented as **green rectangles**.

Contract This command allows a user to specify a new contract. It requires them to specify a list of member variables, a constructor, and a list of methods. For each method the user can specify a list of parameters as well as a method body. The body is provided as a state monad using the monads defined in the corresponding Isabelle theory. The command then generates definitions for the *contract's methods*. To this end each method is mapped to a corresponding partial function definition [11].

Invariant This command supports a user in the specification of an invariant for the contract. An invariant is specified as a HOL formula over the store and balance of the contract. The command then uses the typing information of member variables to generate a corresponding *invariant definition*.

Verification This command triggers the verification of a contract. It requires a user to provide an invariant as well as postconditions for the methods. Then it presents the user with a list of *proof obligations* for each of the contract's methods. The users then need to discharge these proof obligations by providing a corresponding *proof*. To this end they can use the WP calculus provided by our framework. After discharging these obligations, Isabelle/Solidity proves an overall *correctness theorem* which guarantees that the invariant as well as postconditions are not violated.

3 Example: Casino Contract

To demonstrate Isabelle/Solidity in action, we use it to verify the casino contract from the VerifyThis long-term verification challenge [1]. The version of the contract used here is provided in Appendix A. The casino contract implements a betting game based on guessing the outcome of coin-tossing. At any time, the game is characterized by three states, i.e., **IDLE**, **GAME_AVAILABLE** and **BET_PLACED**. Initially, the game is in the **IDLE** state. The game has been implemented in Solidity using the following functions.

- The operator can create a new game by calling the **creatGame** function and provide the hash value of a secret number. The function stores the hash value and changes the state to **GAME_AVAILABLE**.
- Once in state **GAME_AVAILABLE**, a player can place a bet by invoking the **placeBet** function and provide a guess (**HEADS** or **TAILS**). This function stores the player's address, its guess, and the amount of the bet, and changes the state to **BET_PLACED**.
- Now, the operator can decide the bet anytime by calling the **decideBet** function and providing the secret number. The secret number is then used to decide the outcome of the coin tossing (**HEADS** or **TAILS**). If the player's guess is correct, the double of the amount of the bet is transferred to their address. Otherwise, the amount equal to the bet is added to the pot. The function also changes the state of the game to **IDLE**.
- The operator may add money to the bet, at anytime, using **addToPot** but can only remove money if the game is not in state **BET_PLACED** by calling **removeFromPot**.

Note that all interactions with a smart contract are usually visible on the blockchain. This is why the secret number needs to be hashed before stored on the blockchain. Moreover, this is also the reason why we cannot use a boolean value to represent the value of **HEADS** or **TAILS**. Doing so would make it feasible to precompute the hash values for **True** and **False** and compare them to the hash value stored on the blockchain to identify the secret.

4 Specification

To verify a smart contract in Isabelle/Solidity we first need to specify it. To this end Isabelle/Solidity provides the `contract` command which supports a user in this task. The command requires a user to specify a list of storage variables and corresponding types, followed by a specification of the constructor and the contract's methods.

Example 4.1 (Specifying storage variables in Isabelle/Solidity). Listing 1 shows the specification of storage variables for the casino contract in Isabelle/Solidity.

Listing 1: Isabelle/Solidity data types for Casino

```
1 contract Casino
2   for state: StateT
3   and operator: AddressT
4   and player: AddressT
5   and pot: IntT
6   and hashedNumber: BytesT
7   and bet: IntT
8   and guess: CoinT
```

In general, Isabelle/Solidity supports most of the basic Solidity data types, such as bit-sized integers, bytes, and addresses. Enums can be encoded as integers with corresponding abbreviations.

Methods

The specification of a new method can be done using the keyword `emethod`. To allow a method to receive funds we can set the payable flag by using the corresponding `payable` keyword. What follows is a specification of stack (`param`), memory (`memory`), and calldata (`calldata`) parameters and corresponding types. Finally, one can provide the body of the function using the `where` keyword followed by a corresponding monad specification (using "`do {...}`", notation).

We do not show the specification for all of the methods of the casino contract here but we only discuss one. The others can be similarly translated from the original Solidity contract.

Example 4.2 (Specifying functions in Isabelle/Solidity). Listing 2 shows the specification of the function `decideBet`. It is declared to be payable and accepts one parameter `secretNumber`, of integer type. Lines 5-7, implement preconditions for deciding a bet, i.e., only the operator can call the function, the game should be in state `BET_PLACED` and `secretNumber` should be equal to the `hashedNumber`. For this purpose, Isabelle/Solidity employs `assert` which models the Solidity `require` command.

Isabelle/Solidity also supports other Solidity statements, such as control structures and assignment operators. For example, in Line 9, `IF...THEN...ELSE` reveals HEADS or TAILS by taking the modulus (`<%>`) of `secretNumber`. For assignment operators, Isabelle/Solidity distinguishes between stack (`::=`) and storage (`::=s`) assignments along with corresponding lookup operators (`~` and `~s`).



Listing 2: Isabelle/Solidity method for Casino

```

1 emethod decideBet payable
2   param secretNumber: IntT
3   where
4     do {
5       byOperator;
6       inState (Sint BET_PLACED);
7       <assert> (hashedNumber ~s [] <=> (<keccak256> (secretNumber ~ [])));
8       decl TSint secret;
9       secret [] ::= IF ((secretNumber ~ []) <=> (<sint> 2) <=> (<sint> 0)
10        THEN <sint> HEADS ELSE <sint> TAILS;
11       IF (secret ~ []) <=> (guess ~s []) THEN
12         do {
13           pot [] ::=s ((pot ~s []) <-> (bet ~s []));
14           bet [] ::=s <sint> 0;
15           <transfer> (player ~s []) ((bet ~s []) <*> (<sint> 2))
16         }
17       ELSE
18         do {
19           pot [] ::=s pot ~s [] <+> bet ~s [];
20           bet [] ::=s <sint> 0
21         };
22       state [] ::=s <sint> IDLE
23     },

```

5 Verification

Isabelle/Solidity facilitates the specification of invariants using the `invariant` command. This command requires a user to provide the name of the invariant, followed by its specification in terms of a predicate formulated over the contract's store and balance. It then generates a definition for the invariant which, in addition to the predicate specified by the user, also requires the value of member variables to adhere to their types. The command also proves corresponding introduction and elimination rules which can be invoked during verification.

Example 5.1 (Specifying invariants in Isabelle/Solidity). Assume that we want to ensure our contract has always enough funds to cover the payout of players. More formally, we want to ensure that, whenever the game is in the `BET_PLACED` state, the contract's internal balance satisfies:

$$pot_balance(s, b) = b \geq s("pot") + s("bet") \wedge s("bet") \leq s("pot") \quad (1)$$

and if it is not in `BET_PLACED`, then

$$pot_balance(s, b) = b \geq pot \quad (2)$$

The corresponding specification in Isabelle/Solidity is given in Listing 3.



Listing 3: Invariant in Isabelle/Solidity

```

1 invariant pot_balance sb where
2   (fst sb state = Value (Sint BET_PLACED)
3    → snd sb ≥ unat (sint (vt (fst sb pot)))
4      + unat (sint (vt (fst sb bet)))
5      ∧ sint (vt (fst sb bet)) ≤ sint (vt (fst sb pot))) ∧
6   (fst sb state ≠ Value (Sint BET_PLACED)
7    → snd sb ≥ unat (sint (vt (fst sb pot))))
8   for "casino"

```

To formally verify the invariant, Isabelle/Solidity provides the `verification` command. It requires the user to provide a name followed by an invariant specified using the invariant keyword. Moreover, a user can provide postconditions for the constructor and each of the contract's methods.

Example 5.2 (Verifying contracts in Isabelle/Solidity). The corresponding specification for our casino contract is shown in Listing 4.



Listing 4: Verification in Isabelle/Solidity

```

1 verification pot_balance:
2   pot_balance
3   "K (K (K True))"
4   "createGame" "createGame_post" and
5   "placeBet" "placeBet_post" and
6   "decideBet" "decideBet_post" and
7   "addToPot" "K (K (K True))" and
8   "removeFromPot" "K (K (K (K True)))"
9   for "casino"

```

The postconditions require a method to update the corresponding state of the game properly. For example, `placeBet_post` requires the `placeBet` method to change the state of the contract to `BET_PLACED`. The postcondition is expressed using `abbreviation` in Isabelle/Solidity as show in Listing 5.



Listing 5: Post-condition in Isabelle/Solidity

```

1 abbreviation(in Contract) placeBet_post where
2   "placeBet_post hn start_state return_value end_state ≡
3     state.Storage end_state this state = BET_PLACED"

```

The verification command tries to prove a general correctness theorem for our contract. To this end, it provides the user with a set of proof obligations which are required to be discharged to verify the contract.

Example 5.3 (Verifying contracts in Isabelle/Solidity). For our example, the verification command provides us with six different proof obligations (one for each of the contract's functions). The one for `decideBet` is shown in Listing 6.



Listing 6: Verifying casino contract in Isabelle/Solidity

```

1  show  $\bigwedge$  call secretNumber. ( $\bigwedge$  x h r. effect (call x) h r  $\Rightarrow$  vcond x) h r)
2   $\Rightarrow$  effect (decideBet call secretNumber) s r
3   $\Rightarrow$  inv_state pot_balance s
4   $\Rightarrow$  post s r pot_balance (K True) (decideBet_post secretNumber)
5  unfolding decideBet_def
6  apply (erule post_exc_true, erule_tac post_wp)
7  unfolding inv_state_def
8  apply (vcg | solve<auto simp add: wpsimps>)+
9  ...

```

The goal basically requires to show that the invariant is preserved by the function. To discharge it, a user can use our verification condition generator and general Isabelle reasoning infrastructure.

6 Related Work

Early approaches to the verification of smart contracts are mostly based on automatic verification techniques. A popular example of research in this area is solc-verify [10] which is based on the boogie verifier [7]. While tools in this category are fully automated, they are often limited in expressiveness compared to interactive verification approaches.

One line of research in this area focusses on the development of tools to verify smart contracts independent of the programming language. One example in this area is the work of Cassez et al. [3, 4] about deductive verification of smart contracts with Dafny. While these tools can also be used to verify Solidity smart contracts, they may reach their limits when it comes to contracts involving more specialized language features.

Other tools focus on the verification of contracts written in a particular language, such as Solidity. One example here is SoliDiKeY [2] which allows to formally specify and verify Solidity smart contracts using the KeY prover. SoliDiKeY is based on an axiomatic semantics of Solidity which is the main difference to Isabelle/Solidity, which is based on a denotational semantics and implemented in a definitional approach.

7 Conclusion

In this paper, we present Isabelle/Solidity, a tool for the formal specification and verification of Solidity smart contracts. The tool is implemented as a definitional extension of the Isabelle proof assistant and provides features for the specification of contracts and invariants, and the verification of invariants and postconditions. To support the user in the verification it also provides a verification condition generator based on a weakest precondition calculus.

Isabelle/Solidity currently supports many features of Solidity, including domain specific expressions and advanced data types such as maps and arrays and their representation in different types of stores (storage, memory, calldata). There are, however, more advanced features of the language, such as inheritance, which is currently not supported and a task for future work.

Another limitation is the lack of a verified compiler for Isabelle/Solidity. Thus, it cannot be guaranteed that the verified properties hold on the level of EVM bytecode. Thus, another direction for future work is the development of a verified compiler for Isabelle/Solidity.

References

- 1 Wolfgang Ahrendt. Welcome to Fabulous Las Contract Blockchain, December 2024. URL: <https://web.archive.org/web/20241209135636/https://verifythis.github.io/02casino/>.

- 2 Wolfgang Ahrendt and Richard Bubel. Functional Verification of Smart Contracts via Strong Data Integrity. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications: 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part III 9*, pages 9–24. Springer, 2020. doi:10.1007/978-3-030-61467-6_2.
- 3 Franck Cassez, Joanne Fuller, and Horacio Mijail Anton Quiles. Deductive Verification of Smart Contracts with Dafny. In Jan Friso Groote and Marieke Huisman, editors, *Formal Methods for Industrial Critical Systems - 27th International Conference, FMICS 2022, Warsaw, Poland, September 14–15, 2022, Proceedings*, volume 13487 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 2022. doi:10.1007/978-3-031-15008-1_5.
- 4 Franck Cassez, Joanne Fuller, and Horacio Mijail Anton Quiles. Deductive Verification of Smart Contracts with Dafny. *Int. J. Softw. Tools Technol. Transf.*, 26(2):131–145, 2024. doi:10.1007/S10009-024-00738-1.
- 5 CipherTrace. Cryptocurrency Crime and Anti-money Laundering Report. Technical report, mastercard, 2021.
- 6 David Cock, Gerwin Klein, and Thomas Sewell. Secure Microkernels, State Monads and Scalable Refinement. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 167–182. Springer, 2008. doi:10.1007/978-3-540-71067-7_16.
- 7 Robert DeLine and K Rustan M Leino. Boogiepl: A Typed Procedural Language for Checking Object-oriented Programs. Technical report, Citeseer, 2005.
- 8 Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM*, 18(8):453–457, August 1975. doi:10.1145/360933.360975.
- 9 Etherscan. Ethereum Daily Deployed Contracts Chart, February 2025. URL: <https://etherscan.io/chart/deployed-contracts>.
- 10 Ákos Hajdu and Dejan Jovanović. solc-verify: A Modular Verifier for Solidity Smart Contracts. In *Verified Software. Theories, Tools, and Experiments: 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13–14, 2019, Revised Selected Papers 11*, pages 161–179. Springer, 2020.
- 11 Alexander Krauss. Recursive Definitions of Monadic Functions. *Electronic Proceedings in Theoretical Computer Science*, 43:1–13, 2010. doi:10.4204/eptcs.43.1.
- 12 Diego Marmosoler, Asad Ahmed, and Achim D Brucker. Secure Smart Contracts with Isabelle/Solidity. In *International Conference on Software Engineering and Formal Methods*, pages 162–181. Springer, 2024. doi:10.1007/978-3-031-77382-2_10.
- 13 Diego Marmosoler and Achim D. Brucker. A Denotational Semantics of Solidity in Isabelle/HOL. In Radu Calinescu and Corina S. Păsăreanu, editors, *Software Engineering and Formal Methods*, pages 403–422, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-92124-8_23.
- 14 Diego Marmosoler and Achim D. Brucker. Isabelle/Solidity: A deep embedding of Solidity in Isabelle/HOL. *Form. Asp. Comput.*, 37(2), March 2025. doi:10.1145/3700601.
- 15 Diego Marmosoler and Billy Thornton. SSCalc: A Calculus for Solidity Smart Contracts. In Carla Ferreira and Tim A. C. Willemse, editors, *Software Engineering and Formal Methods*, pages 184–204, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-47115-5_11.
- 16 T. Nipkow, L. C. Paulson, and M. Wenzel. Isabelle/HOL: A Proof Assistant for Higher-Order Logic, 2002.
- 17 Philip Wadler. Monads for Functional Programming. In Manfred Broy, editor, *Program Design Calculi*, pages 233–264. Springer, 1993. doi:10.1007/978-3-662-02880-3_8.
- 18 Ycharts. Ethereum Transactions Per Day, February 2025. URL: https://ycharts.com/indicators/ethereum_transactions_per_day.

A Casino: Solidity Smart Contract

Listing 7: Solidity source code for the Casino

```

1  contract Casino {
2      enum Coin { HEADS, TAILS } ;
3      enum State { IDLE, GAME_AVAILABLE, BET_PLACED }
4      State private state;
5      address public operator, player;
6      uint public pot;
7      bytes32 public hashedNumber;
8      uint public bet;
9      Coin guess;
10
11     function createGame(bytes32 hashNum)
12         public byOperator, inState(IDLE) {
13         hashedNumber = hashNum;
14         state = GAME_AVAILABLE;
15     }
16
17     function placeBet(Coin _guess) public payable inState(GAME_AVAILABLE) {
18         require (msg.sender != operator);
19         require (msg.value <= pot);
20         state = BET_PLACED;
21         player = msg.sender;
22         bet = msg.value;
23         guess = _guess;
24     }
25
26     function decideBet(uint secretNumber)
27     public byOperator, inState(BET_PLACED) {
28         require (hashedNumber == keccak256(secretNumber));
29         Coin secret = (secretNumber % 2 == 0)? HEADS : TAILS;
30         if (secret == guess) {
31             pot = pot - bet;
32             bet = 0;
33             player.transfer(bet*2);
34         } else {
35             pot = pot + bet;
36             bet = 0;
37         }
38         state = IDLE;
39     }
40
41     function addToPot() public payable byOperator {
42         pot = pot + msg.value;
43     }
44
45     function removeFromPot(uint amount) public byOperator, noActiveBet {
46         operator.transfer(amount);
47         pot = pot - amount;
48     }
49 }

```


A Benchmark Framework for Byzantine Fault Tolerance Testing Algorithms

João Miguel Louro Neto   

Delft University of Technology, The Netherlands

Burcu Kulahcioglu Ozkan   

Delft University of Technology, The Netherlands

Abstract

Recent discoveries of vulnerabilities in the design and implementation of Byzantine fault-tolerant protocols underscore the need for testing and exploration techniques to ensure their correctness. While there has been some recent effort for automated test generation for BFT protocols, there is no benchmark framework available to systematically evaluate their performance.

We present **BYZZBENCH**, a benchmark framework designed to evaluate the performance of testing algorithms in detecting Byzantine fault tolerance bugs. **BYZZBENCH** is designed for a standardized implementation of BFT protocols and their execution in a controlled testing environment. It controls the nondeterminism in the concurrency, network, and process faults in the protocol execution, enabling the functionality to enforce particular execution scenarios and thereby facilitating the implementation of testing algorithms for BFT protocols.

2012 ACM Subject Classification Security and privacy → Distributed systems security

Keywords and phrases Byzantine Fault Tolerance, BFT Protocols, Automated Testing

Digital Object Identifier 10.4230/OASICS.FMBC.2025.13

Category Tool Paper

Funding *Burcu Kulahcioglu Ozkan*: This work has been supported by the University Blockchain Research Initiative (UBRI), funded by Ripple.

1 Introduction

Byzantine Fault-Tolerant (BFT) protocols are at the heart of modern consortium-based blockchain systems. These systems use BFT protocols to reach consensus on the total order of transactions to commit, among a cluster of processes in a fault-tolerant manner. BFT protocols promise correctness even if some processes in the cluster behave maliciously or fail to follow the protocol specification. Along with the increasing popularity of blockchain systems in the last decades, numerous BFT protocols have been proposed.

The correct design and implementation of BFT protocols is crucial to ensure the correct functioning of these systems. However, BFT systems are prone to Byzantine fault tolerance bugs, i.e., flaws or errors that break their fault-tolerance and correctness properties in the presence of Byzantine and network attacks. These bugs allow attackers to exploit vulnerabilities in the system, undermining its ability to maintain correct behavior, as any mistake can lead to serious reliability and security problems with potentially catastrophic consequences. Several studies [1, 27, 21, 5, 26, 34] have discovered vulnerabilities in various BFT protocols and their implementations, underscoring the need for robust analysis and testing methodologies.

Recent work [6, 34] proposes new methods for automated testing of Byzantine fault tolerance in large-scale distributed systems. Twins [6], which systematically tests for Byzantine fault-tolerance by simulating Byzantine processes using twin replicas, is implemented in DiemBFT [32] and is available in Diem's production repository. ByzzFuzz [34], a randomized



© João Miguel Louro Neto and Burcu Kulahcioglu Ozkan;
licensed under Creative Commons License CC-BY 4.0

6th International Workshop on Formal Methods for Blockchains (FMBC 2025).

Editors: Diego Marmosoler and Meng Xu; Article No. 13; pp. 13:1–13:11

OpenAccess Series in Informatics



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

test generator simulating Byzantine faults using message mutations, is available to test the implementations of Tendermint [8] and the XRP Ledger [29]. Given the discovery of new Byzantine fault tolerance bugs in production BFT systems and their increasing popularity, we can expect the development of more analysis and testing methods to analyze their correctness.

Despite advancements in automated test generation for Byzantine Fault Tolerance (BFT) protocols, the evaluation of these algorithms is often limited to the specific systems for which they were developed. The algorithms are assessed based on different implementations of BFT systems, making it challenging to compare their results across various contexts. To enable a comparative evaluation of their performance, it is necessary to have a collection of benchmark applications that represent a diverse range of bugs. While such benchmark suites exist for concurrency bugs such as RADBench [19] and JaConTeBe [22] or for programs with defects in specific programming languages such as BegBunch [12], Defects4J [20], BEARS [24], and GoBench [35] no such benchmark suite is available for Byzantine fault tolerance bugs. **BYZZBENCH** specifically addresses these limitations by enabling Byzantine fault injection capabilities and providing specialized property checkers, making it uniquely tailored to the challenges faced in the design and implementation of BFT protocols.

A major challenge for building a *benchmark suite* of Byzantine fault tolerance bugs in BFT protocols is the lack of a *benchmarking framework* that provides functionality for (i) uniform implementation of BFT protocols and (ii) a controlled protocol execution environment to facilitate the implementation of testing algorithms. Testing algorithms for BFT algorithms generate execution scenarios with particular concurrency, network, and process behavior, e.g., with particular delivery ordering or timing of the protocol messages, network faults to delay or lose some messages, and Byzantine process faults with specific malicious behavior. The enforcement of the generated test scenarios requires controlling the nondeterminism in the executions of the systems under test to exercise the generated specific test scenarios, which demands technical effort and makes it hard to test a broad set of BFT systems.

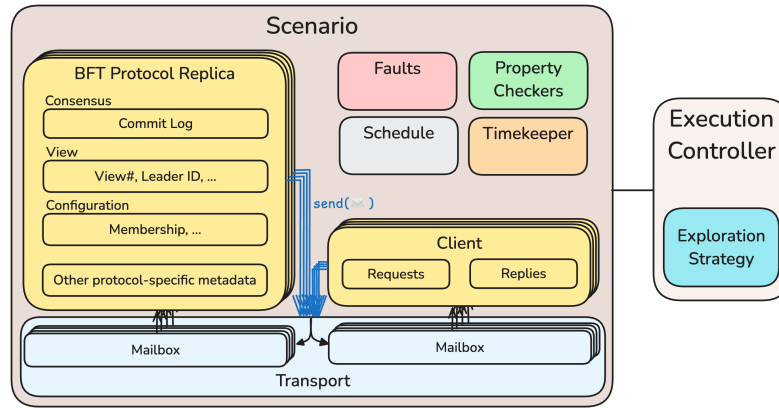
We present **BYZZBENCH** [28], a publicly available and extensible benchmark framework for implementing BFT protocols and evaluating the performance of the testing algorithms in detecting Byzantine fault tolerance bugs. The contribution of **BYZZBENCH** is two-fold. First, it allows protocol designers to validate their protocol implementations by testing them using the testing and exploration algorithms in the framework. Second, it allows functionality and interfaces for implementing Byzantine fault tolerance testing protocols in a controlled execution environment, offering a unified evaluation of testing methods on a set of benchmark protocol implementations. In summary, **BYZZBENCH** allows the users to:

- *prototype* BFT protocols in a unified framework of benchmark applications,
- *implement* testing and exploration methods for BFT protocols in a unified framework,
- *evaluate* relative performances of different testing and exploration methods,
- *gain insight* about the BFT protocol vulnerability characteristics.

The long-term goal of this work is to build a comprehensive, publicly available benchmark suite of BFT protocol implementations to evaluate the effectiveness of testing and exploration tools for BFT protocols. As an initial step, this paper presents a benchmark framework for BFT protocol implementations.

2 The Benchmark Framework

Figure 1 presents a high-level overview of the architecture of **BYZZBENCH**, with the main components of (i) a cluster of processes running a BFT protocol for state machine replication, (ii) client processes that submit requests to the cluster, (iii) controlled transport layer which



■ **Figure 1** High-level architecture of the **BYZZBENCH** framework.

maintains the in-flight messages and delivers them to recipients, and (iv) an execution controller, which dictates a specific execution scenario to the transport layer. The execution controller allows the implementation of different BFT testing algorithms by offering control over the timeouts, network faults, and benign and Byzantine process faults.

BYZZBENCH allows the implementation of BFT protocols by only focusing on the protocol logic, avoiding the boilerplate code for state machine replication, message communication, and command log operations. Similarly, it provides the controlled replica communication and fault injection features available for test scenario generation algorithms. Hence, the algorithm developers can implement their algorithms on top of the intercepted messages, decoupled from the implementation of the protocol under test. Moreover, it has available correctness checkers that run on the recorded replicated logs, which check for generic consensus properties and can be extended with more checks by developers.

Although our current benchmarks and examples primarily target BFT consensus protocols, **BYZZBENCH** is designed to be applicable to any BFT protocol, including those used for state machine replication and broadcast.

2.1 Implementing BFT Protocol Benchmarks

BFT protocols establish a set of rules for message exchanges that enable a group of processes to coordinate in the existence of some arbitrary process behavior. Each message includes specific metadata related to the protocol, such as the current protocol step, which helps the receiving processes understand the protocol's state and respond appropriately. The executions of the protocols are organized as a sequence of lock-step *communication rounds* in which processes exchange messages following the protocol rules. In every round, the processes can send messages to the other processes, receive and process the messages delivered in that round, and update their local states accordingly. This process involves updating the local state of the receiving process and may also include sending, multicasting, or broadcasting new protocol messages to other processes.

BYZZBENCH models a cluster of *Replica* processes, which run the protocol for state machine replication of processing *Client* requests. The *Client* processes submit operation requests to the cluster and collect replies from the replicas. Given a set of concurrent client requests, the *Replica* processes run the BFT protocol to agree on a total order of client requests to commit in a similar fashion to Paxos or Raft protocols for state machine replication or the commitment of a total order of transactions in a blockchain system.

BYZZBENCH enables practical implementation of BFT protocol benchmarks by offering essential features such as setting up a distributed cluster, enabling high-level message communication and handling, and managing local replica logs that are typically employed in BFT protocols. The framework implements *Replica* processes using the actor model of programming [18], where each actor runs independently on their local states concurrently to each other and communicates via point-to-point message exchanges over a potentially faulty network. The local state of each *Replica* keeps the data for state machine replication, including the local *commit log*, protocol state, cluster configuration, and other meta-data.

Implementing a BFT protocol benchmark in **BYZZBENCH** requires only extending a *Replica* abstract class and implementing the initialization and message handler methods. To simplify the implementation of message handlers, **BYZZBENCH** provides an API that allows *Replicas* to exchange messages using various methods, including sending, broadcasting, and multicasting. It also includes options for setting timeouts on time-triggered messages and a common API for managing the commit log of a *Replica*, as provided in Section B.

BYZZBENCH currently provides two implementations of the PBFT protocol [11], i.e., the buggy implementation in [10, 34] along with its correct implementation. Our ongoing work extends the framework with the implementations of the BFT protocols in Appendix A.

2.2 Modeling Time

BFT protocols often employ time-triggered messages, which resend some protocol messages or propose to advance protocol steps (e.g., for electing a new leader or moving to a new view) in case of suspected failures. The nondeterminism in the time-triggered messages makes it challenging for the BFT exploration algorithms to enumerate or reproduce executions.

BYZZBENCH controls the advancement of time during the protocol execution to ensure reproducibility and enumeration. This is achieved through the *Timekeeper* component, which manages time progression within each execution. Time-triggered messages can be implemented by timeout-based method calls in the BFT replica interface to enable some events only when a timeout has been reached. The *Timekeeper* also provides methods for getting and advancing time to be used by the protocol implementations.

The test scenario generators can trigger the protocol's timeout messages by advancing the corresponding replica's time value to the designated trigger value. This allows for the controlled execution of time-based messages. The framework does not synchronize the clocks of the distributed replicas and allows their local clocks to drift apart in certain execution scenarios, as in the real-world execution of the protocols.

2.3 Modeling and Injecting Network and Process Faults

BFT protocols promise correctness even in the existence of network and Byzantine process faults. However, recent studies [1, 27, 21, 5, 26, 34] have shown that several BFT protocols fail to uphold their assumptions under these conditions. Therefore, it is crucial for a testing algorithm to generate scenarios that incorporate network faults, such as isolating specific processes or partitioning the network, as well as process faults, including benign crashes or Byzantine behaviors.

Testing algorithms for consensus protocols [14, 6, 34] inject network and process faults based on the *communication rounds* of the consensus protocols. The communication round provides an abstraction of the protocol state and the step of the execution that is useful for effectively specifying the period of faulty behaviors.

BYZZBENCH supports round-based fault injection using an extended model of predicate-based fault injection. The model describes a fault as a pair consisting of a boolean predicate and a side effect, represented as $(predicate, effect)$. The side effect *effect* on a message is applied only if the message satisfies the *predicate*. This model facilitates round-based fault injection by allowing the specification of the predicate for a particular protocol round, receiver, and sender. Moreover, it enables specifying more general conditions (e.g., not only checking the protocol round but any other information of the in-flight message or distributed cluster state) for injecting faults for certain messages.

Operationally, **BYZZBENCH** injects specific faults into the execution of a BFT protocol operating at the Transport layer. The Transport layer shown in Figure 1 maintains the in-flight messages sent to the Replicas in their mailboxes. Depending on the prescribed network faults in the scenario, it reorders the delivery of messages or drops some messages, e.g., to isolate a process from the network or partition the network by dropping the messages exchanged between partitions. Similarly, it can inject process faults by omitting, duplicating, or modifying the in-flight messages sent from a Byzantine Replica.

2.4 Implementing Testing Algorithms for BFT Protocols

BYZZBENCH enables implementing algorithms for testing and exploration of BFT protocol implementations using an *Exploration Strategy* interface. Simply, the developers implement a testing algorithm by providing fault injection predicates as explained in Section 2.3 and implement the functionality for selecting and processing the next in-flight message from the transport layer. The transport layer enforces the delivery of or fault injection into the selected in-flight message, as dictated by the exploration strategy. During the execution, the processed messages and injected faults are recorded in an execution schedule and made available for inspection after the execution.

BYZZBENCH currently provides the implementations of the following testing algorithms for BFT protocols: a naive random fault injection algorithm, which decides to deliver, drop or mutate messages at each execution step; ByzzFuzz randomized testing algorithm [34], which models Byzantine faults using round-based small-scope message mutations, and a preliminary implementation of the Twins systematic test scenario generator using twin replicas of the processes. Our ongoing work explores the relative performances of these testing algorithms in the exploration of bugs in the BFT protocol implementations.

2.5 Correctness Specification of BFT Protocol Executions

BYZZBENCH's *Property Checker* checks the correctness of BFT protocol test executions by checking the following correctness properties of BFT consensus [9]: (i) Agreement: No two processes decide differently, (ii) No correct process decides twice, (iii) Validity: A correct process may only decide a value that was proposed by a correct process, and (iv) Termination: Every correct process eventually decides on some value.

Typically, BFT protocols promise the safety properties of agreement, integrity, and validity under both synchronous and asynchronous network conditions. For the liveness property of *termination*, most protocols rely on a partially synchronous network [15] to overcome the FLP impossibility result [16] for asynchronous networks.

BYZZBENCH simulates a variant of partial synchrony, called eventual synchrony, by employing a graceful period of execution without any faults after the execution of a test scenario with some network and process faults. While eventual synchrony satisfies the network assumptions of some BFT protocols, some protocols (e.g., Tendermint [8], Sync

HotStuff [3]) rely on stronger synchronization and delivery requirements. **BYZZBENCH** leaves the enforcement of the network assumptions of protocols to the developers, which depends on the BFT protocol under test and the test generation algorithm.

BYZZBENCH checks the termination property by checking the two conditions of *deadlock* and *violation of bounded termination*. The deadlock condition occurs when all the replicas' mailboxes are empty, i.e., there are no available protocol messages to process, and the cluster has not yet reached consensus. An example of such a deadlock condition occurs in the violating execution of Fast Byzantine Consensus [25] uncovered in recent work [2]. In that execution, the protocol rules lead to a stuck network state, where the processes do not exchange any more messages and cannot make a decision. The violation of bounded termination occurs when the processes continue exchanging messages, but the communication does not lead to consensus in a predefined, bounded length of execution. The violations to bounded termination do not guarantee that the detected execution is a violation of (unbounded) termination, as it can return false positives. However, it is useful in detecting termination violations. An example violation detected by this condition is the termination violation in a previous version of the XRP Ledger [34], where some process faults lead to corrupted states of some processes, preventing them from achieving consensus despite ongoing message communication. **BYZZBENCH** checks bounded termination by checking agreement for the client requests after an execution with a bounded duration of test execution.

BYZZBENCH checks the safety properties of agreement, integrity, and validity using the replicas' commit logs and the exchanged messages between the processes, which keep the proposed values and commitment decisions.

In addition to these properties, the *Property Checker* supports specifying more expressive predicates, allowing **BYZZBENCH** to be extended with progress-related checks that are crucial for uncovering subtle liveness and fairness issues not captured by deadlock or bounded termination alone.

3 Preliminary Evaluation on PBFT

We demonstrate how **BYZZBENCH** can be used to assess the effectiveness of different testing algorithms through a preliminary evaluation on the seminal PBFT [11] protocol.

PBFT provides distributed agreement in a cluster of $3f + 1$ processes, tolerating up to f faulty processes, which can deviate arbitrarily from the protocol specification. An execution of the PBFT protocol is decomposed into views, in each of which one of the processes acts as a leader. In each view, the leader executes a sequence of client operations. For each request, the leader broadcasts a proposal, followed by two rounds of message exchanges in which the participants vote for the proposal. If a quorum agrees on the proposal, they execute the operation and reply with the result to the client. Processes store the total order of executed client operations in their message logs, and the protocol ensures that the correct processes agree on their contents.

The **BYZZBENCH** framework provides a buggy implementation of the PBFT protocol as a benchmark, seeding the same implementation errors found in the publicly available implementation, PBFT-Java [10]. The benchmark is vulnerable to agreement violations due to (i) incorrect assignment of sequence numbers in the protocol messages, (ii) incorrect processing of prepared certificates, and (iii) missing implementation of message digests.

We compare the performance of detecting the agreement violations in the benchmark using a recent testing algorithm, ByzzFuzz, compared to the naive random testing algorithm. Table 1 lists the percentage of test executions out of 1000 runs that detect violations using

■ **Table 1** Percentage of test executions (out of 1000 runs) in which random and ByzzFuzz testing strategies detected violations in PBFT-Java benchmark. ByzzFuzz results (denoted $\text{BF}(c, d)$) represent runs with c process and d network partition faults injected over 10 rounds ($r = 10$).

Random	BF(0,1)	BF(0,2)	BF(1,0)	BF(1,1)	BF(1,2)	BF(2,0)	BF(2,1)	BF(2,2)
0.0%	0.0%	0.0%	7.1%	5.4%	5.0%	11.6%	9.2%	9.5%

random testing and the ByzzFuzz algorithm. We evaluate ByzzFuzz using different algorithm parameters, i.e., $c = [0, 2]$ process faults and $d = [0, 2]$ network partition faults distributed into $r = 10$ protocol rounds of the execution. Our results show that ByzzFuzz can detect violations more frequently than the naive random testing, in alignment with the findings in the previous work [34]. Moreover, we observe that the increasing number of faults injected into the execution increases the likelihood of exposing violations.

4 Related Work

Several benchmark frameworks have been designed for evaluating the testing approaches to detect concurrency bugs [19, 22], data storage bugs [23], or benchmark programs in specific programming languages [12, 20, 24, 35]. However, none of these frameworks target Byzantine fault tolerance benchmarks and Byzantine fault tolerance bugs.

Previous work on BFT protocol benchmarks targets performance benchmarking under various workloads, computation power, and network configurations. BFTSim [31] offers a network simulator to evaluate protocol performance based on a range of network conditions. BFT-Bench [17] evaluates and compares the performance of BFT protocols under faulty network behaviors and workloads. BLOCKBENCH [13] targets private blockchains, allowing fine-grained testing of the blockchain system to evaluate their performance under smart contract workloads. BFTDiagnosis [33] is an automated fault injection framework for the security evaluation of BFT protocols. Bedrock [4] explores the trade-offs between different design space dimensions to determine the protocol that best meets application needs.

Unlike the existing work, which focuses on the performance of BFT protocols, **BYZZBENCH** targets the correctness of BFT protocols and provides a framework to evaluate the performances of exploration algorithms for detecting bugs in BFT protocol implementations.

5 Conclusions

We presented **BYZZBENCH**, an extensible benchmark framework for evaluating the testing algorithms for Byzantine fault tolerance bugs. The framework offers practical implementation of BFT protocols and provides functionality for developing testing algorithms to control the time, network, and process fault nondeterminism in the executions of test scenarios.

BYZZBENCH lays the groundwork for benchmarking in this space, although it currently has a limited set of built-in benchmark protocols and testing algorithms. At present, the available benchmark applications include both correct and faulty implementations of PBFT, and the built-in testing algorithms consist of a basic random tester, ByzzFuzz, and an initial version of Twins. Our ongoing efforts aim to expand the range of implemented BFT protocols, particularly those with known bugs, and to integrate state abstraction mechanisms.

The long-term objective of this work is to develop a comprehensive, publicly accessible benchmark suite of BFT protocol implementations to assess the effectiveness of testing and exploration algorithms for BFT protocols.

References

- 1 Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. Revisiting Fast Practical Byzantine Fault Tolerance, December 2017. [arXiv:1712.01367](#).
- 2 Ittai Abraham, Guy Gueta, Dahlia Malkhi, and Jean-Philippe Martin. Revisiting Fast Practical Byzantine Fault Tolerance: Thelma, Velma, and Zelma, January 2018. Comment: [arXiv admin note: text overlap with arXiv:1712.01367](#). [arXiv:1801.10022](#).
- 3 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync HotStuff: Simple and Practical Synchronous State Machine Replication. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 106–118, San Francisco, CA, USA, May 2020. IEEE. doi:10.1109/SP40000.2020.00044.
- 4 Mohammad Javad Amiri, Chenyuan Wu, Divyakant Agrawal, Amr El Abbadi, Boon Thau Loo, and Mohammad Sadoghi. The Bedrock of Byzantine Fault Tolerance: A Unified Platform for BFT Protocols Analysis, Implementation, and Experimentation. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 371–400, Santa Clara, CA, April 2024. USENIX Association. URL: <https://www.usenix.org/conference/nsdi24/presentation/amiri>.
- 5 Ignacio Amores-Sesar, Christian Cachin, and Jovana Micic. Security Analysis of Ripple Consensus. In Quentin Bramas, Rotem Oshman, and Paolo Romano, editors, *24th International Conference on Principles of Distributed Systems, OPODIS 2020, December 14–16, 2020, Strasbourg, France (Virtual Conference)*, volume 184 of *LIPIcs*, pages 10:1–10:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.OPODIS.2020.10.
- 6 Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, Zekun Li, Avery Ching, and Dahlia Malkhi. Twins: BFT Systems Made Robust. In Quentin Bramas, Vincent Gramoli, and Alessia Milani, editors, *25th International Conference on Principles of Distributed Systems, OPODIS 2021, December 13–15, 2021, Strasbourg, France*, volume 217 of *LIPIcs*, pages 7:1–7:29. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.OPODIS.2021.7.
- 7 Christian Berger, Hans P. Reiser, and Alysson Bessani. Making reads in BFT state machine replication fast, linearizable, and live. In *40th International Symposium on Reliable Distributed Systems, SRDS 2021, Chicago, IL, USA, September 20–23, 2021*, pages 1–12. IEEE, 2021. Comment: 12 pages, to appear in the 40th IEEE Symposium on Reliable Distributed Systems (SRDS). doi:10.1109/SRDS53918.2021.00010.
- 8 Ethan Buchman. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains*. PhD thesis, University of Guelph, 2016.
- 9 Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. Ed.)*. Springer, 2011. doi:10.1007/978-3-642-15260-3.
- 10 Johnny Cao. A Practical Byzantine Fault Tolerance (PBFT) emulator written in Java, 2020. <http://github.com/caojohnny/pbft-java>.
- 11 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In Margo I. Seltzer and Paul J. Leach, editors, *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22–25, 1999*, pages 173–186. USENIX Association, 1999. URL: <https://dl.acm.org/citation.cfm?id=296824>.
- 12 Cristina Cifuentes, Christian Hoermann, Nathan Keynes, Lian Li, Simon Long, Erica Mealy, Michael Mounteney, and Bernhard Scholz. BegBunch: Benchmarking for C bug detection tools. In *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, pages 16–20, Chicago Illinois, June 2009. ACM. doi:10.1145/1555860.1555866.
- 13 Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. BLOCKBENCH: A Framework for Analyzing Private Blockchains. In Semih Salihoglu,

- Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1085–1100. ACM, 2017. Comment: 16 pages. doi: 10.1145/3035918.3064033.
- 14 Cezara Dragoi, Constantin Enea, Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Filip Niksic. Testing consensus implementations using communication closure. *Proc. ACM Program. Lang.*, 4(OOPSLA):210:1–210:29, 2020. doi:10.1145/3428278.
 - 15 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988. doi:10.1145/42282.42283.
 - 16 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985. doi: 10.1145/3149.214121.
 - 17 Divya Gupta, Lucas Perronne, and Sara Bouchenak. BFT-Bench: Towards a Practical Evaluation of Robustness and Effectiveness of BFT Protocols. In Márk Jelasity and Evangelia Kalyvianaki, editors, *Distributed Applications and Interoperable Systems - 16th IFIP WG 6.1 International Conference, DAIS 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9687 of *Lecture Notes in Computer Science*, pages 115–128. Springer, 2016. doi:10.1007/978-3-319-39577-7_10.
 - 18 Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence, IJCAI'73*, pages 235–245, San Francisco, CA, USA, August 1973. Morgan Kaufmann Publishers Inc.
 - 19 Nicholas Jalbert, Cristiano Pereira, Gilles Pokam, and Koushik Sen. Radbench: a concurrency bug benchmark suite. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism, HotPar'11*, page 2, USA, 2011. USENIX Association. URL: <https://www.usenix.org/conference/hotpar-11/radbench-concurrency-bug-benchmark-suite>.
 - 20 René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, San Jose CA USA, July 2014. ACM. doi:10.1145/2610384.2628055.
 - 21 Minjeong Kim, Yujin Kwon, and Yongdae Kim. Is Stellar As Secure As You Think? In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 377–385, Stockholm, Sweden, June 2019. IEEE. doi:10.1109/EuroSPW.2019.00048.
 - 22 Ziyi Lin, Darko Marinov, Hao Zhong, Yuting Chen, and Jianjun Zhao. JaConTeBe: A benchmark suite of real-world java concurrency bugs (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 178–189, November 2015. doi:10.1109/ASE.2015.87.
 - 23 Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. BugBench: Benchmarks for Evaluating Bug Detection Tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.
 - 24 Fernanda Madeiral, Simon Urli, Marcelo de Almeida Maia, and Martin Monperrus. BEARS: an extensible java bug benchmark for automatic program repair studies. In Xinyu Wang, David Lo, and Emad Shihab, editors, *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, pages 468–478. IEEE, 2019. doi:10.1109/SANER.2019.8667991.
 - 25 J.-P. Martin and L. Alvisi. Fast Byzantine Consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, July 2006. doi:10.1109/TDSC.2006.35.
 - 26 Lara Mauri, Stelvio Cimato, and Ernesto Damiani. A formal approach for the analysis of the XRP ledger consensus protocol. In Steven Furnell, Paolo Mori, Edgar R. Weippl, and Olivier Camp, editors, *Proceedings of the 6th International Conference on Information Systems*

- Security and Privacy, ICISSP 2020, Valletta, Malta, February 25-27, 2020*, pages 52–63. SCITEPRESS, 2020. doi:10.5220/0008954200520063.
- 27 Atsuki Momose. Force-Locking Attack on Sync Hotstuff. *IACR Cryptol. ePrint Arch.*, page 1484, 2019. URL: <https://eprint.iacr.org/2019/1484>.
 - 28 João Miguel Louro Neto. ByzzBench: A benchmark framework for byzantine fault tolerance testing algorithms, 2024. <https://github.com/joaomlneto/byzzbench>. URL: <https://github.com/joaomlneto/byzzbench>.
 - 29 David Schwartz, Noah Youngs, and Arthur Britto. The Ripple Protocol Consensus Algorithm. *Ripple Labs Inc White Paper*, 2014.
 - 30 Nibesh Shrestha, Mohan Kumar, and SiSi Duan. Revisiting hBFT: Speculative Byzantine Fault Tolerance with Minimum Cost, April 2019. Comment: 4 pages. [arXiv:1902.08505](https://arxiv.org/abs/1902.08505).
 - 31 Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. BFT protocols under fire. In Jon Crowcroft and Michael Dahlin, editors, *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*, pages 189–204. USENIX Association, 2008. URL: http://www.usenix.org/events/nsdi08/tech/full_papers/singh/singh.pdf.
 - 32 The Diem Team. DiemBFT v4: State Machine Replication in the Diem Blockchain, 2021.
 - 33 Jitao Wang, Bo Zhang, Kai Wang, Yuzhou Wang, and Weili Han. BFTDiagnosis: An automated security testing framework with malicious behavior injection for BFT protocols. *Computer Networks*, 249:110404, July 2024. doi:10.1016/j.comnet.2024.110404.
 - 34 Levin N. Winter, Florena Buse, Daan de Graaf, Klaus von Gleissenthall, and Burcu Kulahcioglu Ozkan. Randomized Testing of Byzantine Fault Tolerant Algorithms. *Proc. ACM Program. Lang.*, 7(OOPSLA1):757–788, 2023. doi:10.1145/3586053.
 - 35 Ting Yuan, Guangwei Li, Jie Lu, Chen Liu, Lian Li, and Jingling Xue. Gobench: A benchmark suite of real-world go concurrency bugs. In Jae W. Lee, Mary Lou Soffa, and Ayal Zaks, editors, *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, pages 187–199. IEEE, 2021. doi:10.1109/CGO51591.2021.9370317.

A Benchmark BFT Protocols

Our ongoing work extends the currently available BFT protocol benchmarks in [BYZZBENCH](#) with implementations of BFT protocols with known protocol and implementation bugs from the literature, as summarized in Table 2. The table shows the discovery year of the bug, the violation type, and the source of the bug (i.e., whether it is a protocol, implementation, or configuration bug). It also lists the number of processes, protocol views (or ledger blocks), and protocol rounds with process and network faults in the violating execution, along with the reference to the study reporting the violation.

B API for Implementing BFT Protocols

Listings 1 and 2 show the *Replica Interface* for implementing BFT protocols and the API available for the *Replicas* for exchanging messages, setting timeout-based messages, and committing operations to the replicated log.

■ **Table 2** BFT protocol benchmarks with known bugs in their design or implementation.

Year	Protocol	Violation	Bug Source	Execution Parameters				Reference
				#processes	#views (or blocks)	#process faults	#network faults	
2017	FaB	liveness	protocol	4	2	1	2	[1]
2017	Zyzyva	safety	protocol	4	3	1	4	[1]
2019	hBFT	safety	protocol	4	2	2	2	[30]
2020	Sync HotStuff	safety	protocol	5	3	2	7	[27]
2020	XRPL	liveness	trust config	7	2	1	0	[5, 34]
2020	XRPL	safety	trust config	7	2	2	0	[5, 34]
2021	PBFT	liveness	protocol	4	1	2	0	[7]
2022	Fast-HotStuff	safety	protocol	4	11	0	3	[6]
2023	PBFT	safety	pbft-java	4	2	1	0	[34]
2023	XRPL	liveness	rippled v1.7.2	7	3	1	0	[34]

■ **Listing 1** Replica Interface in [BYZZBENCH](#), for protocol implementations.

```
// Initialize this Replica
void initialize();

// Handle a message received by this Replica
void handleMessage(String sender, MessagePayload message);
```

■ **Listing 2** API available for Replicas in [BYZZBENCH](#).

```
// Messaging API
void sendMessage(MessagePayload message, String recipient);
public void broadcastMessage(MessagePayload message);
public void multicastMessage(MessagePayload message, SortedSet<String> recipients);

// Timeouts
long setTimeout(String name, Runnable r, Duration timeout);
void clearTimeout(long eventId);
void clearAllTimeouts();

// Commit Log
void commitOperation(LogEntry operation);

// Time
Instant getCurrentTime();
```


Scar: Verification-Based Development of Smart Contracts

Jonas Schiff¹  

KASTEL - Institute of Information Security and Dependability, Karlsruhe Institute of Technology, Germany

Bernhard Beckert  

KASTEL - Institute of Information Security and Dependability, Karlsruhe Institute of Technology, Germany

Abstract

Compared to other kinds of computer programs, smart contracts have some unique characteristics, *e.g.*, immutability, and the public availability of source code. This means that any vulnerability has a large probability of being exploited. Since smart contracts cannot be patched, it is very important that smart contracts are correct and secure upon deployment. While much research has been invested in this goal, smart contract correctness and security remains a challenging problem.

In this paper, we present the SCAR approach for model-driven development of correct and secure smart contract applications. Before implementing an application, smart contract developers first describe it in terms of an intuitive, platform-agnostic metamodel. Within this model, they can also specify high-level security and behavioral correctness properties, and check whether the model contains any inconsistencies. Finally, a combination of code generation, static analyses, and formal verification of automatically generated formal annotations leads to an implementation that is correct and secure w.r.t. the initial model.

2012 ACM Subject Classification Software and its engineering → Formal methods

Keywords and phrases Smart Contracts, Formal Verification, Security, Safety and Liveness

Digital Object Identifier 10.4230/OASICS.FMBC.2025.14

Category Tool Paper

Funding This work was supported by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs.

1 Introduction

Static analysis and formal verification of smart contracts have been very active fields of research. Numerous approaches for detecting bugs and common weaknesses as well as specifying and verifying user-defined correctness properties have been proposed; a recent overview lists 194 works [9]. However, smart contract security is still a problem. There are entire websites dedicated to listing all the instances of smart contract applications being hacked (for example, [16] and [4]). Many of the exploited applications were even audited, prior to the attacks, by companies like CertiK [1], who perform audits consisting of both manual code review and automated static analysis. This indicates that the methods used in the audits are not sufficient to guarantee security.

Although static analysis tools may help to avoid common vulnerabilities, current research shows that they are not sufficient to ensure smart contract application security. For example, Zhang et. al. come to the conclusion that the majority of bugs in smart contracts is “hard to find” and “not machine-auditable” [17]. Only 8% of the vulnerabilities which led to these

¹ Corresponding author



© Jonas Schiff and Bernhard Beckert;

licensed under Creative Commons License CC-BY 4.0

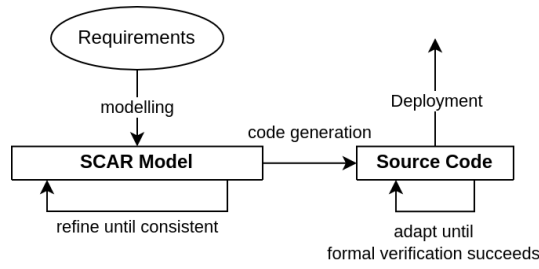
6th International Workshop on Formal Methods for Blockchains (FMBC 2025).

Editors: Diego Marmosier and Meng Xu; Article No. 14; pp. 14:1–14:13

OpenAccess Series in Informatics



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



■ **Figure 1** The SCAR process of smart contract application development.

attacks could have been detected with *any* state-of-the-art static analysis tool [2]. Moreover, most current vulnerability detection tools “can only detect vulnerabilities in a single and old version of smart contracts” [6]. Formal verification tools go a different route by allowing their users to define the desired properties, and attempting a proof. Unlike with vulnerability detection tools, there is not always a yes or no answer; a proof attempt can always time out. If a proof is found, the program is deemed to fulfill its specification. Examples for formal verification approaches include SOLC-VERIFY [5], CELESTIAL [3], or VERX [10].

One difficulty with formal verification on the source code level is a lack of abstraction. Some properties, *e.g.*, resource ownership, are easy to grasp for developers, but hard to specify in terms of invariants or pre- and postconditions. To allow more intuitive specification of such properties, model-driven development approaches have been proposed, such as FSolidM [7] or SmartPulse [15]. These approaches target specific platforms and specific properties (*e.g.*, SmartPulse targets temporal properties of Solidity smart contracts).

In this paper, we present SCAR, a highly general approach for modeling smart contract applications. The SCAR approach consists of three main parts:

- A **metamodel of smart contract applications**, in which these applications can be defined in terms of their structure, *i.e.*, the contracts, state, and functions they consist of. A type system is also part of the metamodel.
- The **ScarML specification language**, which serves to describe the behavior of individual functions in terms of pre- and postconditions, and properties of individual contracts in terms of invariants.
- A **model-driven process** based on formal verification, which leads from the creation of a SCAR model to a source code implementation that can be deployed on a specific platform. Formal methods ensure that the implementation is consistent with the model.

This paper is structured as follows: In Section 2, we describe the structure and the semantics of the SCAR metamodel and the SCARML specification language. Section 5 gives an overview of the applications of SCAR, of its Scala implementation, and about our evaluation. Section 6 concludes.

2 The Scar Approach

In this section, we present SCAR, a modeling and analysis approach for smart contract applications.

An overview of the intended workflow with the SCAR platform is given in Figure 1. After collecting the requirements for an application, developers create a basic SCAR model. This basic model defines an application in terms of its smart contracts, their state variables and functions, and the behavior of these functions. The developer then proceeds in two directions:

First, they specify application-level correctness and security properties on the model. They conduct the analyses provided by SCAR (or by the integration of SCAR with other tools) to verify that these properties are consistent with the basic model. Secondly, they use the code generation capabilities provided by SCAR to automatically create a source code skeleton in the desired target programming language, annotated with specification in the language of a formal verification tool suitable for that language. They proceed to implement the functions such that the verification tool is able to prove that the implementation is correct w.r.t. the generated specification. This process is inspired by the Design by Contract paradigm [8].

Figure 2 gives an overview of the **basic metamodel**. The topmost element of the metamodel is the *application*. It consists of *contract* elements. Contracts, in turn consist of *stateVariable* and *function* elements. State variables have a name and a type. Functions have a name, a list of named and typed parameters, an optional return type, and a function contract written in the SCARML functional specification language (see Section 3 below).

In order to describe the initialization of an application, an initial condition can be specified for each contract in the same specification language. To closely reflect the constructors in smart contract programming languages, the initial condition can be parameterized. The full grammar for expressing a SCAR model in text is given in Figure 9.

The SCAR type system has four primitive types: Booleans (*Bool*), signed and unsigned integers (*Int* and *UInt*, respectively), and strings (*String*). Accounts (*Account*) can be either external accounts, representing a real-world entity, or contracts. Both have a non-negative balance, and every account has a unique ID. Each contract in an application defines its own type, which consists of a name, a list of typed and named state variables, and a list of the functions of that contract. Furthermore, there are two composite types: Arrays and Mappings, which map keys of a primitive type to values (whose type is not restricted). In addition, there are two user-defined types: Enums have a *String* list of constant names. Structs are records with a list of named and typed fields. The type system is visualized in Figure 5.

For specifying the behavior of an application, we introduce **ScarML**, the SCAR modeling language. SCARML is a specification language for contract invariants (for specifying properties of smart contract instances) and function contracts (for specifying the behavior of single functions). Function contracts consist of preconditions, postconditions, and frame conditions. Invariants, preconditions, and postconditions are defined as Boolean *specification expressions*. Frame conditions are specified as *frame expressions*.

The full grammar of SCARML specification expressions is defined in Figure 7. Expressions are typed. Terms of Boolean type can be negated and combined with the standard Boolean connectors. Existentially and universally quantified expressions are also allowed. There are two equality operators, reflecting value equality and referential equality.

There is a special operator for evaluating a term in the pre-state of a transaction (`\old`). Furthermore, there are some constructs that represent the context of a transaction, *i.e.*, the caller, the amount transferred with the call, the system time of the call, and the number of the block in which the transaction is executed. Another domain-specific term is the `\hash` operator. Furthermore, the specification can contain terms representing a (pure) function call. All Boolean, integer, and string literals are also terms. Terms of Boolean type can be used as top-level expressions in invariants, preconditions and postconditions.

3 Semantics

We define the **semantics of ScarML** in terms of an evaluation function over the state. Invariants and pre- and postconditions are evaluated in a *step* τ_i of the execution, which consists of the call context ctx_i and a state s_i (cf. the description of the trace semantics below). The **partial evaluation function** $\llbracket \cdot \rrbracket$ maps terms of the SCARML specification language to the set of all possible values. It is parameterized with the step τ_i in which it occurs. As an example, the evaluation of a variable is that variable's state in the context where it is evaluated: $\llbracket v \rrbracket := s_i(v)$. The evaluation of mathematical and logical operators is usually just the operation, conducted on the evaluation of the subterms (e.g., $\llbracket \phi_1 + \phi_2 \rrbracket := \llbracket \phi_1 \rrbracket + \llbracket \phi_2 \rrbracket$). The full definition of $\llbracket \cdot \rrbracket$ is given in Figure 8.

We say that a condition c is **fulfilled** (or satisfied) in a step τ_i if $\llbracket c \rrbracket_{\tau_i} = \top$. A condition c is **satisfiable** if there exists a possible step (i.e., a combination of a state s and a call environment) τ such that c is fulfilled in τ .

As for the **semantics of a Scar model**, we say that a SCAR model m is **plausible** if

- all contract invariants are satisfiable, and
- all function pre- and postconditions are satisfiable, and
- the initial conditions of each contract type in m are satisfiable, and
- for each contract type in m , the initial conditions of m imply each contract invariant, i.e., every possible state which satisfies the initial conditions must also satisfy every contract invariant.

For a given SCAR model, an execution of the application is an infinite trace of steps $\tau_0, \tau_1, \tau_2, \dots$ where each step τ_i is either an *environment step* or an *application step*.

The **initial step** τ_0 is defined in the `appInit` part of the application specification. In it, the initial set of contracts is declared, along with a list of parameters according to the initial parameter types that each contract requires. After the initial step, the set of known contracts \mathcal{C} consists of all contracts declared in `appInit`, as well as all contract-typed locations within these (i.e., state variables of contract type, as well as contract-typed elements of arrays, mappings, and structs).

Similarly, the set of known external accounts is initialized with all account-typed state variables and account-typed elements of arrays, mappings, and structs.

For τ_0 to be valid, the initial condition of each contract in \mathcal{C} must be fulfilled.

An **environment step** characterizes a step in which no function of the application is called, and in which the application state does not change, with the possible exception of account balances. Furthermore, block number and system time can increase.

Formally, an environment step τ_i consists of ctx_i , the context of the step (which, in turn, consists of $sysTime_i$, the system time of the step, and $blockNum_i$, the block number of the step), and s_i , the state of the application after τ_i .

For an environment step τ_i to be **valid**, the following conditions must hold:

- $blockNum_i = blockNum_{i-1} \vee blockNum_i = blockNum_{i-1} + 1$
- $sysTime_i = sysTime_{i-1}$ if $blockNum_i = blockNum_{i-1}$
- $sysTime_i > sysTime_{i-1}$ if $blockNum_i = blockNum_{i-1} + 1$
- $\forall l \in Locs : \begin{cases} s_i >= s_{i-1}, l \text{ is balance} \\ s_i = s_{i-1}, \text{ else} \end{cases}$

An **application step** is the result of a call to one of the application's functions. Formally, an application step τ_i consists of ctx_i and s_i as above, as well as f_i , the function that was called to reach τ_i . f_i , in turn, consists of $caller_i$, the caller of the function; amt_i , the amount

transferred with the function call; $params_i$, the call parameters; pre_i , the precondition of f_i ; $post_i$, the postcondition of f_i ; $return_i$, the return value of f_i ; and mod_i , the set of locations f_i may modify.

For an application step τ_i to be **valid**, the following conditions must hold:

- $blocknum_i = blocknum_{i-1} \vee blocknum_i = blocknum_{i-1} + 1$
- $systime_i = systime_{i-1}$ if $blocknum_i = blocknum_{i-1}$
- $systime_i > systime_{i-1}$ if $blocknum_i = blocknum_{i-1} + 1$
- pre_i must be fulfilled in τ_{i-1}
- $post_i$ must be fulfilled in τ_i
- $\forall l \in Locs \setminus mod_i : \begin{cases} s_i \geq s_{i-1}, l \text{ is balance} \\ s_i = s_{i-1}, \text{ else} \end{cases}$

With this, we now define the semantics \mathcal{S} of a SCAR model as the set of all infinite traces of valid steps: For a plausible SCAR model m , the semantics $\mathcal{S}(m)$ is the set of traces $\{T := \tau_0, \tau_1, \tau_2, \dots\}$ where τ_0 is a valid initial step for m , and each $\tau_i \in \mathcal{N}_{>0}$ is either a valid environment step or a valid application step. A visualization of the SCAR trace semantics is presented in Figure 4.

4 Ensuring Consistency between Model and Code

In order to deploy an application, a model needs to be converted into platform-specific source code. The main point of our model-driven approach is that properties that were proven to hold for the model also hold for the implementation. Therefore, the conversion from model to code needs to be semantics-preserving: The implementation must be a refinement of the model. This means that for every possible execution of the implementation, when deployed, there must be a corresponding trace in the model semantics.

To show that this is the case, the semantics of the platform and of the verification tool have to be taken into account, and it has to be shown that each execution of a deployed application corresponds to an execution trace of its model. This argument has to be made separately for each target platform.

For the Solidity programming language, code is generated as follows from a SCAR model: First, a Solidity version header is prepended to the file. Then, a library contract `UTIL` containing the user-defined data types is created. SCAR structs and enums are translated to their respective Solidity counterparts. For every contract in the SCAR model, a corresponding Solidity `contract` of the same name is created. Since SCAR was written with Solidity as the main target language, each SCAR type has a directly corresponding Solidity type, making the translation of state variables, function parameters, and return types straightforward. For functions, only headers are generated, which consist of the function name, the translated function parameters, the `public` modifier indicating that the function can be called from everywhere, as well as the return type.

For verification, we use the SOLC-VERIFY tool [5] for formal verification of Solidity smart contracts. As in SCAR, the main specification elements in SOLC-VERIFY are invariants and function contracts.

The differences between SCAR and Ethereum/Solidity are mainly in the area of data types. SCAR supports fewer types than Solidity. Therefore, the Solidity implementation has to be limited to the variables that actually occur in the model to ensure consistency.

However, the SCAR types contain mathematical integers, which are not present in Solidity; the SCAR type system is not a subset of the Solidity type system. Therefore, care has to be taken to avoid situations where this discrepancy leads to a violation of consistency. To give

an example, this could happen if a precondition is not satisfiable over mathematical integers, but satisfiable over Solidity's machine integers. This would lead to a situation where the function could never occur in the semantics of the model, but it could be called successfully in the implementation. Thus, the implementation would allow more behaviors than the model. This needs to be avoided by the translation, or by additional measures or precautions developers have to take.

In case of the SCAR to Solidity code and annotation generation we developed, the implementation is consistent if the generated source code is not changed except for implementing the generated functions and adding helper functions (*i.e.*, functions that cannot be called from outside their contract and which do not change the state in any way). Furthermore, the generated function preconditions and contract invariants must not contain arithmetic overflows, and the implementation must be proven correct by SOLC-VERIFY against the generated specification (function contracts and contract invariants).

With this, it is guaranteed that the original model was syntactically correct, and that the application state cannot be changed outside of the behavior specified in the model. Thus, every execution of the implementation is also a trace of the model: Model and implementation are consistent.

5 Application and Evaluation

SCAR allows specification and verification of application-level properties of smart contracts. These properties are difficult to specify on the source code level. On the level of a SCAR model, however, the relevant specification constructs can be easily introduced in a way that keeps specification concise and intuitive. We have previously developed two approaches for specifying and verifying application-level properties, namely, temporal properties, and access control policies.

In [12], we developed a specification language for temporal properties of smart contract applications. The properties can be specified and verified directly on the model level. If there is an implementation consistent with the model (as described in Section 4), then it also fulfills the temporal properties specified on the model.

In [13], we developed a security approach for smart contract applications, based on the notion of actors and capabilities. Using this approach in SCAR, developers can define roles and capabilities on the model level. Capabilities include calling a certain functionality, changing the state of an application, and transferring currency. Our approach makes it possible to specify security policies on the abstract model level, which matches the intuition much closer than the source code level. As with temporal properties, the adherence of a model to a given security specification can be verified purely on the model level, given that the implementation is consistent (in the above sense) with the model.

SCAR has been implemented in the Scala programming language. The repository [11] is publicly available. The project's core is the `model` package, in which all model elements are defined as Scala classes. The top-level model element is the smart contract application `SCApp`. The syntax of the `.scar` model is defined in an ANTLR grammar. A simplified overview grammar for the basic metamodel is shown in Figure 9. First, the user-defined types are specified, followed by the initial configuration of the application. Then, the contracts are declared with their state variables, initial conditions, invariants, and functions (including function contracts).

We evaluated the SCAR approach on a set of example applications, including a bank, an escrow, an auction, and a casino contract, as well as the more complex Palinodia application [14]. While this does not constitute a full evaluation, some conclusions can still be drawn:

First of all, SCAR is general enough: Its data types and structural model elements are sufficient to model all desired applications and their basic functionality.

Second, even though the set of examples is not large, almost all syntactic elements of the SCAR specification language appear in at least one of the examples. From this, it can be concluded that the modeling language is not overly complex.

Another metric for assessing the quality of a metamodel is how succinct the models are, compared to some reference. In Figure 12, the length of the SCAR models of the examples described in this section is compared to the length of the corresponding Solidity code. This comparison should be taken with a grain of salt; it is easy to change the relative length by leaving out or specifying additional properties in the SCAR model.

When comparing the length of the SCAR models and the Solidity source code the models are based on, it can be observed that the models are always shorter, but often not by much. For very simple functionality, such as assignments or transfers, the SCAR specification and the corresponding Solidity code are practically identical. In such cases, SCAR's approach leads to a duplication, and might increase the workload of a developer. On the other hand, functions with such an easy specification can also be implemented without much effort, and in the future, SCAR will incorporate functionality for automatic program synthesis, or correct-by-construction code generation.

SCAR's benefits become more obvious when the modeled functionality becomes more complex. This includes properties of array and mapping data types, which can be succinctly specified in SCAR. Another recurring case was the use of an array as an index data structure to a mapping, which happens both in the simple bank example and in the real-world Palinodia application. Implementing this functionality is rather complex, requiring tens of lines of code. The abstract property describing the intended behavior, however, is very simple to describe in both cases, and this is reflected in the succinctness of the SCAR model.

As for verification, we modeled all applications in SCAR, generated a Solidity code skeleton from the model, and attempted to implement it so that verification with the SOLC-VERIFY tool succeeded. This was possible in most cases. Failures include functionality which relies on hashing; while our implementation is likely correct, SOLC-VERIFY is unable to reason about the Keccak hash function used in Solidity. In our examples, the implementation did not require auxiliary specification such as loop invariants.

6 Conclusion and Future Work

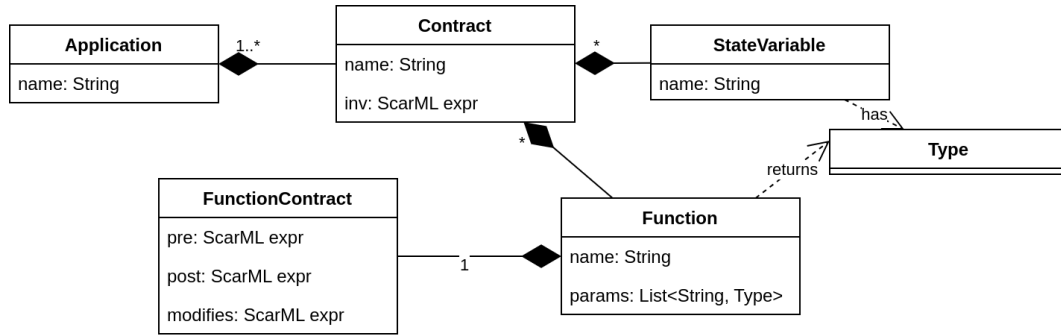
In this paper, we presented the SCAR approach for model-driven development of correct and secure smart contract applications. Our approach enables developers to model an application and its behavior in an abstract, intuitive manner. At the same time, it provides a process, based on formal verification, which guarantees that the implementation is consistent with the model.

There are several future research directions. One is the application of the SCAR approach to other platforms, languages, and verification tools. We are currently developing methods to apply SCAR to Rust smart contracts written for the Solana platform.

Other possible research directions include simulation, and generating runtime checks. SCAR's state transition semantics make it very suitable for the execution with random parameters, and the relevant parts of the application state can easily be highlighted for visualization. This can help developers to quickly build an intuition whether their model actually captures their intention. In cases where formal verification is not successful, runtime checks can be generated from SCARML specification, to ensure that the application fails gracefully in the presence of programming errors.

References

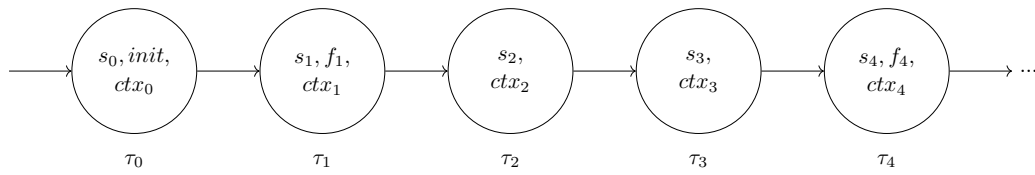
- 1 CertiK. CertiK - Securing The Web3 World, 2024. URL: <https://www.certik.com>.
- 2 Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Ben Livshits. Smart Contract and DeFi Security: Insights from Tool Evaluations and Practitioner Surveys, April 2023. doi:10.48550/arXiv.2304.02981.
- 3 Samvid Dharanikota, Suvam Mukherjee, Chandrika Bhardwaj, Aseem Rastogi, and Akash Lal. Celestial: A Smart Contracts Verification Framework. In *2021 Formal Methods in Computer Aided Design (FMCAD)*, pages 133–142, October 2021. doi:10.34727/2021/ISBN.978-3-85448-046-4_22.
- 4 David Gerard. Attack of the 50 Foot Blockchain, June 2024. URL: <https://davidgerard.co.uk/blockchain/>.
- 5 Akos Hajdu and Dejan Jovanovic. Solc-verify: A Modular Verifier for Solidity Smart Contracts. In Supratik Chakraborty and Jorge A. Navas, editors, *Verified Software. Theories, Tools, and Experiments*, pages 161–179, Cham, 2020. Springer International Publishing.
- 6 Daojing He, Rui Wu, Xinji Li, Sammy Chan, and Mohsen Guizani. Detection of Vulnerabilities of Blockchain Smart Contracts. *IEEE Internet of Things Journal*, pages 1–1, 2023.
- 7 Anastasia Mavridou and Aron Laszka. Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach. In Sarah Meiklejohn and Kazue Sako, editors, *Financial Cryptography and Data Security*, pages 523–540, Berlin, Heidelberg, 2018. Springer. doi:10.1007/978-3-662-58387-6_28.
- 8 B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, October 1992.
- 9 Sundas Munir and Walid Taha. Pre-deployment Analysis of Smart Contracts – A Survey, January 2023. doi:10.48550/arXiv.2301.06079.
- 10 Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. VerX: Safety Verification of Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1661–1677, May 2020. doi:10.1109/SP40000.2020.00024.
- 11 Jonas Schiff. Jonas Schiff / Scar · GitLab, June 2024. URL: <https://gitlab.kit.edu/jonas.schiff/Scar>.
- 12 Jonas Schiff and Bernhard Beckert. A Practical Notion of Liveness in Smart Contract Applications. In *OASICS FMBC 2024*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/OASICS.FMBC.2024.8.
- 13 Jonas Schiff, Alexander Weigl, and Bernhard Beckert. Static Capability-based Security for Smart Contracts. In *2023 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*, Athens, July 2023.
- 14 Oliver Stengele, Andreas Baumeister, Pascal Birnstill, and Hannes Hartenstein. Access Control for Binary Integrity Protection using Ethereum. In *Proceedings of the 24th ACM Symposium on Access Control Models and Technologies, SACMAT '19*, pages 3–12, New York, NY, USA, May 2019. Association for Computing Machinery. doi:10.1145/3322431.3325108.
- 15 Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. SmartPulse: Automated Checking of Temporal Properties in Smart Contracts. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 555–571, May 2021. doi:10.1109/SP40001.2021.00085.
- 16 Molly White. Web3 is Going Just Great, February 2024. URL: <https://web3isgoinggreat.com/>.
- 17 Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. Demystifying Exploitable Bugs in Smart Contracts. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 615–627, May 2023. doi:10.1109/ICSE48619.2023.00061.



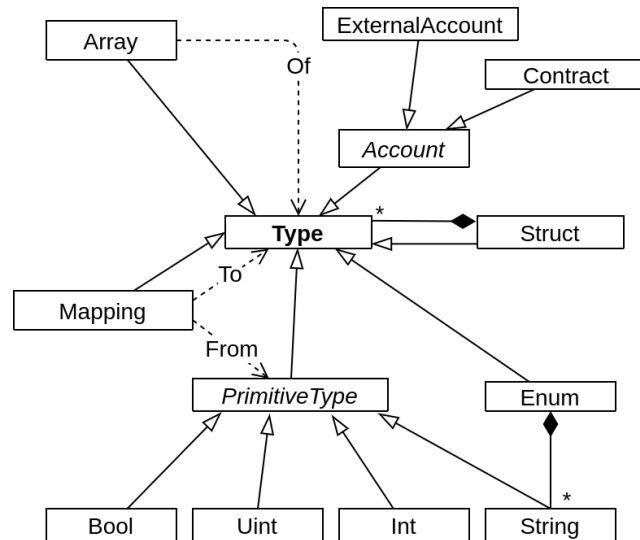
■ **Figure 2** The core metamodel. The Type hierarchy is presented in Figure 5. Function contracts and invariants are expressions in the SCARML functional specification language described in Figure 7.

$$\text{frameExpr} ::= \text{modifies} : id \mid id.e \mid id.* \mid id[e] \mid id[e1..e2] \mid id[*]$$

■ **Figure 3** The grammar for SCARML frame condition expressions.



■ **Figure 4** An example execution trace. Steps τ_3 and τ_4 are environment steps.



■ **Figure 5** A graphical description of the type system.

$Expr$	$\llbracket Expr \rrbracket_{loc}$
$\backslash nothing$	\emptyset
id	$\{id\}$
$id.e$	$(\llbracket id \rrbracket_{loc}, \llbracket e \rrbracket_{loc})$
$c.*$	$\llbracket c \rrbracket_{loc} \times statevars(c)$
$s.*$	$\llbracket c \rrbracket_{loc} \times fields(c)$
$id[e]$	$(\llbracket id \rrbracket_{loc}, \llbracket e \rrbracket_{loc})$
$m(*)$	$\llbracket a \rrbracket_{loc} \times dom(m)$
$m[*]$	$\llbracket a \rrbracket_{loc} \times ValsOf(t_1)$
$a[e1..e2]$	$\llbracket a \rrbracket_{loc} \times \{n \in \mathbb{N} \mid \llbracket e1 \rrbracket \leq n \wedge n < \llbracket e2 \rrbracket\}$
$a(*)$	$\llbracket a \rrbracket_{loc} \times dom(a)$
$a[*]$	$\llbracket a \rrbracket_{loc} \times \mathbb{N}$

■ **Figure 6** Definition of the function frame evaluation function $\llbracket \cdot \rrbracket_{loc}$, where c is a contract, s is a struct, m is a mapping from type t_1 to t_2 and a is an array.

```

 $\phi ::= v \mid v[t] \mid v.id$ 
       $\mid [1-9][0-9]^+$ 
       $\mid \neg \phi_3$ 
       $\mid \phi_3 [+ \mid - \mid * \mid / \mid \%] \phi_4$ 
       $\mid \backslash result \mid \backslash old(t)$ 
       $\mid \backslash caller \mid \backslash amt \mid \backslash systime \mid \backslash blocknum$ 
       $\mid \backslash send(\phi_3, \phi_4, \phi_5)$ 
       $\mid \backslash hash(t)$ 
       $\mid true \mid false$ 
       $\mid \phi_1 [== \mid !=] \phi_2$ 
       $\mid \phi_1 [=== \mid !==] \phi_2$ 
       $\mid !\phi \mid \phi_1 [\&\& \mid \mid \mid ==>] \phi_2$ 
       $\mid \phi_1 [< \mid <= \mid >= \mid >] \phi_2$ 
       $\mid [\forall \mid \exists] (qVar) : \phi$ 
       $\mid \backslash sum(\phi_6)$ 
       $\mid StringLiteral$ 
       $\mid funName(args)$ 
       $\mid \backslash values(\phi) \mid \backslash keys(\phi) \mid \backslash size(\phi)$ 
       $\mid \backslash in$ 
       $\mid \backslash creates type:id(params)$ 

```

■ **Figure 7** The grammar of SCARML. ϕ_1 and ϕ_2 are of type bool, ϕ_3 and ϕ_4 are number-typed, ϕ_5 is of type account, and ϕ_6 is a set of number-typed values.

$\llbracket \text{true} \rrbracket$	$:= \top$	$\llbracket !\phi \rrbracket$	$:= \neg \llbracket \phi \rrbracket$
$\llbracket \text{false} \rrbracket$	$:= \perp$	$\llbracket \phi_1 \ \&\& \ \phi_2 \rrbracket$	$:= \llbracket \phi_1 \rrbracket \wedge \llbracket \phi_2 \rrbracket$
$\llbracket v \rrbracket$	$:= s_i(v)$	$\llbracket \phi_1 \ \ \phi_2 \rrbracket$	$:= \llbracket \phi_1 \rrbracket \vee \llbracket \phi_2 \rrbracket$
$\llbracket v[t] \rrbracket$	$:= s_i((v, \llbracket t \rrbracket))$	$\llbracket \phi_1 \Rightarrow \phi_2 \rrbracket$	$:= \llbracket \phi_1 \rrbracket \Rightarrow \llbracket \phi_2 \rrbracket$
$\llbracket \neg \phi \rrbracket$	$:= \neg \llbracket \phi \rrbracket$	$\llbracket \phi_1 < \phi_2 \rrbracket$	$:= \llbracket \phi_1 \rrbracket < \llbracket \phi_2 \rrbracket$
$\llbracket \phi_1 + \phi_2 \rrbracket$	$:= \llbracket \phi_1 \rrbracket + \llbracket \phi_2 \rrbracket$	$\llbracket \phi_1 \leq \phi_2 \rrbracket$	$:= \llbracket \phi_1 \rrbracket \leq \llbracket \phi_2 \rrbracket$
$\llbracket \phi_1 - \phi_2 \rrbracket$	$:= \llbracket \phi_1 \rrbracket - \llbracket \phi_2 \rrbracket$	$\llbracket \phi_1 \geq \phi_2 \rrbracket$	$:= \llbracket \phi_1 \rrbracket \geq \llbracket \phi_2 \rrbracket$
$\llbracket \phi_1 * \phi_2 \rrbracket$	$:= \llbracket \phi_1 \rrbracket * \llbracket \phi_2 \rrbracket$	$\llbracket \phi_1 > \phi_2 \rrbracket$	$:= \llbracket \phi_1 \rrbracket > \llbracket \phi_2 \rrbracket$
$\llbracket \backslash \text{result} \rrbracket$	$:= \text{return}_i$	$\llbracket \forall qVar : \phi \rrbracket$	$:=$
$\llbracket \backslash \text{old}(t) \rrbracket$	$:= \llbracket t \rrbracket_{s_{i-1}}$	$\forall x \in \text{RangeEval}(qVar) : \llbracket \{qVar/x\} \phi \rrbracket$	
$\llbracket \backslash \text{caller} \rrbracket$	$:= \text{caller}_i$	$\llbracket \exists qVar : \phi \rrbracket$	$:=$
$\llbracket \backslash \text{amt} \rrbracket$	$:= \text{amt}_i$	$\exists x \in \text{RangeEval}(qVar) : \llbracket \{qVar/x\} \phi \rrbracket$	
$\llbracket \backslash \text{systime} \rrbracket$	$:= \text{systime}_i$	$\llbracket \backslash \text{keys}(m) \rrbracket$	$:= \text{domain}(m)$
$\llbracket \backslash \text{blocknum} \rrbracket$	$:= \text{blocknum}_i$	$\llbracket \backslash \text{values}(m) \rrbracket$	$:= \text{codomain}(m)$
$\llbracket \backslash \text{send}(\phi_3, \phi_4, \phi_5) \rrbracket$	$:= \text{send}(\llbracket \phi_3 \rrbracket, \llbracket \phi_4 \rrbracket, \llbracket \phi_5 \rrbracket)$	$\llbracket \backslash \text{sum}(a) \rrbracket$	$:= \sum_{i \in \text{domain}(a)} \llbracket a \rrbracket(i)$
$\llbracket \backslash \text{hash}(t) \rrbracket$	$:= \text{hash}(\llbracket t \rrbracket)$	$\llbracket \backslash \text{size}(m) \rrbracket$	$:= \text{codomain}(m) $
$\llbracket \phi_1 == \phi_2 \rrbracket$	$:= \llbracket \phi_1 \rrbracket =_{\text{val}} \llbracket \phi_2 \rrbracket$	$\llbracket v(t) \rrbracket$	$:= s_i((v, \llbracket t \rrbracket))$ if $\llbracket t \rrbracket \in \text{domain}(v)$
$\llbracket \phi_1 != \phi_2 \rrbracket$	$:= \llbracket \phi_1 \rrbracket \neq_{\text{val}} \llbracket \phi_2 \rrbracket$	$\llbracket \phi_1 / \phi_2 \rrbracket$	$:= \llbracket \phi_1 \rrbracket / \llbracket \phi_2 \rrbracket$ if $\llbracket \phi_2 \rrbracket \neq 0$
$\llbracket \phi_1 === \phi_2 \rrbracket$	$:= \llbracket \phi_1 \rrbracket =_{\text{ref}} \llbracket \phi_2 \rrbracket$	$\llbracket \phi_1 \% \phi_2 \rrbracket$	$:= \llbracket \phi_1 \rrbracket \bmod \llbracket \phi_2 \rrbracket$ if $\llbracket \phi_2 \rrbracket \neq 0$
$\llbracket \phi_1 !== \phi_2 \rrbracket$	$:= \llbracket \phi_1 \rrbracket \neq_{\text{ref}} \llbracket \phi_2 \rrbracket$	$a \bmod n$	$:= a - n * (a/n)$

■ **Figure 8** The SCARML evaluation function $\llbracket \cdot \rrbracket$. The function is parameterized with the step τ_i , which contains the environment values caller_i , amt_i , systime_i , blocknum_i , and the prestate s_{i-1} and poststate s_i .

<i>app</i>	$:=$	application : <i>userType</i> * <i>appInit</i> <i>contract</i> +
<i>userType</i>	$:=$	<i>structDef</i> <i>enumDef</i>
<i>appInit</i>	$:=$	appInit : <i>id</i> : <i>contractType</i> (<i>params</i>) [, <i>id</i> : <i>contractType</i> (<i>params</i>)]*
<i>contract</i>	$:=$	<i>init?</i> <i>id</i> <i>state</i> <i>functions</i>
<i>init</i>	$:=$	<i>initParams?</i> <i>initCond</i> +
<i>initParams</i>	$:=$	initParams : <i>id</i> : <i>typeExpr</i> [, <i>id</i> : <i>typeExpr</i>]*
<i>initCond</i>	$:=$	init : <i>specExpr</i>
<i>state</i>	$:=$	state : <i>stateVar</i> +
<i>stateVar</i>	$:=$	var <i>id</i> : <i>typeExpr</i>
<i>functions</i>	$:=$	functions : <i>function</i> +
<i>function</i>	$:=$	fun <i>id</i> : <i>params?</i> <i>ret?</i> <i>pre</i> * <i>post</i> * <i>frame</i> *
<i>params</i>	$:=$	params : <i>id</i> : <i>typeExpr</i> [, <i>id</i> : <i>typeExpr</i>]*
<i>ret</i>	$:=$	returns : <i>typeExpr</i>
<i>pre</i>	$:=$	pre : <i>specExpr</i>
<i>post</i>	$:=$	post : <i>specExpr</i>
<i>typeExpr</i>	$:=$	<i>primitiveType</i> <i>arrType</i> <i>mapType</i> <i>id</i>
<i>primitiveType</i>	$:=$	bool int uint string
<i>arrType</i>	$:=$	<i>typeExpr</i> []
<i>mapType</i>	$:=$	mapping (<i>typeExpr</i> \Rightarrow <i>typeExpr</i>)
<i>frame</i>	$:=$	modifies : <i>frameExpr</i>

■ **Figure 9** A grammar for the basic metamodel. The *specExpr* symbol is defined in the functional specification language grammar (see Figure 7). The *frameExpr* symbol is defined in Figure 3.

14:12 Scar: Verification-Based Development of Smart Contracts

```
application: bank

contract bank:

  invariant: total >= 0
  invariant: total = \sum(\values(m))

  state:
    var total: uint
    var balances: mapping(account=>uint)

  init: total == 0
  init: \size(balances) == 0

  functions:
    fun deposit:
      post: balances[\caller]
           == \old(balances[\caller]) + \amt
      post: \send(\caller, \this, \amt)
      modifies: balances[\caller]
    fun withdraw:
      params: uint amount
      pre: amount <= balances[\caller]
      post: balances[\caller]
           == \old(balances[\caller]) - \amt
      post: \send(\this, \caller, amount)
      modifies: balances[\caller]
```

■ **Figure 10** A simple SCAR application model.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity >= 0.7;

library UTIL {
  enum Role {ANY}
  function hasRole(address a, Role r) internal pure returns (bool) { }
}

contract bank {
  mapping(address=>uint) balances;
  int total;
  constructor () { }

  /// @notice precondition balances[msg.sender] >= amount
  /// @notice postcondition balances[msg.sender] == __verifier_uint_old(
  ///   balances[msg.sender]) - amount
  /// @notice postcondition address(this).balance == __verifier_uint_old(
  ///   address(this).balance) - amount
  /// @notice postcondition address(msg.sender).balance ==
  ///   __verifier_uint_old(address(msg.sender).balance) + amount
  function withdraw(int amount) public { }

  /// @notice postcondition balances[msg.sender] == __verifier_uint_old(
  ///   balances[msg.sender]) + msg.value
  /// @notice postcondition address(this).balance == __verifier_uint_old(
  ///   address(this).balance) + msg.value
  /// @notice postcondition address(msg.sender).balance ==
  ///   __verifier_uint_old(address(msg.sender).balance) - msg.value
  function deposit() public { }
}
```

■ **Figure 11** The generated Solidity code with SOLC-VERIFY annotations.

Example	LoC Solidity	LoC SCAR
Bank	11	20
Escrow	24	18
Auction	38	34
Casino	58	53
Palinodia	299	174

■ **Figure 12** Lines of code comparison.

