5th International Workshop on Formal Methods for Blockchains (Pre-Proceedings)

FMBC 2024, April 7, 2024, Luxembourg City, Luxembourg

Edited by

Bruno Bernardo
Diego Marmsoler

Editors

Bruno Bernardo
Nomadic Labs, Paris, France
bruno@nomadic-labs.com

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

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 978-3-95977-317-1

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/978-3-95977-317-1.

Publication date
Publication date will be entered by OASIcs office.

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/OASIcs.FMBC.2024.0

OASIcs – OpenAccess Series in Informatics

OASIcs is a series of high-quality conference proceedings across all fields in informatics. OASIcs 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

Preliminary ToC
Entries will be updated during final typesetting by DagPub.

Preface
Bruno Bernardo and Diego Marmsoler 0:vii

Invited Talk

Deductive verification of smart contracts
Franck Cassez 1:1–1:1

Consensus

Formal specification of the Cardano blockchain ledger, mechanized in Agda
Andre Knispel, Orestis Melkonian, James Chapman, Alasdair Hill, Joosep Jääger, William DeMeo, and Ulf Norell 2:1–2:18

Formally Verifying the Safety of Pipelined Moonshot Consensus Protocol
M. Praveen, Raghavendra Ramesh, and Isaac Doidge 3:1–3:16

Towards Mechanised Consensus in Isabelle
Elliot Jones and Diego Marmsoler 4:1–4:22

Smart Contracts

Formalizing Automated Market Makers in the Lean 4 Theorem Prover
Daniele Pusceddu and Massimo Bartoletti 5:1–5:13

Towards benchmarking of Solidity verification tools
Massimo Bartoletti, Fabio Fioravanti, Giulia Matricardi, Roberto Pettinau, and Franco Sainas 6:1–6:15

Towards Formally Specifying and Verifying Smart Contract Upgrades in Coq
Derek Sorensen 7:1–7:14

A Practical Notion of Liveness in Smart Contract Applications
Jonas Schiffl and Bernhard Beckert 8:1–8:13

Securing Aptos Framework with Formal Verification
Junkil Park, Teng Zhang, Wolfgang Grieskamp, Meng Xu, Gerardo Di Giacomo, Kundu Chen, Yi Lu, and Robert Chen 9:1–9:16

Structured Contracts in the EUTxO Ledger Model
Polina Vinogradova, Orestis Melkonian, Philip Wadler, Manuel Chakravarty, Jacco Krijnen, Michael Peyton Jones, James Chapman, and Tudor Ferariu 10:1–10:19

5th International Workshop on Formal Methods for Blockchains (FMBC 2024).
Editors: Bruno Bernardo and Diego Marmsoler
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

The 5th International Workshop on Formal Methods for Blockchains (FMBC) took place on April 7, 2024, as part of the European Joint Conferences on Theory and Practice of Software (ETAPS 2024). FMBC's purpose is to be a forum to identify theoretical and practical approaches that apply formal methods to blockchain technology.

This fifth edition of FMBC attracted 17 submissions: 13 full papers, 1 short paper, and 3 extended abstracts. Each of these papers was reviewed by at least three program committee members or appointed external reviewers. This led to a selection of 9 (full) papers that were presented at the workshop as regular talks, as well as 2 extended abstracts that were presented as lightning talks. Additionally, we were very pleased to have an invited keynote by Franck Cassez (Head of Research at Mantle).

We thank all the authors that submitted a paper, as well as the program committee members and external reviewers for their immense work. We are grateful to Maxime Cordy and Renzo Gaston Degiovanni, Workshop Chairs of ETAPS 2024, for their help and guidance. FMBC 2024 was financially supported by the Ethereum Foundation's Ecosystem Support Program and Mantle.

April 2024

Bruno Bernardo
Diego Marmsoler

5th International Workshop on Formal Methods for Blockchains (FMBC 2024).
Editors: Bruno Bernardo and Diego Marmsoler
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Program Committee

Massimo Bartoletti
University of Cagliari, Italy

Bernhard Beckert
Karlsruhe Institute of Technology, Germany

Bruno Bernardo
Nomadic Labs, France

Martin Ceresa
IMDEA Software Institute, Spain

Manuel Chakravarty
Tweag, France

Sylvain Conchon
Paris-Saclay University, France

Denisa Diaconescu
University of Bucharest, Romania

Fritz Henglein
University of Copenhagen, Denmark

Maurice Herlihy
Brown University, US

Florian Kammueller
Middlesex University London, UK

Diego Marmsoler
University of Exeter, UK

Baoluo Meng
GE Research, US

Ron Van Der Meyden
University of New South Wales, Australia

Burcu Kulahcioglu Ozkan
Delft University of Technology, Netherlands

Gordon J. Pace
University of Malta, Malta

Maria Potop-Butucaru
Sorbonne University, France

Vincent Rahli
University of Birmingham, UK

Sophie Rain
Vienna University of Technology, Austria

Albert Rubio
Complutense University of Madrid, Spain

Bas Spitters
Aarhus University, Denmark

Meng Sun
Peking University, China

5th International Workshop on Formal Methods for Blockchains (FMBC 2024).
Editors: Bruno Bernardo and Diego Marmsoler
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

5th International Workshop on Formal Methods for Blockchains (FMBC 2024).
Editors: Bruno Bernardo and Diego Marmsoler
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Supporting Reviewers

Pablo Gordillo

Alejandro Hernández-Cerezo

Xiangyu Li

Xiaokun Luan

Saswata Paul

Sarat Chandra Varanasi

Part I.

Invited Talk

Deductive Verification of Smart Contracts

Franck Cassez
Mantle R&D

Abstract

At the core of the Ethereum network is the Ethereum Virtual Machine (EVM) which can execute programs written in EVM bytecode. This remarkable feature empowers users to define complex business logic that can be executed programmatically by programs called smart contracts. Smart contracts are programs and may contain bugs. There are several examples of smart contract vulnerabilities that have been exploited in the past: in 2016, a re-entrance vulnerability in the Decentralised Autonomous Organisation (DAO) smart contract was exploited to steal more than USD50 Million. The total value netted from DeFi hacks in 2023 is estimated to be more than $1.5 billion. In this talk I will discuss formal verification of smart contracts. The main technique is deductive verification supported by the verification-friendly language Dafny. I will show how we can use deductive verification to reason about smart contracts, from high-level specifications (Dafny), to intermediate representation (Yul) and finally low-level EVM bytecode.

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

Keywords and phrases Smart Contracts, Deductive Verification

Digital Object Identifier 10.4230/OASIcs.FMBC.2024.1

Category Invited Talk

Bio

Dr. Franck Cassez is currently Head of Research at Mantle. From 2019 to April 2023, he was Lead Researcher at Consensys in the R&D department. Before joining ConsenSys, he worked as a research scientist/academic for 25 years, at the French National Centre for Scientific Research (CNRS, France), National ICT Australia (NICTA now DATA61, Sydney AU) and Macquarie University (Sydney AU). He received his dual Engineering Degree in Computer Science/M.S. (1990) and a Ph.D. in Computer Science from from Ecole Centrale, Nantes, France in 1993.

Franck has published papers at major conferences including CONCUR, CAV, TACAS, ATVA, LPAR and several other venues. He was the recipient of several best paper awards including LPAR 2015, FMICS 2022, a Test-of-Time Award at CONCUR'22, and a Marie Curie Fellowship (2008-2011), an individual EU research excellence competitive grant. Franck has collaborated with many research groups in Europe and Australia and is known for some important results on timed automata (e.g. Test-of-Time Award at CONCUR'22 for the 2002 CONCUR paper The Impressive Power Of Stopwatches with Kim G. Larsen) and some efficient algorithms for timed games and time Petri nets.

Franck's contributions also include bringing research in practice, including static analysis tools (Goanna at NICTA, patent on Analysis of Program code, Skink at the International Software Verification Competition) or general purpose software packages (e.g. ScalaSMT a Scala interface combining state-of-the-art SMT-solvers). Franck's interest for blockchain technology started in 2019 when he joined ConsenSys and worked on several Ethereum research projects including the verification of the Deposit Contract, the formal verification of Smart Contracts in Dafny(best paper award at FMICS'22), a semantics of the EVM in Dafny, a semantics of Yul in Dafny. He has co-authored over 90 academic papers.

Franck's research interests include smart contracts security and analysis, formal verification techniques, programming languages, concurrent systems, zk-proof technology and rollups.

© Franck Cassez;
licensed under Creative Commons License CC-BY 4.0
5th International Workshop on Formal Methods for Blockchains (FMBC 2024).
Editors: Bruno Bernardo and Diego Marmsoler; Article No. 1; pp. 1:1–1:1
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Part II.

Consensus

Formal Specification of the Cardano Blockchain Ledger, Mechanized in Agda

Andre Knispel
Input Output, Global

Orestis Melkonian
Input Output, Global

James Chapman
Input Output, Global

Alasdair Hill
Input Output, Global

Joosep Jääger
Input Output, Global

William DeMeo
Input Output, Global

Ulf Norell
QuviQ, Göteborg, Sweden

Abstract

Blockchain systems comprise critical software that handle substantial monetary funds, rendering them excellent candidates for *formal verification*. One of their core components is the underlying ledger that does all the accounting: keeping track of transactions and their validity, etc.

Unfortunately, previous theoretical studies are typically confined to an idealized setting, while specifications for real implementations are scarce; either the functionality is directly implemented without a proper specification, or at best an informal specification is written on paper.

The present work expands beyond prior meta-theoretical investigations of the EUTxO model to encompass the full scale of the Cardano blockchain: our formal specification describes a hierarchy of modular transitions that covers all the intricacies of a realistic blockchain, such as fully expressive smart contracts and decentralized governance.

It is mechanized in a proof assistant, thus enjoys a higher standard of rigor: type-checking prevents minor oversights that were frequent in previous informal approaches; key meta-theoretical properties can now be formally proven; it is an *executable* specification against which the implementation in production is being tested for conformance; and it provides firm foundations for smart contract verification.

Apart from a safety net to keep us in check, the formalization also provides a guideline for the ledger design: one informs the other in a symbiotic way, especially in the case of state-of-the-art features like decentralized governance, which is an emerging sub-field of blockchain research that however mandates a more exploratory approach.

All the results presented in this paper have been mechanized in the Agda proof assistant and are publicly available. In fact, this document is itself a literate Agda script and all rendered code has been successfully type-checked.

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

Keywords and phrases blockchain, distributed ledgers, UTxO, Cardano, formal verification, Agda

Digital Object Identifier 10.4230/OASIcs.FMBC.2024.2

1 Introduction

This paper gives a high-level overview of the Cardano ledger specification in the Agda proof assistant, which is one of three core pieces of the Cardano blockchain:

- **Networking**: deals with sending messages across the internet.
- **Consensus**: establishes a common order of valid blocks.
- **Ledger**: decides whether a sequence of blocks is valid.

© Andre Knispel, Orestis Melkonian, James Chapman, Alasdair Hill, Joosep Jääger, William DeMeo, and Ulf Norell;
licensed under Creative Commons License CC-BY 4.0
5th International Workshop on Formal Methods for Blockchains (FMBC 2024).
Editors: Bruno Bernardo and Diego Marmsoler; Article No. 2; pp. 2:1–2:18
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Such *separation of concerns* is crucial to enable a rigidly formal study of each individual component; the ledger is based on the *Extended UTxO* model (EUTxO), an extension of Bitcoin's model of unspent transaction outputs [19] – in contrast to Ethereum's account-based model [8] – to accommodate fully expressive *smart contracts* that run on the blockchain. Luckily for us, EUTxO enjoys a well-studied meta-theory [9, 10] that is also mechanized in Agda, albeit in a much simpler setting where a single ledger feature is considered at a time, but not how multiple concurrent features interact. We take this to the next level by scaling up these prior theoretical results to match the complexity of the real world: the Cardano blockchain being one of the top ten cryptocurrencies today by market capitalization, it handles gigabytes of transactions that transfer hundred of millions US dollars, while simultaneously supporting all these features plus many more that have not been formally studied before.

We are happy to report that the formalization overhead has proven minuscule compared to the development effort of the actual implementation, measured either by lines of code (˜10 thousand lines of Agda formalization *versus* ˜200 thousand of Haskell implementation) or by number of man hours put in so far (only a couple of full-time formal methods engineers *versus* tens of production developers). The result is a *mechanized* document that leaves little room for error, additionally proves crucial invariants of the overall system ,e.g., that the global value carried by the system stays constant, formally stated in Section 4. It doubles as an executable reference implementation that we can utilize in production for conformance testing. All of our work, much like this paper, is mechanized in Agda and is publicly available:

`https://github.com/IntersectMBO/formal-ledger-specifications`

Scope. Cardano's evolution proceeds in *eras*, each introducing a new vital feature to the previous ones. While we would ideally want to provide a multitude of formal artifacts, each describing a single era in full detail, the specification formalized here is that of the **Voltaire** era that introduces *decentralized governance* as described in the Cardano Improvement Proposal (CIP) 1694.[1] This stems from the fact that the design of the blockchain happens in tandem with the formal specification; one informs the other in an intricate, non-linear fashion. Thus arises a pragmatic need to think of the process as an act of balance between keeping up with the *past*, i.e., going back to previous eras and incrementally incorporating their features, and co-evolving with the current design of the *future* ledger capabilities. Therefore, we set aside details of the previous **Byron**, **Shelley**, and **Alonzo** eras while at the same time missing orthogonal features related to smart contracts brought in the **Babbage** era.

Transitions as relations. The ledger can itself be conceptually divided into multiple sub-components, each described by a transition between states that only contains the relevant parts of the overarching ledger state and possibly some internal auxiliary information that is discarded at the outer layer. These transitions are not independent, but form a hierarchy of "state machines" where some higher-level transition might demand successful transition of a sub-component down the dependency graph as one of its premises. Eventually, these cascading transitions all get combined to dictate the top-level transition that handles an individual block of transactions submitted to the blockchain.

Formally, we formulate such (labeled) transitions as relations X between the environment Γ inherited from a higher layer, an initial state s, a signal b that acts as user input, and a final state s':

[1] `https://github.com/cardano-foundation/CIPs/blob/17771640/CIP-1694/README.md`

$$\Gamma \vdash s \xrightarrow[X]{b} s'$$

Environments (Signals)	*States*
Possible transitions	

We will henceforth present such transitions as shown on the right; a *triptych* defining environments and possibly signals (top left), states (top right), and the rules that *inductively* define the transition (bottom).

1.1 Agda preliminaries

In Agda, the aforementioned ledger transitions are modeled as *inductive families* of type:

```
_⊢_⇀⦇_,_⦈_ : Env → State → Signal → State → Type
```

Reflexive transitive closure. We will often need to apply a transition repeatedly until we arrive at a final state, which corresponds to the standard mathematical construction of taking the relation's *reflexive transitive closure*:

```
data _⊢_⇀⦇_,_⦈*_ : Env → State → List Signal → State → Type where

  base :
    ─────────────────────
    Γ ⊢ s ⇀⦇ [] ,_⦈* s

  step :
    • Γ ⊢ s  ⇀⦇ b      ,_⦈  s'
    • Γ ⊢ s' ⇀⦇ bs     ,_⦈* s''
    ─────────────────────
    Γ ⊢ s  ⇀⦇ b ∷ bs ,_⦈* s''
```

Finite sets & maps. One particular trait we inherited from previous pen-and-paper iterations of the ledger specification is a heavy use of set theory, which goes against Agda's foundations in Type Theory, both technically and in a philosophical sense. To remedy this, we have developed an in-house library for conducting *Axiomatic Set Theory* within the type-theoretic setting of Agda [18]; we stay in its finite fragment for this application. Crucially, the type of sets is entirely *abstract*: there is no way to utilize proof-by-computation (e.g., as one would do when modeling sets as lists of distinct elements), so that all proofs eventually resort to the axioms and the library's implementation details stay irrelevant. At the same time, when extracting executable code the library provides a properly executable implementation – the abstraction layer only exists at compile-time. Implementing this abstraction layer helped us greatly reduce code complexity and size over a previous list-based approach. In fact, it is highly encouraged to provide *multiple* implementations without affecting the formalization and the validity of the established proofs therein.

Equipped with the axioms provided by the library, e.g., the ability to construct power sets ℙ, it is remarkably easy to define common set-theoretic concepts like set inclusion and extensional equality of sets (left), as well as re-purpose sets of key-value pairs to model *finite maps*[2] by imposing uniqueness of keys (right):

```
_⊆_ : {A : Type} → ℙ A → ℙ A → Type
X ⊆ Y = ∀ {x} → x ∈ X → x ∈ Y

_≈_ : {A : Type} → ℙ A → ℙ A → Type
X ≈ Y = X ⊆ Y × Y ⊆ X
```

```
_⇀_ : Type → Type → Type
A ⇀ B = ∃ λ (ℜ : ℙ (A × B)) →
  ∀ {a b b'} → (a , b) ∈ ℜ → (a , b') ∈ ℜ → b ≡ b'
```

[2] It is natural to think of maps as partial functions, but unrestricted Agda functions would not do here.

2 Fundamental entities

2.1 Cryptographic primitives

There are two types of credentials that can be used on Cardano: VKey and script credentials. VKey credentials use a public key signing scheme (Ed25519) for verification. Some serialized (Ser) data can be signed, and isSigned is the property that a public VKey signed some data with a given signature (Sig). There are also other cryptographic primitives in the Cardano ledger, for example KES and VRF used in the consensus layer, but we omit those here.

Script credentials correspond to a hash of a script that has to be executed by the ledger as part of transaction validation. There are two different types of scripts, native and Plutus, but the details of this are not relevant for the rest of this paper.

```
VKey Sig Ser : Type                    isSigned : VKey → Ser → Sig → Type
```

In the specification, all definitions that require these primitives must accept these as additional arguments. To streamline this process, these definitions are bundled into a record and, using Agda's module system, are quantified only once per file. We are using this pattern many times, either to introduce additional abstraction barriers or to effectively provide foreign functions within a safe environment. Additionally, particularly fundamental interfaces like the one presented above are sometimes re-bundled transitively into larger records, which further streamlines the interface. This is in stark contrast to the Haskell implementation, which often needs to repeat tens of type class constraints on many functions in a module.

2.2 Addresses

There are various types of addresses used for storing funds in the UTxO set, which all contain a payment Credential and optionally a staking Credential. Addr is the union of all of those types. A Credential is a hash of a public key or script, types for which are kept abstract. The most common type of address is a BaseAddr which must include a staking Credential.

There is also a special type of address (not included in Addr) without a payment credential, called a reward address. It is not used for interacting with the UTxO set, but instead used to refer to reward accounts [31].

```
Credential = KeyHash ⊎ ScriptHash
```

```
record BaseAddr : Type where          record RwdAddr : Type where
  pay   : Credential                    stake : Credential
  stake : Credential
```

```
Addr = BaseAddr ⊎ ...
```

2.3 Base types

The basic units of currency and time are Coin, Slot and Epoch, which we treat as natural numbers, while an implementation might use isomorphic but more complicated types (for example to represent the beginning of time in a special way).

```
Coin = Slot = Epoch = ℕ
```

A Coin is the smallest unit of currency, a Slot is the smallest unit of time (corresponding to 1 second in the main chain), and an Epoch is a fixed number of slots (corresponding to 5 days in the main chain). Every slot, a stake pool has a random chance to be able to mint a block, and one block every five slots is expected [13].

3 Advancing the blockchain

3.1 Protocol parameters

We start with adjustable protocol parameters. In contrast to constants such as the length of an Epoch, these parameters can be changed while the system is running via the governance mechanism. They can affect various features of the system, such as minimum fees, maximum and minimum sizes of certain components, and more.

The full specification contains well over 20 parameters, while we only list a few. The maximum sizes should be self-explanatory, while a and b are the coefficients of a polynomial used in the calculation of the minimum fee for transactions (c.f., function minfee in Appendix B).

```
record PParams : Type where
  maxBlockSize maxTxSize a b : ℕ
```

3.2 Extending the blockchain block-by-block

CHAIN is the main state machine describing the ledger. Since it is not invoked from any other state machine, it does not have an environment. It invokes two other state machines, NEWEPOCH and LEDGER*, where the former detects if the new block *b* is in a new epoch. In that case, NEWEPOCH takes care of various bookkeeping tasks, such as counting votes for the governance system and updating stake distributions for consensus. For a basic version that detects whether a new epoch has been entered, see Appendix C. The potentially updated state is then given to LEDGER*, which is the reflexive-transitive closure of LEDGER and applies all the transactions in the block in sequence. Finally, CHAIN updates ChainState with the resulting states.

There is a key property about NEWEPOCH, which is that it never gets stuck, i.e. that for all states, environments and signals it always transitions to a new state. This property is proven in our development.

```
record Block : Type where
  ts   : List Tx
  slot : Slot
```

```
record NewEpochState : Type where
  lastEpoch : Epoch
  acnt      : Acnt
  ls        : LState
  es        : EnactState
  fut       : RatifyState

record ChainState : Type where
  newEpochState : NewEpochState
```

```
CHAIN :
  • mkNewEpochEnv s ⊢ s .newEpochState ⇀⦇ epoch slot ,NEWEPOCH ⦈ nes
  • ⟦ slot ⊗ constitution .proj₁ .proj₂ ⊗ pparams .proj₁ ⊗ es ⟧ ⊢ nes .ls ⇀⦇ ts ,LEDGER∗ ⦈ ls'
  ────────────────────────────────────────────────────────────────
    _ ⊢ s ⇀⦇ b ,CHAIN ⦈ updateChainState s nes
```

3.3 Extending the ledger transaction-by-transaction

Transaction processing is broken down into three separate parts: accounting & witnessing (UTXOW), application of certificates (CERT) and processing of governance votes & proposals (GOV).

```
record LEnv : Type where
  slot        : Slot
  ppolicy     : Maybe ScriptHash
  pparams     : PParams
  enactState  : EnactState
```

```
record LState : Type where
  utxoSt    : UTxOState
  govSt     : GovState
  certState : CertState
```

```
LEDGER :
  • mkUTxOEnv Γ ⊢ utxoSt ⇀⦇ tx ,UTXOW ⦈ utxoSt'
  • ⟦ epoch slot ⊗ pparams ⊗ txvote ⊗ txwdrls ⟧ ⊢ certState ⇀⦇ txcerts ,CERT* ⦈ certState'
  • ⟦ txid ⊗ epoch slot ⊗ pparams ⊗ enactState ⟧ ⊢ govSt ⇀⦇ txgov txb ,GOV* ⦈ govSt'
  ────────────────────────────────────────────────────────────────
  Γ ⊢ s ⇀⦇ tx ,LEDGER ⦈ ⟦ utxoSt' ⊗ govSt' ⊗ certState' ⟧
```

(The notation ⟦ . . . ⊗ . . . ⟧ constructs records of any type by giving their fields in order.)

4 UTxO

4.1 Witnessing

Transaction witnessing checks that all required signatures are present and all required scripts accept the validity of the given transaction. *witsKeyHashes* and *witsScriptHashes* is the set of hashes of keys/scripts included in the transaction.

```
UTXOW-inductive :
  • witsVKeyNeeded ppolicy utxo txb ⊆ witsKeyHashes
  • scriptsNeeded ppolicy utxo txb ≡ witsScriptHashes
  • ∀[ (vk , σ) ∈ vkSigs ] isSigned vk (txidBytes txid) σ
  • ∀[ s ∈ scriptsP1 ] validP1Script witsKeyHashes txvldt s
  • Γ ⊢ s ⇀⦇ tx ,UTXO ⦈ s'
  ────────────────────────────────────────────
  Γ ⊢ s ⇀⦇ tx ,UTXOW ⦈ s'
```

4.2 Accounting

Accounting is handled by the UTXO state machine. The preconditions for UTXO-inductive ensure various properties or prevent attacks. For example, if txins was allowed to be empty, one could make a transaction that only spends from reward accounts. This does not require a specific hash to be present in the transaction body, so such a transaction could be repeatable in certain scenarios. The equation between produced and consumed ensures that the transaction is properly balanced. For details on some of these functions, see Appendix B.

```
record UTxOEnv : Type where
  slot    : Slot
  pparams : PParams
Deposits = DepositPurpose ⇀ Coin
```

```
record UTxOState : Type where
  utxo           : UTxO
  deposits       : Deposits
  fees donations : Coin
```

```
UTXO-inductive :
• txins ≢ ∅
• txins ⊆ dom utxo
• minfee pp tx ≤ txfee
• txsize ≤ maxTxSize pp
• consumed pp s txb ≡ produced pp s txb
• coin mint ≡ 0
────────────────────────────────────────
                      ⟦ (utxo ∣ txins ᶜ) ∪ outs txb
                      ⊗ updateDeposits pp txb deposits
Γ ⊢ s ⇀⦇ tx ,UTXO ⦈  ⊗ fees + txfee
                      ⊗ donations + txdonation ⟧
```

▶ **Property 4.1** (Value preservation). *Let getCoin be the sum of all coins contained within a UTxOState. Then, for all* $\Gamma \in$ *UTxOEnv,* $s, s' \in$ *UTxOState and* $tx \in$ *Tx, if* tx *.body .txid* $\notin$ *map* $proj_1$ *(dom (s .UTxOState.utxo))and* $\Gamma \vdash s \rightharpoonup⦇ tx$ *,UTXO ⦈* s'*then getCoin* $s \equiv$ *getCoin* s'*.*

Note that this is one of the most important properties of a UTxO-based ledger, as evidenced by its central place in EUTxO's meta-theory [9, 10].

5 Decentralized Governance

5.1 Entities and actions

The governance framework has three bodies of governance, the constitutional committee, delegated representatives and stake pool operators, corresponding to the roles CC, DRep and SPO. Proposals relevant to the governance system come in the form of Governance Actions. They are identified by their GovActionID, which consists of the TxId belonging to the transaction that proposed it and the index within that transaction (a transaction can propose multiple governance actions at once).

```
GovActionID = TxId × ℕ
data GovRole : Type where
  CC DRep SPO : GovRole
data GovAction : Type where
  NoConfidence    :                                          GovAction
  NewCommittee    : Credential ⇀ Epoch → ℙ Credential → ℚ → GovAction
  NewConstitution : DocHash → Maybe ScriptHash             → GovAction
  TriggerHF       : ProtVer                                → GovAction
  ChangePParams   : PParamsUpdate                          → GovAction
  TreasuryWdrl    : (RwdAddr ⇀ Coin)                       → GovAction
  Info            :                                          GovAction
```

For the meaning of these individual actions, see [12].

5.2 Votes and proposals

Before a Vote can be cast it must be packaged together with further information, such as who is voting and for which governance action. This information is combined in the GovVote record. To propose a governance action, a GovProposal needs to be submitted. Beside the proposed action, it requires a deposit, which will be returned to returnAddr.

```
data Vote : Type where
  yes no abstain : Vote
```

```
record GovVote : Type where
  gid        : GovActionID
  role       : GovRole
  credential : Credential
  vote       : Vote
```

```
record GovProposal : Type where
  action     : GovAction
  deposit    : Coin
  returnAddr : RwdAddr
```

5.3 Enactment

Enactment of a governance action is carried out via the ENACT state machine. We just show two example rules for this state machine – there is one corresponding to each constructor of GovAction. For an explanation of the hash protection scheme, see Appendix A.

```
record EnactEnv : Type where
  gid      : GovActionID
  treasury : Coin
  epoch    : Epoch
```

```
record EnactState : Type where
  cc           : HashProtected (Maybe ((Credential ⇀ Epoch) × ℚ))
  constitution : HashProtected (DocHash × Maybe ScriptHash)
  pv           : HashProtected ProtVer
  pparams      : HashProtected PParams
  withdrawals  : RwdAddr ⇀ Coin
```

Enact-NewConst :

$$\frac{}{⟦\ gid \otimes t \otimes e\ ⟧ \vdash s \rightharpoonup⦇\ \text{NewConstitution}\ dh\ sh\ ,\text{ENACT}\ ⦈\ \text{record}\ s\ \{\ \text{constitution} = (dh\ ,\ sh)\ ,\ gid\ \}}$$

Enact-Wdrl :

$$\frac{\text{let}\ newWdrls = s\ .\text{withdrawals} \cup wdrl\ \text{in}\ \sum[\ x \leftarrow newWdrls\ ]\ x \leq t}{⟦\ gid \otimes t \otimes e\ ⟧ \vdash s \rightharpoonup⦇\ \text{TreasuryWdrl}\ wdrl\ ,\text{ENACT}\ ⦈\ \text{record}\ s\ \{\ \text{withdrawals} = newWdrls\ \}}$$

(The record keyword indicates a record update, i.e. we take the existing EnactState and update one of its fields.)

5.4 Voting and Proposing

The order of proposals is maintained by keeping governance actions in a list – this acts as a tie breaker when multiple competing actions might be able to be ratified at the same time.

```
record GovActionState : Type where
  votes      : (GovRole × Credential) ⇀ Vote
  returnAddr : RwdAddr
  expiresIn  : Epoch
  action     : GovAction
  prevAction : NeedsHash action

GovState = List (GovActionID × GovActionState)
```

```
record GovEnv : Type where
  txid       : TxId
  epoch      : Epoch
  pparams    : PParams
  enactState : EnactState
```

GOV-Vote :

- $(aid\ ,\ ast)$ ∈ fromList s
- canVote pparams (action ast) $role$

$(\Gamma\ ,\ k) \vdash s$ ⇀⦇ sig ,GOV ⦈ addVote s aid $role$ $cred$ v

GOV-Propose :

- actionWellFormed a ≡ true
- d ≡ govActionDeposit

$(\Gamma\ ,\ k) \vdash s$ ⇀⦇ inj_2 $prop$,GOV ⦈ addAction s (govActionLifetime + epoch) (txid , k) $addr$ a $prev$

5.5 Ratification

Governance actions are *ratified* through on-chain voting actions. Different kinds of governance actions have different ratification requirements but always involve at least *two of the three* governance bodies. The voting power of the DRep and SPO roles is proportional to the stake delegated to them, while the constitutional committee has individually elected members where each member has the same voting power.

Some actions take priority over others and, when enacted, delay all further ratification to the next epoch boundary. This allows a changed government to reevaluate existing proposals.

```
record RatifyEnv : Type where
  stakeDistrs  : StakeDistrs
  currentEpoch : Epoch
  dreps        : Credential ⇀ Epoch
```

```
record RatifyState : Type where
  es      : EnactState
  removed : ℙ (GovActionID × GovActionState)
  delay   : Bool
```

RATIFY-Accept :

- accepted Γ es st
- ¬ delayed action prevAction es d
- ⟦ a .proj₁ ⊗ treasury ⊗ currentEpoch ⟧ ⊢ es ⇀⦇ action ,ENACT ⦈ es'

Γ ⊢ ⟦ es ⊗ $removed$ ⊗ d ⟧ ⇀⦇ a ,RATIFY ⦈
⟦ es' ⊗ { a } ∪ $removed$ ⊗ delayingAction action ⟧

RATIFY-Reject :

- ¬ accepted Γ es st
- expired currentEpoch st

```
    ──────────────────────────────────────────────────────────────
    Γ ⊢ ⟦ es ⊗ removed ⊗ d ⟧ ⇀⦇ a ,RATIFY ⦈ ⟦ es ⊗ ❴ a ❵ ∪ removed ⊗ d ⟧

RATIFY-Continue :
    ( • ¬ accepted Γ es st • ¬ expired currentEpoch st)
  ⊎ ( • accepted Γ es st
      • ( delayed action prevAction es d
        ⊎ (∀ es' → ¬ ⟦ a .proj₁ ⊗ treasury ⊗ currentEpoch ⟧ ⊢ es ⇀⦇ action ,ENACT ⦈ es')))
    ──────────────────────────────────────────────────────────────
    Γ ⊢ ⟦ es ⊗ removed ⊗ d ⟧ ⇀⦇ a ,RATIFY ⦈ ⟦ es ⊗ removed ⊗ d ⟧
```

The main new ingredients for the rules of the RATIFY state machine are:

- accepted, which is the property that there are sufficient votes from the required bodies to pass this action;
- delayed, which expresses whether an action is delayed;
- expired, which becomes true a certain number of epochs after the action has been proposed.

The three RATIFY rules correspond to the cases where an action can be ratified and enacted (in which case it is), or it is expired and can be removed, or, otherwise it will be kept around for the future. This means that all governance actions eventually either get accepted and enacted via RATIFY-Accept or rejected via RATIFY-Reject. It is not possible to remove actions by voting against them, one has to wait for the action to expire.

6 Transactions

A transaction is made up of a transaction body and a collection of witnesses.

```
Ix TxId : Type
TxIn    = TxId × Ix
TxOut = Addr × Value × Maybe DataHash
UTxO  = TxIn ⇀ TxOut
```

```
record TxBody : Type where
  txins   : ℙ TxIn
  txouts  : Ix ⇀ TxOut
  txfee   : Coin
  txvote  : List GovVote
  txprop  : List GovProposal
  txsize  : ℕ
  txid    : TxId

record TxWitnesses : Type where
  vkSigs  : VKey ⇀ Sig
  scripts : ℙ Script

record Tx : Type where
  body  : TxBody
  wits  : TxWitnesses
```

Some key ingredients in the transaction body are:

- A set of transaction inputs (txins), each of which identifies an output from a previous transaction. A transaction input (TxIn) consists of a transaction ID and an index to uniquely identify the output.
- An indexed collection of transaction outputs (txouts). A transaction output (TxOut) is an address paired with a multi-asset Value (see [10]).
- A transaction fee (txfee), whose value will be added to the fee pot.

- The size (txsize) and the hash (txid) of the serialized form of the transaction that was included in the block. Cardano's serialization is not canonical, so any information that is necessary but lost during deserialisation must be preserved by attaching it to the data like this.

7 Compiling to a Haskell implementation & Conformance testing

In order to deliver on our promise that the specification is also *executable*, there is still some work to be done given that all transitions have been formulated as relations.

This is precisely the reason we also manually prove that each and every transition of the previous sections is indeed *computational*:

```
record Computational (_⊢_⇀⦇_,X ⦈_ : C → S → Sig → S → Type) : Type where
  compute         : C → S → Sig → Maybe S
  compute-correct : compute Γ s b ≡ just s' ⇔ Γ ⊢ s ⇀⦇ b ,X ⦈ s'
```

The definition above captures what it means for a (small-step) relation to be accurately computed by a function compute, which given as input an environment, source state, and signal, outputs the resulting state or an error for invalid transitions. Most importantly, such a function must be *sound* and *complete*: it does not return output states that are not related, and, *vice versa*, all related states are successfully returned. An alternative interpretation is that this rules out *non-determinism* across all ledger transitions, i.e., there cannot be two distinct states arising from the same inputs.

There is one last obstacle that hinders execution: we have leveraged Agda's module system[3] to parameterize our specification over some abstract types and functions that we assume as given, e.g., the cryptographic primitives. As a final step, we instantiate these parameters with concrete definitions, either by manually providing them within Agda, or deferring to the Haskell *foreign function interface* to reuse existing Haskell ones that have no Agda counterpart.

Equipped with a fully concrete specification and the Computational proofs for each relation, it is finally possible to generate executable Haskell code using Agda's MAlonzo compilation backend.[4] The resulting Haskell library is then deployed as part of the automated testing setup for the Cardano ledger in production, so as to ensure the developers have faithfully implemented the specification. This is made possible by virtue of the implementation mirroring the specification's structure to define transitions, which one can then test by randomly generating environments/states/signals, and executing both state machines on these same random inputs to compare the final results for *conformance*.

One small caveat remains though: production code might use different data structures, mainly for reasons of *performance*, which are not isomorphic to those used in the specification and might require non-trivial translation functions and notions of equality to perform the aforementioned tests. In the future, we plan to also formalize these more efficient representations in Agda and prove that soundness is preserved regardless.

[3] https://agda.readthedocs.io/en/v2.6.4/language/module-system.html#parameterised-modules
[4] https://agda.readthedocs.io/en/v2.6.4/tools/compilers.html#ghc-backend

8 Related Work

EUTxO. The approach we followed is a natural evolution of prior meta-theoretical results on the EUTxO model [9, 10], but now employed at a much larger scale to cover all the features of a realistic ledger: epochs, protocol parameters, decentralized governance, etc.

All this complexity does not come for free though: one has to be economical about which properties to prove of the resulting system, and this might entail limiting oneself to mechanizing just the core properties, such as global value preservation as we saw with Property 4.1, otherwise the whole effort can quickly become practically infeasible to maintain from a software-engineering perspective.

Formal Methods, generally. The overarching methodology – formally specifying the system under design – is by no means particular to the blockchain space. A principal success story in the wider computing world nowadays is definitely the *WebAssembly* language, an alternative to Javascript to act as a compilation target for web applications with performance and security in mind [16], which was designed in tandem with a formalization of its semantics [29].

Apart from keeping programming language designers honest by making sure no edge cases are overlooked, it allows the language to evolve in a much more robust fashion: every future extension has to pass through a rigorous process which eventually involves extending the formalization itself.

While the WebAssembly line of work [29, 30] provided much inspiration for us, we believe our approach to be even more radical by mitigating the need for informal processes altogether: the formalization *is* the specification!

Formal Methods, specifically for blockchain. The work presented here fits well within Cardano's vision for *agile formal methods* [17], which strikes a good balance between a fully certified implementation (too much effort, too few resources) and an informal, under-specified product (quicker, easier, but far less trustworthy). Instead of demanding the impossible by extracting the actual production from the formalization itself, we find the sweet spot lies in the middle: extracting a *reference implementation* in Haskell and using *conformance testing* to ensure the system in production behaves as it should (c.f., Section 7).

Apart from our work, there are very few mechanized results on UTxO-based blockchains (modeled after Bitcoin [19]), and all of them invariably are formulated on a idealized setting [26, 1, 9, 10], abstracting away the complexity that ensues when multiple features interact. Thus, the mechanized specification presented here for the Cardano ledger is the first of its kind, and we hope this sets a higher standard for subsequent work and pushes forward a more formal agenda for blockchain research in the future.

Although not directly comparable to our use case, account-based blockchains (modeled after Ethereum [8]) fair better in this respect, with plenty of formal method tools available, ranging from model checking [15, 28] to full-blown formal verification [11, 7, 23]. Notable blockchains that spearhead progress in this direction include Tezos [5, 6, 14], Ziliiqa and its Scilla smart-contract language [25, 24], and Concordium [3, 21, 2, 27, 20]. The main difference with our work lies in *readability*, partly due to the choice of tool (Agda being notorious for its beautiful renderings but lack of proper support for practical "big" proofs that arise in large scale software verification projects, where tactic-based proof assistants like Coq [4] and Isabelle [22] are more common), and the point where mechanization is placed within the development pipeline: most aforementioned work builds upon informal pen-and-paper documents and some of its aspects are only mechanized *a-posteriori*. Having said that,

the fundamental split stems from a completely different *target audience*; our formalization is meant to be read by researchers, formal methods engineers, compiler engineers, and developers alike. In contrast, the majority of the aforementioned work is primarily targeted at a select team of experts which complement other (informal) documentation and software.

9 Conclusion

We have outlined the mechanized specification of the EUTxO-based ledger rules of the Cardano blockchain, by taking a *bird's-eye view* of the hierarchy of transitions handling different sub-components in a modular way.

Although space limitations preclude us from exhaustively fleshing out all the gory details of our formalization, we hope to have conveyed the general *design principles* that will be helpful to others when attempting to mechanize something of this kind and at this scale. In the little space we could afford for more thorough details, we made a conscious choice of putting emphasis on the most novel aspect of the current era of the Cardano blockchain: *decentralized governance*. There, the introduction of the notions of voting, ratification, and enactment complicate the ledger rules of previous eras – albeit in a fairly orthogonal way, which we found particularly satisfying.

A mechanized formal artifact of this kind is rigid enough to eliminate any ambiguity that would often arise in pen-and-paper specifications, all the while sustaining a readable document that is accessible to a wide audience and allows for varied uses.

By virtue of conducting our work within a proof assistant based on *constructive* logic, our result extends beyond a purely theoretical exercise to an *executable* resource that can be leveraged as a *reference implementation*, against which a system-in-production can be tested for conformance.

Last but not least, it is evident that developing a ledger on these foundations opens up a plethora of opportunities for further formalization work, e.g., instantiating the abstract notion of scripts with actual *Plutus* scripts brings us close to enabling practical smart contract verification where developers write their programs immediately in Agda, prove properties about their behavior, and then extract Plutus code they can deploy to the actual Cardano blockchain. All these point to bright prospects for formal methods in UTxO-based blockchains, which we are excited to explore in the future and hope that others do as well.

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 Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. Extracting smart contracts tested and verified in Coq. In Catalin Hritcu and Andrei Popescu, editors, *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, pages 105–121. ACM, 2021. `doi:10.1145/3437992.3439934`.

3 Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. Concert: a smart contract certification framework in Coq. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 215–228. ACM, 2020. `doi:10.1145/3372885.3373829`.

4. Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq proof assistant reference manual: Version 6.1.* PhD thesis, Inria, 1997.
5. Bruno Bernardo, Raphaël Cauderlier, Guillaume Claret, Arvid Jakobsson, Basile Pesin, and Julien Tesson. Making tezos smart contracts more reliable with Coq. 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 60–72. Springer, 2020. `doi:10.1007/978-3-030-61467-6_5`.
6. Bruno Bernardo, Raphaël Cauderlier, Zhenlei Hu, Basile Pesin, and Julien Tesson. Mi-cho-coq, a framework for certifying Tezos smart contracts. In Emil Sekerinski, Nelma Moreira, José N. Oliveira, Daniel Ratiu, Riccardo Guidotti, Marie Farrell, Matt Luckcuck, Diego Marmsoler, José Creissac Campos, Troy Astarte, Laure Gonnord, Antonio Cerone, Luis Couto, Brijesh Dongol, Martin Kutrib, Pedro Monteiro, and David Delmas, editors, *Formal Methods. FM 2019 International Workshops - Porto, Portugal, October 7-11, 2019, Revised Selected Papers, Part I*, volume 12232 of *Lecture Notes in Computer Science*, pages 368–379. Springer, 2019. `doi:10.1007/978-3-030-54994-7_28`.
7. Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, pages 91–96, 2016.
8. Vitalik Buterin. A next-generation smart contract and decentralized application platform (white paper). `https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf`, 2014.
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. Xiaohong Chen, Daejun Park, and Grigore Roşu. A language-independent approach to smart contract verification. In *International Symposium on Leveraging Applications of Formal Methods*, pages 405–413. Springer, 2018.
12. Jared Corduan, Matthias Benkort, Kevin Hammond, Charles Hoskinson, Andre Knispel, and Samuel Leathers. A first step towards on-chain decentralized governance. `https://cips.cardano.org/cip/CIP-1694`, 2023.
13. 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 .
14. 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.

15 Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in Ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–27, 2018.

16 Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. Bringing the web up to speed with WebAssembly. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 185–200. ACM, 2017. doi:10.1145/3062341.3062363.

17 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 - 21st International Symposium, TFP 2020, Krakow, Poland, February 13-14, 2020, Revised Selected Papers*, volume 12222 of *Lecture Notes in Computer Science*, pages 94–120. Springer, 2020. doi:10.1007/978-3-030-57761-2_5.

18 Andre Knispel. Constructive zf-style set theory in type theory. unpublished, 2023. URL: https://whatisrt.github.io/papers/ZF-style-set-theory-in-type-theory.pdf.

19 S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/en/bitcoin-paper, October 2008.

20 Eske Hoy Nielsen, Danil Annenkov, and Bas Spitters. Formalising decentralised exchanges in Coq. In Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve Zdancewic, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*, pages 290–302. ACM, 2023. doi:10.1145/3573105.3575685.

21 Jakob Botsch Nielsen and Bas Spitters. Smart contract interactions in Coq. In Emil Sekerinski, Nelma Moreira, José N. Oliveira, Daniel Ratiu, Riccardo Guidotti, Marie Farrell, Matt Luckcuck, Diego Marmsoler, José Creissac Campos, Troy Astarte, Laure Gonnord, Antonio Cerone, Luis Couto, Brijesh Dongol, Martin Kutrib, Pedro Monteiro, and David Delmas, editors, *Formal Methods. FM 2019 International Workshops - Porto, Portugal, October 7-11, 2019, Revised Selected Papers, Part I*, volume 12232 of *Lecture Notes in Computer Science*, pages 380–391. Springer, 2019. doi:10.1007/978-3-030-54994-7_29.

22 Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

23 George Pîrlea 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.

24 Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Temporal properties of smart contracts. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*, volume 11247 of *Lecture Notes in Computer Science*, pages 323–338. Springer, 2018. doi:10.1007/978-3-030-03427-6_25.

25 Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. Safer smart contract programming with Scilla. *Proc. ACM Program. Lang.*, 3(OOPSLA):185:1–185:30, 2019. doi:10.1145/3360611.

26 Anton Setzer. Modelling Bitcoin in Agda. *CoRR*, abs/1804.06398, 2018. arXiv:1804.06398.

27 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.

28 Petar Tsankov. Security analysis of smart contracts in Datalog. In *International Symposium on Leveraging Applications of Formal Methods*, pages 316–322. Springer, 2018.

29 Conrad Watt. Mechanising and verifying the WebAssembly specification. 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 53–65. ACM, 2018. doi:10.1145/3167082.

30 Conrad Watt, Maja Trela, Peter Lammich, and Florian Märkl. Wasmref-isabelle: A verified monadic interpreter and industrial fuzzing oracle for WebAssembly. *Proc. ACM Program. Lang.*, 7(PLDI):100–123, 2023. doi:10.1145/3591224.

31 Joachim Zahnentferner. Chimeric ledgers: Translating and unifying UTXO-based and account-based cryptocurrencies. *Cryptology ePrint Archive, Report 2018/262*, 2018. URL: https://eprint.iacr.org/2018/262.

A Governance helper calculations

The design of the hash protection mechanism is elaborated here. The issue at hand is that different actions of the same type may override each other, and they allow for partial modifications to the state. So if arbitrary actions were allowed to be applied, the system may end up in a particular state that was never intended and voted for.

In the original design of the governance system, the fix for this issue was to allow only a single governance action of each type to be enacted per epoch. This restriction is a potentially severe limitation and may open the door to some types of attacks.

The final design instead requires some types of governance actions to reference the ID of the parent they are building on, similar to a Merkle tree. Then, in a single epoch the system can take arbitrarily many steps down that tree, and since IDs are unforgeable, the system is only ever in a state that was publically known prior to voting.

There are two governance actions where this mechanism is not required, because they either commute naturally or they do not actually affect the state. For these it is more convenient to not enforce dependencies.

```
NeedsHash : GovAction → Type
NeedsHash NoConfidence              = GovActionID
NeedsHash (NewCommittee _ _ _)      = GovActionID
NeedsHash (NewConstitution _ _)     = GovActionID
NeedsHash (TriggerHF _)             = GovActionID
NeedsHash (ChangePParams _)         = GovActionID
NeedsHash (TreasuryWdrl _)          = ⊤
NeedsHash Info                      = ⊤

HashProtected : Type → Type
HashProtected A = A × GovActionID
```

The two functions adjusting the state in GOV are addVote and addAction.

- addVote inserts (and potentially overrides) a vote made for a particular governance action by a credential in a role.
- addAction adds a new proposed action at the end of a given GovState, properly initializing all the requiered fields.

```
addVote : GovState → GovActionID → GovRole → Credential → Vote → GovState
addVote s aid r kh v = map modifyVotes s
  where modifyVotes = λ (gid , s') → gid , record s'
          { votes = if gid ≡ aid then insert (votes s') (r , kh) v else votes s'}

addAction : GovState
          → Epoch → GovActionID → RwdAddr → (a : GovAction) → NeedsHash a
          → GovState
addAction s e aid addr a prev = s :: (aid , record
  { votes = ∅ ; returnAddr = addr ; expiresIn = e ; action = a ; prevAction = prev })
```

B UTxO

Some of the functions used to define the UTXO and UTXOW state machines are defined here; inject is the function takes a Coin and turns it into a multi-asset Value [10].

```
outs : TxBody → UTxO
outs tx = mapKeys (tx .txid ,_) (tx .txouts)

minfee : PParams → Tx → Coin
minfee pp tx = pp .a * tx .body .txsize + pp .b


consumed : PParams → UTxOState → TxBody → Value
consumed pp st txb
  = balance (st .utxo | txb .txins)
  + txb .mint
  + inject (depositRefunds pp st txb)

produced : PParams → UTxOState → TxBody → Value
produced pp st txb
  = balance (outs txb)
  + inject (txb .txfee)
  + inject (newDeposits pp st txb)
  + inject (txb .txdonation)


credsNeeded : Maybe ScriptHash → UTxO → TxBody → ℙ (ScriptPurpose × Credential)
credsNeeded p utxo txb
  = map (λ (i , o) → (Spend i , payCred (proj₁ o))) ((utxo | txins) )
  ∪ map (λ a → (Rwrd a , RwdAddr.stake a)) (dom $ txwdrls .proj₁)
  ∪ map (λ c → (Cert c , cwitness c)) (fromList txcerts)
  ∪ map (λ x → (Mint x , inj₂ x)) (policies mint)
  ∪ map (λ v → (Vote v , GovVote.credential v)) (fromList txvote)
  ∪ (if p then (λ {sh} → map (λ p → (Propose p , inj₂ sh)) (fromList txprop))
      else ∅)
  where open TxBody txb

witsVKeyNeeded : Maybe ScriptHash → UTxO → TxBody → ℙ KeyHash
witsVKeyNeeded sh = mapPartial isInj₁ ∘₂ map proj₂ ∘₂ credsNeeded sh
```

```
scriptsNeeded : Maybe ScriptHash → UTxO → TxBody → ℙ ScriptHash
scriptsNeeded sh = mapPartial isInj₂ ∘₂ map proj₂ ∘₂ credsNeeded sh
```

C Advancing epochs

The NEWEPOCH state machine is responsible for detecting epoch changes: either the epoch remains unchanged (NEWEPOCH-Not-New), or the immediately next epoch is reached and the state is updated subject to some ratification requirements (NEWEPOCH-New).

```
NEWEPOCH-New :
  • e ≡ suc lastEpoch
  • record { currentEpoch = e ; treasury = treasury ; GState gState ; NewEpochEnv Γ }
      ⊢ ⟦ es ⊗ ∅ ⊗ false ⟧ ⇀⦇ govSt' ,RATIFY∗ ⦈ fut'
    ────────────────────────────────────────────────
    Γ ⊢ nes ⇀⦇ e ,NEWEPOCH ⦈ ⟦ e ⊗ acnt' ⊗ ls' ⊗ es ⊗ fut' ⟧

NEWEPOCH-Not-New :
  e ≢ suc lastEpoch
  ────────────────────────────────────────────────
  Γ ⊢ nes ⇀⦇ e ,NEWEPOCH ⦈ nes
```

Formally Verifying the Safety of Pipelined Moonshot Consensus Protocol

M. Praveen
Chennai Mathematical Institute, India
ReLaX, Chennai, India

Raghavendra Ramesh
Supra Research, Brisbane, Australia

Isaac Doidge
Supra Research, Brisbane, Australia

Abstract

Decentralized Finance (DeFi) has emerged as a contemporary competitive as well as complementary to traditional centralized finance systems. As of 23rd January 2024, per Defillama [6] approximately USD 55 billion is the total value locked on the DeFi applications on all blockchains put together.

A Byzantine Fault Tolerant (BFT) State Machine Replication (SMR) protocol, popularly known as the consensus protocol, is the central component of a blockchain. If forks are possible in a consensus protocol, they can be misused to carry out double spending attacks and can be catastrophic given high volumes of finance that are transacted on blockchains. Formal verification of the safety of consensus protocols is the golden standard for guaranteeing that forks are not possible. However, it is considered complex and challenging to do. This is reflected by the fact that not many complex consensus protocols are formally verified except for Tendermint [4] and QBFT [5].

We focus on Supra's Pipelined Moonshot consensus protocol. Similar to Tendermint's formal verification, we too model Pipelined Moonshot using IVy and formally prove that for all network sizes, as long as the number of Byzantine validators is less than 1/3, the protocol does not allow forks, thus proving that Pipelined Moonshot is safe and double spending cannot be done using forks. The IVy model and proof of safety is available on [1].

2012 ACM Subject Classification Networks → Protocol testing and verification; Theory of computation → Logic and verification; Theory of computation → Automated reasoning

Keywords and phrases Blockchain consensus, Safety, Formal verification

Digital Object Identifier 10.4230/OASIcs.FMBC.2024.3

Supplementary Material *Model (Source-code)*: `https://github.com/Entropy-Foundation/suprabft-fv/tree/master/suprabft`

Funding *M. Praveen*: Funded by Supra

Acknowledgements We acknowledge and thank Chandradeep Dey and Namrata Reddy, who were part of this project during its initial phase. We acknowledge Supra Research for funding M. Praveen, the academic partner of this project and providing other support.

1 Introduction

Public blockchains are revolutionising modern society by rebranding traditional services mainly the traditional finance based services, and offering them on a "decentralized trust" platform. Here no single entity need be trusted as the network is typically open for permissionless participation and tolerates malicious behaviour of the participants up to a certain threshold. Though blockchains are being adopted by multiple domains of applications, finance or Decentralised Finance (DeFi), happens to be the *killer* application that has shot the

© Supra. This work, as part of the collaborative efforts of M. Praveen, Raghavendra Ramesh and Isaac Doidge, falls under the intellectual property rights asigned to Supra in accordance with their agreements.;
licensed under Creative Commons License CC-BY 4.0
5th International Workshop on Formal Methods for Blockchains (FMBC 2024).
Editors: Bruno Bernardo and Diego Marmsoler; Article No. 3; pp. 3:1–3:16
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

blockchain technology to fame as well as towards a popular adoption. As of 23rd January 2024, per Defillama [6] approximately USD 55 billion is the total value locked on the DeFi applications on all blockchains put together.

Every node in a blockchain network runs a *consensus protocol*, more precisely known as *state machine replication (SMR)* protocol, that enables that node to transition from one blockchain state to the next in a consistent way so that no two nodes in the network end up in different states after processing the same sequence of transitions. The transitions are the clients' submitted ledger transactions that are batched into a block. The sequence of transitions form the chain of blocks, hence the name blockchain. We are interested in consensus protocols in a *partially synchronous network* setting. In this setting it is well known that an SMR protocol tolerates up to one-third of the network nodes being Byzantine – nodes that may crash or deviate arbitrarily from the protocol but are assumed to be unable to break cryptographic primitives like signatures.

Many such protocols have been proposed as well as successfully been adopted in practice such as [10, 11, 27, 15, 20] but only a few protocols have been formally verified to the best of our knowledge: Both the safety and liveness of Tendermint [4] have been formally verified using Microsoft IVy [22]. The safety of QBFT (called IBFT earlier) has been verified [5] using Microsoft's Dafny [18].

Pipelined Moonshot [12] is a novel rotating leader-based Byzantine fault tolerant SMR protocol that leverages *optimistic proposals* to achieve a high block throughput – one block per network hop, and the lowest block finalization latency – 3 network hops, in the scenario of a normal path. It is well known that designing protocols and proving them correct by hand are notoriously prone to errors as many critical errors have been found even in peer-reviewed distributed protocols (see [25] and references therein). In this paper, we focus on formally proving the safety of this protocol.

Safety of a BFT SMR protocol is a critical requirement, ensuring that any two honest processes agree on the set of transactions executed and the order in which they are executed. Formally, if two honest processes have committed chains of blocks, then one of the chains must be (not necessarily strict) prefix of the other one. When a protocol loses safety, forks in the blockchain are possible, essentially yielding to the possibility of *double spending* which is catastrophic to the finances built on top of this blockchain. Pipelined Moonshot protocol is proved to be safe and live [12] with a handwritten proof. The goal of this project is to provide a proof of safety in a formal verification tool.

Our Contributions.

- We provide formal specification of the Pipelined Moonshot protocol in IVy [22], serving as a reference for any implementation.
- We formally verify safety of Pipelined Moonshot successfully. This makes the formal specification a safety-error-free basis for any implementations to be developed.
- We identify several invariants of the protocol to prove it safe. Invariants are useful in generating test cases to test implementations of the protocol [30, 28].
- We record our experience in the form of challenges faced and the corresponding mitigations used. Learning from this experience we extend our wisdom as recommendations for applying formal verification to large projects.

2 Related Work

In this section we detail various formal verification approaches for consensus protocol verification and motivate our choice of IVy for verifying the safety of Pipelined Moonshot protocol.

Model checking approach typically models the protocol as some finite representation of a state transition system and expresses the correctness properties in some logic and enumerates exhaustively the state space validating against the given logical specifications. Various model checking tools like SPIN [2], TLC [3], Apalache [16] etc are popular. Typically for the consensus protocols of interest as long as the number of nodes in the system is not fixed the state space is unbounded and generally does not yield a decidable algorithm to model check. There are bounded model checking approaches where the number of nodes is fixed typically and that yields a finite state machine against which the correctness properties are checked. For instance, [9] model checks the block synchronization protocol of Tendermint after fixing the number of nodes in the network using TLC and Apalache model checker. We are focused on the general problem of safety of the Pipelined Moonshot with no bounds on the network, hence model checking is not applicable and bounded model checking is not satisfactory.

There are other approaches that identify a *small model property* in the given protocol verification problem and apply model checking against the small model. The small model property of a protocol P essentially is a bound on the number of nodes, say k, such that the satisfaction of a property θ by P when run with k nodes imply that P satisfies θ for all $n \geq k$. The threshold automata approach of [17] leverages this and builds a counter abstraction by counting the number of processes in each state. This has been applied to the verification of DBFT [25] asynchronous consensus protocol. However this approach is known to be hard and has not been applied so far for any of the partially synchronous BFT consensus protocols, and we too could not find any direct ways of applying this approach to the safety verification of Pipelined Moonshot.

We now turn to the deductive verification tools. In this approach, the protocol is modeled in some logic (typically first-order logic, its fragments or extensions) and the properties to be verified are also written in the same logic. From these, formulas called Verification Conditions (VCs) are generated, whose unsatisfiability implies that the protocol has the desired property. With interactive theorem provers, proof of unsatisfiability is developed in a proof system (such as natural deduction system or its variants). With automated deductive verification tools, proof of unsatisfiability is given by Satisfiability Modulo Theory solver (SMT solver).

TLA+ [3] supports a very expressive logic called TLA – Temporal Logic of Actions, for specifying state machines and properties. We found that expressing the Pipelined Moonshot protocol in TLA+ to be very complex and huge, and so also the verification in TLAPS – TLA Proof System, to be effortful as each and every lemma has to be proven more or less interactively. We were on the look out for solvers that push more automation and lessen the interaction with the solver. Another requirement of us was that the formal specification should be close to the real world programming languages so that the developer community may be comfortable using the formal specification as the basis for their implementation. We found the TLA+ specifications to be far from the interest of the developer community, unless they are trained specifically towards verification.

Dafny [18] is a verification-aware programming language that facilitates specifying pre and post conditions for procedures and verify at compile time. Correctness by construction is the philosophy here. The code may also be compiled to regular programming languages like C#, Java, JavaScript, Go and Python. The safety of QBFT (previously known as IBFT [20]) has been verified using Dafny.

It is also well known that the formal verification of distributed protocols is an arduous effort. Hence we were on the look out for a tool that maximally uses automation in proof building and we found **IVy** [22] to fit the bill. IVy is a language and a tool for the formal

specification and verification of distributed systems. IVy supports deductive verification using automated provers such as Z3 [8], model checking, automated testing, manual theorem proving and generation of executable code. In order to achieve greater verification productivity, a key design goal for IVy is to allow the engineer to apply automated provers in the realm in which their performance is relatively predictable, stable and transparent. In particular IVy focuses on the use of decidable fragments of first-order logic. IVy supports modularisation of the specifications and proofs, aiding their readability and also ensuring that formulas passed to provers are in decidable fragments. This helps to some extent in getting the provers to return with answers quickly.

As IVy embodies an imperative language, the protocol specification in IVy serves as a sound reference for any implementation. Note that the safety of Tendermint has been verified using IVy [4]. For all these reasons we favoured IVy as the formal verification tool for verifying the safety of Pipelined Moonshot.

To make proofs easier, Pretend Synchrony [26] takes another route of reducing the problem of verifying asynchronous distributed protocols to the problem of verifying synchronous distributed protocols. However it has been applied only in the setting of crash faults setting but not in Byzantine faults setting, which is the focus of this paper.

3 Safety of Pipelined Moonshot Consensus

In this section we summarise Pipelined Moonshot and elucidate the scope of our formal verification endeavour.

Pipelined Moonshot

Pipelined Moonshot [12] is a chain-based, rotating leader Byzantine Fault Tolerant (BFT) State Machine Replication (SMR) protocol optimized for wide-area networks. It satisfies the safety and liveness properties of SMR under the partially synchronous network model [13] given at most f of the n total participants in the protocol (which we will call *validators*) are Byzantine, such that $f < \frac{n}{3}$. Without loss of generality, we assume that $n = 3f + 1$ for the rest of the paper and use the term *quorum* to refer to a set of $2f + 1$ validators. The full details of Pipelined Moonshot's setting are provided in [12].

The protocol, presented in Figure 1 as given in [12], constructs a chain of blocks of client transactions (or some abstraction thereof) over a sequence of numbered views advanced by quorum decisions in the form of certificates. In Pipelined Moonshot, a view, say v, may produce two types of certificates; a Quorum Certificate $\mathcal{C}_v(B_k)$ comprised of $2f + 1$ votes for some block B_k (where k is the position or *height* of B in the blockchain) proposed for v, or a Timeout Certificate $\mathcal{TC}_v$ comprised of $2f + 1$ timeout messages for v. A Pipelined Moonshot validator in view v votes for B_k when it receives B_k in a valid proposal (described momentarily) and sends a timeout message for v, denoted $\mathcal{T}_v$, that contains its locked QC – i.e., the QC with the highest view that it has observed so far – when it fails to exit v before its view timer expires or when it observes evidence that at least one honest validator has already sent $\mathcal{T}_v$. These rules together ensure that the protocol continually generates new certificates, preventing it from halting.

A validator that receives a certificate for view v advances its local view to $v + 1$ and resets its view timer. If it enters $v + 1$ via a QC then it also locks the QC and multicasts it to ensure that its peers enter the view promptly. Otherwise, if it enters $v + 1$ via $\mathcal{TC}_v$ then it unicasts this certificate to the designated leader for $v + 1$, denoted L_{v+1}, enabling it to enter the view and propose promptly.

A Pipelined Moonshot node P_i runs the following protocol whilst in view v:

1. **Propose.** Upon entering v, the leader L_v proposes using one of the following rules:
   a. **Normal Propose.** If L_v entered v by receiving $\mathcal{C}_{v-1}(B_{k-1})$, multicast $\langle \mathsf{propose}, B_k, \mathcal{C}_{v-1}(B_{k-1}), v\rangle$ such that B_k extends B_{k-1}.
   b. **Fallback Propose.** If L_v entered v by receiving $\mathcal{TC}_{v-1}$, multicast $\langle \mathsf{fb\text{-}propose}, B_k, \mathcal{C}_{v'}(B_{k-1}), \mathcal{TC}_{v-1}, v\rangle$ such that $\mathcal{C}_{v'}(B_{k-1})$ is the highest ranked certificate in $\mathcal{TC}_{v-1}$ and B_k extends B_{k-1}.
2. **Vote.** P_i votes at most twice in view v when the following conditions are met:
   a. **Optimistic Vote.** Upon receiving the first optimistic proposal $\langle \mathsf{opt\text{-}propose}, B_k, v\rangle$ where B_k extends B_{k-1}, if (i) $\mathsf{timeout_view}_i < v-1$, (ii) $\mathsf{lock}_i = \mathcal{C}_{v-1}(B_{k-1})$ and (iii) P_i has not voted in v, multicast an optimistic vote $\langle \mathsf{opt\text{-}vote}, H(B_k), v\rangle_i$ for B_k.
   b. After executing *Advance View* and *Lock* with all embedded certificates, vote once when one of the following conditions are satisfied:
      i. **Normal Vote.** Upon receiving the first normal proposal $\langle \mathsf{propose}, B_k, \mathcal{C}_{v-1}(B_h), v\rangle$, if (i) $\mathsf{timeout_view}_i < v$, (ii) B_k directly extends B_h and (iii) P_i has not sent an optimistic vote for an equivocating block $B'_{k'}$ in v, multicast $\langle \mathsf{vote}, H(B_k), v\rangle_i$ for B_k.
      ii. **Fallback Vote.** Upon receiving the first fallback proposal $\langle \mathsf{fb\text{-}propose}, B_k, \mathcal{C}_{v'}(B_h), \mathcal{TC}_{v-1}, v\rangle$ if (i) $\mathsf{timeout_view}_i < v$ and (ii) B_k directly extends B_h and $\mathcal{C}_{v'}(B_h)$ is the highest ranked certificate in $\mathcal{TC}_{v-1}$, multicast $\langle \mathsf{fb\text{-}vote}, H(B_k), v\rangle_i$ for B_k.
3. **Optimistic Propose.** If P_i is L_{v+1} and voted for B_k in view v, multicast $\langle \mathsf{opt\text{-}propose}, B_{k+1}, v+1\rangle$ where B_{k+1} extends B_k.
4. **Timeout.** If $\mathsf{view\text{-}timer}_i$ expires and P_i has not already sent $\mathcal{T}_v$, then multicast $\langle \mathsf{timeout}, v, \mathsf{lock}_i\rangle_i$ and set $\mathsf{timeout_view}_i = \max(\mathsf{timeout_view}_i, v)$. Additionally, upon receiving $f+1$ distinct $\langle \mathsf{timeout}, v', _\rangle_*$ messages or $\mathcal{TC}_{v'}$ such that $v' \geq v$ and not having sent $\mathcal{T}_{v'}$, multicast $\langle \mathsf{timeout}, v', \mathsf{lock}_i\rangle_i$ and set $\mathsf{timeout_view}_i = \max(\mathsf{timeout_view}_i, v')$.
5. **Advance View.** P_i enters v' where $v' > v$ using one of the following rules:
   - Upon receiving $\mathcal{C}_{v'-1}(B_h)$. Also, multicast $\mathcal{C}_{v'-1}(B_h)$.
   - Upon receiving $\mathcal{TC}_{v'-1}$. Also, unicast $\mathcal{TC}_{v'-1}$ to $L_{v'}$.

   Finally, reset $\mathsf{view\text{-}timer}_i$ to 3Δ and start counting down.

P_i additionally performs the following actions in any view:

1. **Lock.** Upon receiving $\mathcal{C}_v(B_k)$ whilst having $\mathsf{lock}_i = \mathcal{C}_{v'}(B_{k'})$ such that $v > v'$, set lock_i to $\mathcal{C}_v(B_k)$.
2. **Direct Commit.** Upon receiving $\mathcal{C}_{v-1}(B_{k-1})$ and $\mathcal{C}_v(B_k)$ such that B_k extends B_{k-1}, commit B_{k-1}.
3. **Indirect Commit.** Upon directly committing B_{k-1}, commit all of its uncommitted ancestors.

Figure 1 The Pipelined Moonshot Protocol [12].

Upon entering $v+1$, L_{v+1} creates a new block, say B_l, and multicasts it in a proposal that depends on the type of certificate it used to enter the view. If the view change was triggered by $\mathcal{C}_v(B_k)$, then B_l *directly extends* B_k (i.e. $l = k+1$ and B_l contains the hash digest of B_k) and L_{v+1} multicasts a Normal Proposal containing both B_l and $\mathcal{C}_v(B)$. Otherwise, B_l extends the block certified by the QC with the highest view included in $\mathcal{TC}_v$ and L_{v+1} multicasts a Fallback Proposal containing both B_l and $\mathcal{TC}_v$. A validator in $v+1$ that receives a proposal of either type from L_{v+1} that is constructed as previously described and has yet to either send a vote for an equivocating block (as described in Figure 1) or a timeout

message for $v+1$, multicasts a vote of the corresponding type for B_l for $v+1$. Importantly, Pipelined Moonshot ensures that votes cannot be aggregated into a QC unless they have the same type.

Upon voting for B_l, if the validator is L_{v+2} then it also creates an Optimistic Proposal for $v+2$ containing a new block that extends B_l, presuming that B_l will be certified. A validator that receives this proposal votes for it once in $v+2$ if it has not yet voted in $v+2$ and it entered the view via $\mathcal{C}_{v+1}(B_l)$ without having sent a timeout message for $v+1$. Optimistic Proposals, a distinguishing feature of Moonshot protocols, allow votes for the current view to be disseminated in parallel with a proposal for the next view when both leaders are honest. Comparatively, prior protocols require a leader to receive a certificate for the previous view before proposing, inherently sequentializing these actions.

A validator that observes the certification of a block and its immediate successor in the chain for adjacent views commits the block by permanently appending it to its local copy of the blockchain.

Safety

An SMR protocol is *safe* if it ensures that no two validators commit divergent blockchains. Let the local blockchain of validator P_i be denoted by $\mathbf{B}_i$. More formally, the safety property states that for every run of the protocol, and for each pair of honest validators $(P_i, P_j) \in \mathcal{V} \times \mathcal{V}$, either $\mathbf{B}_i$ is a (not necessarily strict) prefix of $\mathbf{B}_j$ or vice-versa.

The Pipelined Moonshot [12] paper contains the handwritten proof of safety and liveness. As well established in the literature some errors may potentially be present in the handwritten proofs that could go overlooked. A recent example is the Chord [24] protocol for distributed hash tables which, despite having more than 6000 citations, was shown incorrect by Zave [29] almost a decade after its publication. Since only formal verification can conclusively guarantee the absence of errors in a protocol, we aimed at developing mechanically verifiable proofs of the safety of Pipelined Moonshot.

4 Formal Specification and Verification using IVy

In this section, we first present some of the preliminaries of IVy modeling, secondly we present an high level overview of the formal IVy specification of the Pipelined Moonshot consensus protocol, and then finally present the structure of our safety proof.

4.1 IVy modeling setup

IVy is a language and a tool for the formal specification and verification of distributed systems. Systems are represented as state transition machines. States are multi-sorted first-order structures, with relations and functions. Transitions specify how the state is mutated. Any update definable in first-order logic is supported. Update instructions can be given in sequence one after another, giving the syntax the flavor of a developer-friendly imperative programming language. Multiple update instructions can be grouped together into an *action*, a keyword in IVy that is used to denote state transition specifications.

The system under consideration is typically split into multiple modules, with internal states of a module not allowed to be modified directly by other modules. One module can call actions of another module, passing parameters. Modules can reason about one another using *assume-guarantee* specifications, which are formulas specifying properties of the modules' states. Properties of the overall system has to be proved by writing *inductive invariants*,

which are properties satisfying two conditions – initiation and inductiveness. Initiation means that the initial state of the system satisfies the invariant. Inductiveness means that if any of the actions are executed in any state that satisfies the invariant, the resulting state also satisfies the invariant.

Modules in IVy, apart from modularising the protocol specification and proofs, serves another deeper purpose. Multiple formulas used in the proof may together necessitate the use of logics that are undecidable. Modules in IVy allow proving different properties in isolation, ensuring that formulas supporting one invariant are invisible to other modules. This will allow users to control which formulas are passed to the underlying SMT solvers together, so that all calls to the SMT solver are within decidable fragments of first-order logic.

Only a high level abstract specification of the protocol is modeled and verified. Some implementation details are hence modeled with Boolean abstractions. For example, timers used in the protocol are replaced by Boolean propositions that indicate whether or not a timer has expired. In the IVy model, the Boolean proposition can switch value anytime non-deterministically to simulate a timer getting expired, instead of tracking the actual time elapsed since the last reset. This is a sound abstraction for proving safety.

Another abstraction we have adapted from the literature is handling quorums [19]. The protocol specification mandates that a validator needs to receive messages from a super majority of all validators (two-thirds of the entire set) in order to achieve a quorum. Verifying this detail would require having arithmetic in the formulas passed on to SMT solvers, potentially affecting the solver's performance. Instead, what is modeled is the *quorum intersection property* [19] – any two quorums have at least one common honest validator. It is this property of quorums that are mainly used in correctness proofs and is modeled in IVy as an axiom, avoiding the usage of arithmetic.

Validators receive messages from the network and verify their authenticity by checking digital signatures. It is assumed that Byzantine validators cannot break cryptographic primitives and hence they cannot forge signatures of honest validators. Checking digital signatures is not modeled in IVy – the model assumes messages sent by honest validators are authentic. The model also disallows byzantine validators to send messages on behalf of other honest validator, though they can send any kind of message on behalf of themselves or other byzantine validators, even if such a message is not mandated to be sent by the protocol specification.

4.2 Pipelined Moonshot Specification

We have published our IVy specification and the formal proof of safety of Pipelined Moonshot online on GitHub [7]. Following are the main modules in our IVy specification of Pipelined Moonshot:

Types. This module contains the declarations of the data types used in the IVy specification. The types *round_t*, *height_t* are declared to be instances of *ubd_seq*, a small modification of *unbounded_sequence*, which are finite but unbounded total linear orders. Round is the technical term used in our IVy model for view as used in the protocol specification [12]. The type *process_index_t* is declared to be an instance of *iterable*, which allows a collection of validators to be iterated in a loop in IVy models. The above types are used conventionally while modeling protocols in IVy. Other types declared correspond to message types specified in the protocol specification: *block_t*, *quorum_t*, *qc_t*, *tc_t*. Common properties of these types are also written in this module, including the quorum intersection axiom.

Network. This module models the network through which validators interact. It is almost same as the network model in Tendermint's IVy model [4], except for the kind of messages that can be sent. Here the kind of messages that can be sent are *normal proposal, fallback proposal, optimistic proposal, normal prepare, fallback prepare, optimistic prepare, quorum certificate, timeout certificate, timeout* and *weak timeout certificate*. A *timeout certificate* is a collection of *timeout* messages from a two thirds majority of validators, whereas a *weak timeout certificate* is a collection *timeout* messages from a number of validators at least one more than the number of Byzantine validators. The network model is that of any asynchronous one, where messages can be dropped or delivered multiple times and/or out of order.

Moonshot. The IVy specification of the Pipelined Moonshot is provided in this module. State variables of individual validators are declared and updates to the state variables are performed in response to specific events as specified in Figure 1. The details of this module are provided in Appendix A.

Quorum verification. In implementation, the integrity of a quorum of messages received by a validator is verified by checking digital signatures accompanying the messages. Here, the integrity is checked by verifying that all honest members of a quorum have actually sent the corresponding messages. It is done in this module using the concept of *monitors* provided by IVy – they are additional updates to state variables that are performed whenever an action is performed by the protocol. This module contains monitors that record prepare and timeout messages sent by the validators. When a validator receives a quorum or timeout certificate, its integrity is checked by verifying from the records that all honest members of the quorum have actually sent the corresponding prepare or timeout message. This can be thought of as some kind of a central authority with a global view of all validators, who records all messages sent by the validators. Of course there is no such central authority in real implementation; it is only modeled here for the sake of proving safety.

Safety. The safety module specifies the desired safety property in the form of an inductive invariant. Numerous supporting invariants are also included here, as detailed in the next sub-section. Following is a code snippet, that states the main safety property.

```
isolate full_safety = {
    relation blockchain_prefix(N1:process_index_t, N2:process_index_t)

    # the latest block committed to b_v by N1 is equal to or ancestor of the
    latest block committed by N2. All blocks committed by N1 are also
    committed by N2. Any block committed by N2 but not by N1 is a descendant
    of the latest block committed by N1
    definition blockchain_prefix(N1,N2) = ...

    # this is the full safety property of the pipelined moonshot protocol: for
    any two honest processors N1,N2 the chain committed by N1 is a prefix of
    N2 or vice versa
    invariant forall N1,N2:process_index_t. is_good(N1) & is_good(N2) ->
    (blockchain_prefix(N1,N2) | blockchain_prefix(N2,N1))
} with block_t, verify_quorum, certified_block_ancestor_m1,
all_ancestors_committed, committed_blocks_ancestors,
latest_committed_ancestors, commit_to_chain, commit_to_chain_m1
```

The definition of the relation blockchain_prefix above is not shown fully due to lack of space, but its intention is captured in the comment above. The *with* clause above lists the names of other isolates, containing invariants supporting this one.

Table 1 provides some statistics of these modules. Note that a typical line in safety.ivy is much longer than those in other files, since one whole invariant is written in one line of safety.ivy. The files also have extensive comments serving the purpose of readability and documentation. There are a total of 190 invariants and 23 monitors. A rough estimate of the ratio of sizes of program code vs. proof is 1:3. Verifying the safety of Pipelined Moonshot took about 140 man hours after the protocol specification itself was stabilized. About 10% of this was needed to model the protocol and the rest to complete the proofs.

Table 1 Modules and their Lines of code.

Module	Contents	Lines
Types	Extended data types	338
Network	Network model	110
Moonshot	Pipelined Moonshot SMR protocol	642
Quorum verification	Validating messages sent by quorum members	165
Safety	Inductive invariants proving safety	1309
	Total	2564

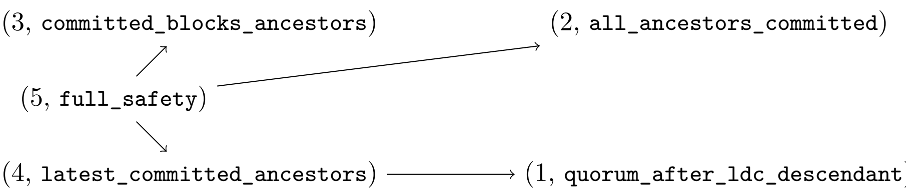
4.3 Structure of the Safety Proof

Mechanically-checked proofs are developed interactively in a dialogue between a Verification engineer and the proof assistant – IVy. The engineer gives the desired specification of the model and of the property, IVy attempts to prove that the model satisfies the property. It may prove, then all is well, it may fail showing a counterexample, or it may not come back for a reasonable amount of time. When satisfiability fails it shows logical errors in the protocol. When it takes an unreasonable amount of time, the engineer has to creatively craft some lemmas that aids the machine in its proof search. This is the standard iterative approach of building mechanised proof.

IVy could not prove the safety specification directly (as is typical). We had to write all the intermediate lemmas given in [12] and many more. We first outline the main steps in the handwritten safety proof.

1. If an honest validator executes direct commit of a block B as given in point 2 at the bottom of Figure 1, then any subsequent block that achieves a quorum is a descendant of B. This is proved in [12, Lemma 2, Lemma 3].
2. If an honest validator commits a block, it also commits all of its ancestors. This is implicit in [12].
3. For any two blocks committed by an honest validator, one is an ancestor of the other. This does not directly correspond to any result stated in [12], but essential in our IVy proof.
4. If B_i (resp. B_j) is the latest block committed by an honest validator v_i (resp. v_j), then B_i is an ancestor of B_j or vice-versa. This is a corollary of item 1 above.
5. If there were two blocks that were divergent, one would be an ancestor of the other (by item 3 above) and both would be ancestors of the latest committed block (by item 4 above). Hence, both would be committed by all honest validators (by item 2 above), contradicting the hypothesis that they are divergent. This argument is essentially the proof of [12, Theorem 3].

IVy verifies that properties are inductive invariants by generating formulas in Finite Almost Uninterpreted (FAU) fragment of first-order logic and passing them on to Z3 [19]. Trying to prove too many properties in one step often degrades the performance of the SMT solver. To overcome this, IVy allows to group together a small number of properties in an *isolate*, specifying other isolates as supporting invariants. When verifying one isolate, other isolates that it depends on are assumed to be true. The dependencies can be checked later. The safety invariants in our model are structured into several isolates. The top level of this structure follows the structure of the handwritten proof that is summarized above, and is illustrated below. Here, (5, `full_safety`) means that the point 5 above is proved in the isolate `full_safety`, likewise for other nodes. The arrow from (5, `full_safety`) to (3, `committed_blocks_ancestors`) means that the isolate `full_safety` depends on other isolates: `committed_blocks_ancestors` being one of them.



The isolate `quorum_after_ldc_descendant` is technically the most involved result in both the handwritten proof and IVy proof. This result is proved in IVy by induction on rounds. The principle of induction is taken to be an axiom and applied to the main invariant in the isolate `quorum_after_ldc`. It states that if a block B is committed directly by an honest validator, then any block proposed in later rounds that achieves a quorum has a parent proposed in the same round as B or later rounds. Proving the invariants in the isolate `quorum_after_ldc` itself is lengthy, indirectly involving around 30 other isolates.

5 Challenges

The development and handwritten proof of safety and liveness of the pipelined Moonshot protocol underwent many cycles (some modifications to ensure liveness and some for simplifying the specification and proofs). This naturally resulted in iterating and refining the IVy specification too. The process of formally verifying safety (including analyzing counter examples given by IVy) uncovered some points in the specifications and proofs that were ambiguous and helped better understand many details that were implicit in the handwritten proofs.

Here we document some of the challenges faced in such a verification effort.

Transitive closure

Ancestor relation is the binary transitive closure of the parent relation. For a validator to commit a block to its canonical chain, the block and its ancestors must have got quorums. The statement and proof of the property in the isolate quorum_after_ldc_descendant uses the ancestor relation. Thus, many crucial parts of the model and safety proof depend on the ancestor relation. However, transitive closure of binary relations are not definable in first-order logic. To overcome this, we adapted a known technique [19]. If a binary relation is known to be the transitive closure of another base relation, then under some conditions the base relation can be defined from its transitive closure in first-order logic. To use this

technique, a monitor in the isolate certified_block_ancestor_m1 tracks when blocks get quorums and records the ancestor relation among them. Whenever a block gets quorum, the monitor updates the record, making the newly certified block a descendant of its parent block and all the parent block's ancestors. In the isolate certified_block_ancestor_m5, we verify that the base relation obtained from the relation recorded by certified_block_ancestor_m1 is indeed the parent relation. This challenge is not there for verifying Tenderming [4], where cross dependencies between isolates is lesser.

Nested subroutine calls

As in most programming languages, actions in IVy can invoke other actions, which can themselves invoke more actions and so on. We observed empirically that with higher depth of nesting of action invocations, the performance of the IVy verifier slows down considerably. The protocol specification use subroutines that are called from multiple sites and the natural way to model it would be to have similar actions in IVy called from multiple actions. However, the slowdown in performance was significant enough that we resorted to inlining the subroutine calls. Further work is needed to understand the causes and more elegant workarounds.

IVy verifier getting stuck without giving an answer

This challenge took up most of the time for executing this project. While verifying properties that were expected to be true, the IVy verifier would call the Z3 SMT solver, which would run for a long time without giving any answer. Such behaviour from SMT solvers cannot be entirely avoided, since they try to solve problems that have quite bad complexity theoretical lower bounds. There is no fixed template for handling this. Experience with using the tool and familiarity with the protocol being verified help a little bit. This is a challenge faced by most formal verification efforts; we felt it more since we had many invariants to prove, due to the complexity of the underlying protocol. Here are a few rules of thumb we resorted to, devised from trial and error.

Isolating the cause in the protocol. If the property being verified involved multiple steps in the protocol, we tried commenting out parts of the protocol and trying to verify the property. If the property was proved to be true/false after a particular section was commented out, then we could concentrate on that part to see what can be causing the SMT solver to diverge. This strategy helped us identify some subtle points that were implicitly assumed in the handwritten proof.

Explicitly writing intermediate results. We illustrate this with an example. In the isolate quorum_after_ldc_descendant_m7, the third invariant states that under some conditions, the block Bp is an ancestor of B. IVy could not prove this in a reasonable amount of time, so it was not clear whether it is due to lack of supporting invariants or because the SMT solver is diverging. We then added the first two invariants. The first one says that under the same condition, Bp is an ancestor of $Bp1$ and the second one says that additionally, $Bp1$ is an ancestor of B. With the first two invariants added, IVy successfully verifies all the three in short time. There are many more examples like this. The main invariant of quorum_after_ldc_descendant_m7 is inferred from a similar series of intermediate invariants, starting from quorum_after_ldc_descendant_m1, ending at quorum_after_ldc_descendant_m8 and then finally proving the invariant in quorum_after_ldc_descendant.

6 Recommendations

In this section we consolidate our experience with the safety verification of Pipelined Moonshot and attempt to distill some recommendations for applying formal verification techniques in proving the correctness of distributed systems.

- Compared to semi-interactive theorem provers like Coq, the manual effort required with IVy is lesser. The proofs had to be flattened out into small steps manageable by SMT solvers, as explained in the last challenge. More research efforts like [14] are needed to reduce the burden of manually working out minute details, letting users concentrate on understanding protocols and correctness proofs intuitively.
- With complex protocols involving correctness proofs using hundreds of invariants, the success of deductive verifications tools that call SMT solvers in the background depends crucially on modularization of the proof, so that every SMT call is restricted to a small number of closely related formulas. For this, it is important at the outset to have a good idea of how the modules are going to be structured and which modules are meant for what. If this is lacking during the initial phase, chasing minute details during the verification process can quickly lead to huge, monolithic, incomprehensible and unmanageable pile of candidate invariants. In earlier attempts at verifying Pipelined Moonshot, we were sometimes in situations where we changed an invariant written earlier to suitably support a newly written invariant, only to realize later that this change affected an earlier dependency. We had lost track of which invariants supported which others and small changes in one invariant affected seemingly unrelated ones elsewhere.
- A related point is to be disciplined while establishing inter-dependencies among modules, specifically isolates in IVy. If invariant 1 in isolate 1 needs invariant 2 in isolate 2 for support, it is tempting to mention the whole of isolate 2 as a dependency for isolate 1, instead of mentioning just invariant 2 of isolate 2. This may seem to be a time saver in the short term, but will result in unnecessary formulas (invariants in isolate 2 different from invariant 2) being passed to the SMT solver. Such unnecessary formulas can drastically degrade the performance of SMT solvers. Due to this, isolate 1 may pass all verification conditions in a short time currently but may not be able to do so in the future if additional invariants are added to isolate 2. If there is a group of invariants that are always together supporting other invariants, they should be recognized as such and grouped into an isolate, which is why this is related to the previous point of starting with well organized modules.
- The state of the art for formal verification of this scale very much requires experts with advanced knowledge of logic and related topics, who also need to understand the protocol being verified. An ideal team for formal verification would consist of experts with experience in using logic based verification tools on the one hand, and designers who understand the workings of the protocol at both abstract level and minute detail level on the other hand.

7 Conclusion

We have successfully verified the safety of a high performance and complex consensus protocol, namely Pipelined Moonshot, in IVy. This conclusively proves the absence of design or logic errors with respect to the protocol safety. Proving liveness is future work, possibly using [21] to reduce liveness to safety.

This effort has yielded a developer friendly formal specification of the Pipelined Moonshot protocol that helps for any implementation of Pipelined Moonshot to safely base on.

We recorded our experience in the form of challenges faced and the mitigations employed during this project. Learning from this experience we enumerate some recommendations for applying formal verification for large distributed protocols.

References

1. IVy modeling of Pipelined Moonshot and its proof of safety. https://github.com/Entropy-Foundation/suprabft-fv/tree/master/suprabft.
2. SPIN. https://spinroot.com/spin/whatispin.html.
3. TLA+. https://lamport.azurewebsites.net/tla/tla.html.
4. Defillama. https://galois.com/blog/2021/07/formally-verifying-the-tendermint-blockchain-protocol/, 2021.
5. Formal Verification of QBFT Safety. https://github.com/Consensys/qbft-formal-spec-and-verification, 2021.
6. Defillama. https://defillama.com, 2024.
7. Moonshot Formal Verification in IVy - GitHub Repository. https://github.com/Entropy-Foundation/suprabft-fv/tree/master/suprabft, 2024.
8. Z3 SMT Solver. https://www.microsoft.com/en-us/research/project/z3-3/, 2024.
9. Sean Braithwaite, Ethan Buchman, Igor Konnov, Zarko Milosevic, Ilina Stoilkovska, Josef Widder, and Anca Zamfir. Formal Specification and Model Checking of the Tendermint Blockchain Synchronization Protocol. In Bruno Bernardo and Diego Marmsoler, editors, *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*, volume 84 of *Open Access Series in Informatics (OASIcs)*, pages 10:1–10:8, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.FMBC.2020.10.
10. Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, University of Guelph, 2016.
11. Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.
12. Isaac Doidge, Raghavendra Ramesh, Nibesh Shrestha, and Joshua Tobkin. Moonshot: Optimizing chain-based rotating leader bft via optimistic proposals, 2024. arXiv:2401.01791.
13. 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.
14. Yotam MY Feldman, James R Wilcox, Sharon Shoham, and Mooly Sagiv. Inferring inductive invariants from phase structures. In *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II 31*, pages 405–425. Springer, 2019.
15. Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *FC*, pages 296–315, 2022.
16. Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. Tla+ model checking made symbolic. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi:10.1145/3360549.
17. Igor Konnov, Marijana Lazić, Helmut Veith, and Josef Widder. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, pages 719–734, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3009837.3009860.
18. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
19. Kenneth L. McMillan and Oded Padon. Deductive verification in decidable fragments with ivy. In Andreas Podelski, editor, *Static Analysis*, pages 43–55, Cham, 2018. Springer International Publishing.

20 Henrique Moniz. The istanbul bft consensus algorithm, 2020. arXiv:2002.03613.

21 Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. Reducing liveness to safety in first-order logic. *Proc. ACM Program. Lang.*, 2(POPL), 2017. doi:10.1145/3158114.

22 Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. *SIGPLAN Not.*, 51(6):614–630, June 2016. doi:10.1145/2980983.2908118.

23 Supra Research. Moonshot: Optimistic proposal for blockchain-based state machine replication. URL: https://supraoracles.com/news/moonshot-consensus/.

24 Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/383059.383071.

25 Pierre Tholoniat and Vincent Gramoli. *Formal Verification of Blockchain Byzantine Fault Tolerance*, pages 389–412. Springer International Publishing, Cham, 2022. doi:10.1007/978-3-031-07535-3_12.

26 Klaus v. Gleissenthall, Rami Gökhan Kıcı, Alexander Bakst, Deian Stefan, and Ranjit Jhala. Pretend synchrony: synchronous verification of asynchronous distributed programs. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290372.

27 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *PODC*, pages 347–356, 2019.

28 Yuan Yuan, Zeng Fanping, Zhu Guanmiao, Deng Chaoqiang, and Xiong Neng. Test case generation based on program invariant and adaptive random algorithm. In *Advances in Information Technology and Education: International Conference, CSE 2011, Qingdao, China, July 9-10, 2011, Proceedings, Part I*, pages 274–282. Springer, 2011.

29 Pamela Zave. Using lightweight modeling to understand chord. *SIGCOMM Comput. Commun. Rev.*, 42(2):49–57, March 2012. doi:10.1145/2185376.2185383.

30 Fanping Zeng, Qing Cao, Liangliang Mao, and Zhide Chen. Test case generation based on invariant extraction. In *2009 5th International Conference on Wireless Communications, Networking and Mobile Computing*, pages 1–4. IEEE, 2009.

A IVy Specification of the Pipelined Moonshot Protocol

The IVy specification of the Pipelined Moonshot is provided in the module **Moonshot**. The white paper [23] by Supra research describes the same protocol as [12] but in a format that is better suited to serve as a starting point for implementation. The structure of our IVy model closely follows the description in [23], so we use it in the following to explain the orgnization of the IVy model.

The Moonshot module consists of declarations of state variables to be maintained by honest validators as given in [23, Table II]. These are then followed by *actions*, the keyword in IVy used to denote updates to the state variables performed in response to specific events. Below we list the actions and the corresponding events specified in [23]. Here, f is the maximum number of Byzantine validators tolerated.

Following is a code snippet from the action `normal_proposal_processing`, broadcasting a prepare message.

Table 2 IVy actions and their corresponding subroutine in the specification.

Action	Subroutine in [23]	Triggering event
`qc_processing`	Algorithm 2 line 27	Receiving a quorum certificate
`optimistic_proposal_processing`	Algorithm 2 line 37	Receiving an optimistic proposal
`normal_proposal_processing`	Algorithm 2 line 49	Receiving a normal proposal
`timer_expire`	Algorithm 3 line 73	Timer expires
`timeout_sync`	Algorithm 3 line 76	Receiving timeout messages from $f+1$ validators
`tc_processing`	Algorithm 3 line 80	Receiving a timeout certificate
`fallback_proposal_processing`	Algorithm 3 line 89	Receiving a fallback proposal

```
# This function encodes the conditions necessary
for processing a normal proposal
function send_prepare_n_condition(B_pr:block_t, QC:qc_t) : bool
definition send_prepare_n_condition(B_pr, QC) = block_t.round(B_pr,r_c)
& a_f < r_c & t_l < r_c & (a_o < r_c  | b_o = B_pr) &
(forall B:block_t. forall R:round_t. qc_t.block(QC,B) &
block_t.round(B,R) -> block_t.parent(B_pr,B) & round_t.succ(R, r_c))

# The procedure in Line 49 of Algorithm 2, executed upon receiving
# a normal proposal
action normal_proposal_processing(b_pr:block_t, qc:qc_t) = {

    require received_proposal_n(b_pr, leader(r_c));
    require received_qc(qc);
    require block_t.cstd(b_pr);

    # Require that the timer has not yet expired for this round
    require ~ t_r;
    require ~ possessed_normal_for_round(r_c);

    #require that the parent of the proposed block b_pr is certified by
    # the accompanying QC qc
    require forall B:block_t. qc_t.block(qc,B) ->
    block_t.parent(b_pr,B);

    possessed_normal_for_round(r_c) := true;

    # If the accompanying qc is not yet processed yet, do that first
    if some b:block_t. qc_t.block(qc,b) & ~ processed_qc(b) {
        call qc_processing(qc);
    }

    # After processing the accompanying the qc, require that the
    # conditions in lines 50-56 of Algorithm are met
    require send_prepare_n_condition(b_pr,qc);
```

```
    # This condition verified by IVy ensures that the parent of the
    # proposed block b_pr is for a strictly lesser round
    ensure block_t.parent(b_pr,Bp) & block_t.round(Bp,Rp) ->
    Rp < r_c;

    #Line 58: propose optimistic
    ###### proposeOptimistic ############
    # Line 7,8 of Algorithm 1
    var rs := round_t.next(r_c);
    if leader(rs) = id & b_o ~= b_pr{

        #Lines 10-11 of Alorithm 1
        var b := block_t.block(rs,b_pr);
        var m : msg;
        m.kind := msg_kind.proposal_o;
        m.block := b;
        m.src := id;

        call shim.broadcast(id, m);
    }

    if send_prepare_n_condition(b_pr,qc) {

        # Line 59 of Algorithm 2: broadcast prepare normal message
        var m : msg;
        m.kind := msg_kind.prepare_n;
        m.block := b_pr;
        m.src := id;

        call shim.broadcast(id, m);

        # Line 60 of Algorithm 2: a_n is updated to the current
        # round
        a_n := r_c;
    }
}
```

Towards Mechanised Consensus in Isabelle

Elliot Jones
Department of Computer Science, University of Exeter, UK

Diego Marmsoler
Department of Computer Science, University of Exeter, UK

Abstract

A blockchain acts as a universal ledger for digital transactions between two parties that require no moderation from a third party. Such transactions are cheaper, quicker, and more secure with high traceability and transparency, with the decentralised structure of a blockchain network allowing for greater scalability and availability. For these reasons, blockchain is at the forefront of emerging technologies, with a wide variety of industries investing billions into the technology. A blockchains consensus protocol is what allows a blockchain network to be decentralised but can be subject to malicious behaviour and faults in its design and implementation that can lead to catastrophic effects like the DAO hack that resulted in a loss of $60 million. From this it is clear to see that the verifications of these protocols are paramount to ensure the safe use of blockchain. In this research, we formally verify the Proof-of-Work consensus protocol, used by Bitcoin, in Isabelle/HOL by modelling the blockchain as the longest branch in a binary tree and proving that the common prefix property holds with the assumption that the network is in majority honest. In this paper, we discuss the validity of our approach, key functions and lemmas we used to complete the proof, advantages and drawbacks of the model, related work and how this research can be taken further.

2012 ACM Subject Classification Security and privacy → Logic and verification

Keywords and phrases Formal Methods, Blockchain, Isabelle/HOL, Consensus, Verification, Theorem Provers

Digital Object Identifier 10.4230/OASIcs.FMBC.2024.4

Supplementary Material *Software (theory files)*: https://doi.org/10.5281/zenodo.10479776

Funding *Diego Marmsoler*: This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/X027619/1].

Acknowledgements We would like to thank the FMBC24 reviewers for the careful reading and constructive suggestions on the paper and the formalisation.

1 Introduction

Blockchain was first introduced to the world in 2008, when the illusive Satoshi Nakamoto published his paper on Bitcoin [26]. Using cryptographic hash functions and consensus protocols, blockchain allows parties to carry out digital transactions 'peer-to-peer', meaning no third-party is required to mediate and requires no trust between parties. This allows transactions to be faster, cheaper, and more secure with high traceability and transparency. Furthermore, the decentralised structure of blockchain allows for greater scalability and availability. Whilst its original purpose was for cryptocurrencies, blockchain has since been identified as a means to transform a variety of industries such as finance, healthcare and energy [24]. It is for these reasons that blockchain has become one of the most promising emerging technologies, with worldwide spending expected to grow from $6.6 billion in 2021 to an estimated $19 billion by 2024 [33].

However, blockchain presents its own unique challenges. The consensus protocol of a blockchain is what allows it to be decentralised and Byzantine Fault Tolerant (BFT), but these protocols can act as a vector of attack for malicious users. For example, a 51% attack is

© Elliot Jones and Diego Marmsoler;
licensed under Creative Commons License CC-BY 4.0
5th International Workshop on Formal Methods for Blockchains (FMBC 2024).
Editors: Bruno Bernardo and Diego Marmsoler; Article No. 4; pp. 4:1–4:22
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

when an attacker controls most of the computing power of a blockchain network, giving the attacker the opportunity to alter transactions on the blockchain and use the same money for multiple transactions in a double spending attack. An example of this was when Ethereum Classic suffered 51% attacks in 2019 and 2020, losing approximately $1.1 million [25] and $5.6 million [11] in double spending.

Moreover, the design and implementation of a blockchain and its consensus protocol led to challenges themselves, with the infamous DAO hack stemming from a smart contract flaw and resulting in a loss of $60 million [8]. Furthermore, A faulty consensus protocol would mean nodes would not be able to agree on the correct chain, allowing malicious actors to potentially change history and double spend without most of the network being malicious. Consequently, the need to verify blockchain consensus protocols is paramount to ensure the security of users. The birth of blockchain has caused a surge of investment into software verification. No organisation showcases this better than CertiK, founded in 2018 and now the world's leading company in Web3 security auditing. With a valuation of $2 billion and annual revenue of $40 million [13], Certik uses cutting-edge formal verification techniques to audit blockchains and smart contracts. The most popular consensus protocol in the world is Proof-of-Work (PoW) due to its usage by Bitcoin, the largest cryptocurrency in the world with a market capitalisation of over $700 billion [9]. PoW assumes that the majority of the network's computing power is acting honestly, making it susceptible to a 51% attack. Despite this, it is still important to verify that the protocol works as expected in the presence of a majority honest network. In doing so, we ensure that consensus will continue to hold as the blockchain network progresses through time and block stability will remain. A verified consensus protocol will promote confidence and encourage participation in the network by minimising faults and malicious behaviour.

In this paper, we formalise a general PoW protocol in Isabelle/HOL [28] and verify consensus by proving the common prefix property [12]. To this end, the paper provides the following main contributions:

1. We describe a general model for blockchain and its formalisation in Isabelle/HOL. A blockchain is modeled as a tree and accompanied by a function to check its validity (Subsubsection 3.2.4). The model provides abstract characterizations of minings (Subsection 3.3) and honest minings (Subsection 3.4) as special types thereof. The protocol is modeled as an inductively defined set of valid event traces, where each event is either a honest or dishonest mining (definition `traces` in Listing 11).

2. We formalise and discuss the common prefix property in Isabelle/HOL (Listing 13). This is an important safety property for consensus protocols which asserts that, once confirmed, blocks cannot be modified anymore.

3. We verify the property from our model. Thereby we discover important assumptions which are required for consensus, such as `b1` and `b2` in Listing 11 or the preconditions for `dishonest_induct` in Listing 11.

In Section 2, we highlight key blockchain properties that are essential to understand our verification. In Section 3, we explain our model, the functions and datatypes we have used, and the mining locales. In Section 4, we describe the blockchain locale and the proof of our consensus property. We conclude the paper with a discussion (Section 5) in which we highlight the simplifications and assumptions we have made for our model.

2 Background

As described in [26], a blockchain is a sequence of blocks, where each block contains a collection of transactions between parties in a network. A blockchain is initialised by a genesis block, a block that contains no transactions but has a unique hash value. To add a new block to the chain, a network participant must first assemble enough transactions into a block, then find the Merkle root hash of these transactions and combine it with the previous block's hash and a nonce to produce its own hash that meet the network's hash requirements. Once the participant finds a nonce that produces a suitable hash, it can link its block to the chain.

However, through a process called forking, a blockchain can split into two or more chains. This happens when two or more blocks are linked to the same block. A blockchain network's consensus protocol allows participants to decide which chain is the 'correct' chain so that they can continue to add blocks to that chain. This mitigates the need for a central authority to decide which chain is the correct chain. A violation of consensus will halt the progress of the blockchain as participants cannot decide where to add a block.

The PoW consensus protocol works on the rule that the longest chain is the correct chain. It does this by assuming that the majority of the network's computing power is honest, meaning that it should be able to solve block hashes quicker than dishonest computing power and keep the correct chain the longest chain. If there are two or more longest chains of the same length, participants will randomly select a chain to work on until one of the chains has a block added and prevails as the longest chain.

Garay et al. identifies the two fundamental properties of PoW consensus as the *common prefix* and *chain quality* properties that ensure consensus if they hold [12]. The common prefix property states that honest network participants agree on the longest chain up to k blocks where k is a variable set by the network. The chain quality property states that the number of blocks produced from dishonest participants will not be very large. As part of our model, we make the assumption of majority honesty, which mitigates the need for the chain quality property. Because of this, the objective of our verification is to prove the common prefix property which subsequently proves consensus.

3 Model

3.1 Binary Tree

As mentioned previously, we will model the blockchain as the longest branch in a binary tree. The rationale behind this is that the tree structure offers an effective way to model the stages a blockchain has as it progresses through time. In a binary tree, each node represents a block of the blockchain, with the root node acting as the genesis block as all branches stem from the root node the same way all blockchains stem from the genesis block. Additionally, the different branches of the binary tree model represent the different competing chains at different stages of the network. It is possible to contain data of an arbitrary type within the tree nodes in Isabelle but it is not necessary for our verification as we are only considering the general construction of a blockchain so there is no need to consider the transactions inside each block.

For each node in a binary tree, there can be zero, one or two child nodes. If there are no child nodes, then we are at the end of a branch and it is either the longest chain or was competing to be the longest chain at an earlier point in time, known as a stale block. If there is a single child node, then this means a block was added to the chain at this point. If

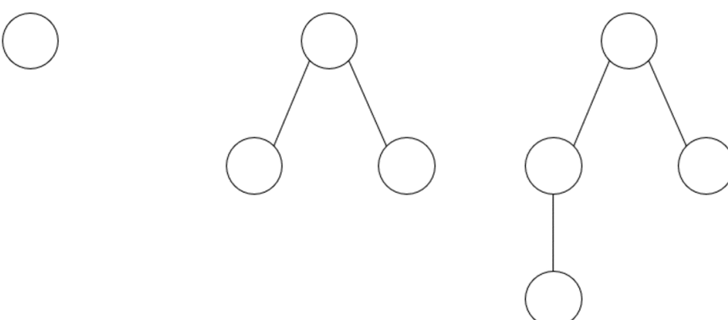


Figure 1 Binary Tree Progression.

```
1  datatype 'a tree =
2      Tip | Node "'a tree" 'a "'a tree"
3
4  definition exampleTree :: "nat tree" where
5    "exampleTree = Node
6                      (Node
7                        (Node
8                          (Node Tip 5 Tip)
9                          2
10                         Tip)
11                       1
12                       (Node Tip 3 Tip))
13                     0
14                     (Node Tip 4 Tip)"
```

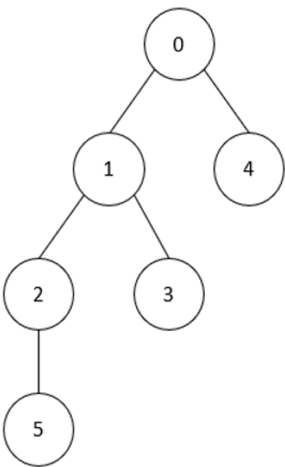


Figure 2 Binary Tree in Isabelle.

there are two child nodes, this is when two blocks are solved simultaneously, so there are now competing chains. Figure 1 showcases each case of binary tree progression. We start with the root node (genesis block), then two nodes (blocks) are added to the tree simultaneously. We now have two valid chains. Eventually, a block is added to the left chain, so we have a single longest chain. Figure 2 shows how we construct a binary tree in Isabelle/HOL with a visual representation of the example tree.

As part of our model, we have made the assumption that all participants in the network are synchronised, meaning they all share the same view of the blockchain. In reality, there is no global view of the blockchain, and each participant has their own copy of the blockchain which may not always be the same for every participant. In our model we only consider a single binary tree for each network so we must assume that participants views are synchronised.

An alternative method for modelling a PoW blockchain network is using a list. Here each entry acts as a block in the longest chain, with the ith entry being the ith block in the chain. However, a list would only keep track of the longest chain and so would not give us a full picture of the network. As a result, we cannot consider a scenario in which a chain that is not the longest chain eventually becoming the longest chain as we would not be keeping a list for it. In reality, this situation would arise from a 51% attack on the network.

3.2 Isabelle Functions

In this section, we will detail the key functions and datatypes that are used to verify the consensus property. We will explain what each function does, how it is encoded into Isabelle, give an example, and showcase any relevant proofs that were later used in the verification. For the functions that require a binary tree as an input, we will use the example binary tree from Figure 2 to show the effect the function has.

3.2.1 Nodes

The *nodes* function counts the number of nodes in the inputted binary tree. In Listing 1 we can see its usage to count six nodes in the example tree. There were no relevant proofs associated with this function.

Listing 1 nodes Function in Isabelle.

```
primrec nodes :: "'a tree ⇒ nat" where
  "nodes Tip = 0"
| "nodes (Node l e r) = Suc (nodes l + nodes r)"

value "nodes exampleTree"
"6" :: "nat"
```

3.2.2 Height

The *height* function calculates the height of the inputted binary tree. In Listing 2 we can see its definition in Isabelle and its usage to calculate that the example tree has a height of four. The height function satisfies some monotonicity properties which can be found in the appendix A.

Listing 2 height Function in Isabelle.

```
primrec height :: "'a tree ⇒ nat" where
  "height Tip = 0"
| "height (Node l e r) = Suc (max (height l) (height r))"

value "height exampleTree"
"4" :: "nat"
```

3.2.3 Longest

The *longest* function returns a set of lists of node data where each list represents a longest chain. In Listing 3 we can see its definition as well as its usage to calculate the longest chain of the example tree. In the example, we have used natural numbers as the node data. There were no relevant proofs associated with this function.

Listing 3 longest Function in Isabelle.

```
primrec longest:: "'a tree ⇒ ('a list) set" where
  "longest Tip = {[]}"
| "longest (Node l e r) =
  { e # p | p. p ∈
```

```
5    (if height l > height r then longest l
6      else if height r > height l then longest r
7      else longest l ∪ longest r)}"
8
9  value "longest exampleTree"
10 "{[0, 1, 2, 5]}" :: "nat list set"
```

3.2.4 Check

The *check* function checks for input tree t that no other branch is within d nodes of one of the longest chains, up to depth n of the tree, returning true if it holds. We call d our difference value. In Listing 4 we can see its definition and its usage to show that the example tree does hold up to depth 1 with a difference value of 1 but does not hold at a depth of 2.

Listing 4 check Function in Isabelle.

```
1  fun check :: "nat ⇒ nat ⇒ 'a tree ⇒ bool" where
2    "check 0 d t = True"
3  | "check (Suc n) d Tip = False"
4  | "check (Suc n) d (Node l e r) =
5      (( (height l - height r > d) ∧ (check n d l) ) ∨
6      ( (height r - height l > d) ∧ (check n d r) ))"
7
8  value "check 1 1 exampleTree"
9  "True" :: "bool"
10
11 value "check 2 1 exampleTree"
12 "False" :: "bool"
```

Furthermore, the first two properties we prove are the weakened statements of the depth and difference values, stating that if the check function is true for a given depth or difference value x then it will also be true for all values less than x. These statements are proven by structural induction over the tree parameter t as well as the depth parameter n. The common prefix lemma states that given a tree t that returns true on the check function for depth n and difference value d, all longest chains will have the same first n nodes. Listing 5 shows these properties in Isabelle, with the proof of the common prefix property in appendix A.

Listing 5 check Function Properties in Isabelle.

```
1  lemma check_weaken_distance:
2    assumes "check n (Suc x) t"
3      shows "check n x t"
4  using assms by (induction rule: check.induct, auto)
5
6  proposition check_weaken_depth:
7    assumes "check (Suc x) d t"
8      shows "check x d t"
9  using assms by (induction rule: check.induct, auto)
10
11 lemma common_prefix[rule_format]:
12   "∀p p'. check n d t ∧ p ∈ longest t
13   ∧ p' ∈ longest t ⟶ take n p = take n p'"
```

The common prefix lemma shows that the common prefix property holds for an arbitrary tree t that passes the check function. To prove consensus, we must now show that this property holds for the set of trees that can occur in our majority honest PoW network.

3.2.5 Event

We model our blockchain network as a sequence of events, where an event is the pair consisting of the *Honest* and *State* variable and describes an action carried out on a binary tree. The Honest field is of type Boolean and it is used to distinguish between honest and dishonest mining events, whilst the State field contains the state of the binary tree after the event has occurred. We then define a function *count* which can be used to count the number of honest or dishonest events in a list of events. We prove properties for adding events to a list of events in appendix A in Listing 16. The definitions can be seen in Listing 6.

Listing 6 Event Properties in Isabelle.

```
1  record 'a event =
2    Honest :: bool
3    State :: "'a tree"
4
5  definition count::"bool ⇒ ('a event) list ⇒ nat" where
6    "count b = List.length ∘ filter (λx. Honest x = b)"
```

3.3 Mining

The mining locale is used to describe how we mine blocks and add them to the blockchain using add functions. It has two key properties:

1. Mining on top of an empty tree results in a tree with one node.
2. Mining on top of a non-empty tree adds the new block either to the left or the right branch.

From this, we prove the lemmas `mining_cases` and `height_add`. `mining_cases` is used to describe the cases of mining on either the left or right branch of a tree and follows easily from property 2 of the locale. `height_add` shows that the height of a tree either stays the same or increases by 1 after mining on it and is proved by induction on the binary tree variable. Listing 7 shows these properties in Isabelle, with the proof of `height_add` in appendix B.

Listing 7 Mining Locale in Isabelle.

```
1  locale mining =
2      fixes add :: "'a tree ⇒ 'a tree"
3      assumes m1: "∃e. add Tip = Node Tip e Tip"
4          and m2: "⋀l e r. add (Node l e r) = Node (add l) e r
5                    ∨ add (Node l e r) = Node l e (add r)"
6
7    lemma mining_cases:
8      fixes l e r
9      obtains (l) "add (Node l e r) = Node (add l) e r"
10           | (r) "add (Node l e r) = Node l e (add r)"
11     using m2 by auto
```

```
12
13   lemma height_add:"height (add t) = height t
14                    ∨ height (add t) = Suc (height t)"
```

From these lemmas, we derive the lemma `check_add` which shows the three possible cases a tree can be after applying an add function to it, assuming the original tree passes the check function with difference value $d+1$ and depth n. The three cases are as follows:

1. We have mined on one of the longest chains. Here, the height of the new tree is higher than the original tree and can pass the check function with difference value $d+2$ and depth n.
2. We have mined on one of the second-longest chains. Here, the height of the new tree is equal to the original tree and can pass the check function with difference value d and depth n.
3. We have mined on one of the chains that is not one of the longest or second longest chains. Here, the height of the new tree is equal to the original tree and can pass the check function with difference value $d+1$ and depth n.

We prove this statement by structural induction over the tree variable. It can be seen in Listing 8 with its proof in appendix B.

Listing 8 *check_add* Lemma in Isabelle.

```
1  lemma check_add[rule_format]:
2  "check n (Suc d) t ⟶
3    height t < height (add t) ∧ check n (Suc (Suc d)) (add t)
4    ∨ height t = height (add t) ∧ check n (Suc d) (add t)
5    ∨ height t = height (add t) ∧ check n d (add t)"
```

Lastly, we create the corollary `check_add_cases` to distinguish between the cases of mining on a tree. This follows directly from `check_add` and can be seen in Listing 9.

Listing 9 *check_add_cases* Lemma in Isabelle.

```
1  corollary check_add_cases:
2      assumes "check n (Suc d) t"
3      obtains "check n (Suc (Suc d)) (add t)"
4            | "check n (Suc d) (add t)"
5            | "check n d (add t)"
6      using check_add[OF assms] by auto
```

3.4 Honest Mining

The honest locale is used to describe how honest participants mine blocks and add them to the blockchain. It has the same properties as the mining locale with the additional property that a mined block is always added to the longest chain of the network. Like in the mining locale, we prove similar `mining_cases`, `height_add` and `check_add` lemmas.

The cases for honest minings are adding to the left or right branch of a tree depending on which branch is longer. This follows from the new locale property. `height_add` shows that the height of the tree always increases by 1 when the mining is honest and is proved by induction on the binary tree variable. From these lemmas, we derive the `check_add` lemma which shows the only possible cases a tree can be after applying an honest add function to it,

assuming the original tree passes the check function with difference value d and depth n, is that the new tree passes the check function with difference value $d + 1$ and depth n. This is the same as case 1 for the `check_add` lemma in the mining locale as we have mined on the longest chain. Listing 10 shows the locale in Isabelle, with the proofs of `check_add` and `height_add` found in Appendix C.

Listing 10 Honest Mining Locale in Isabelle.

```
1  locale honest = mining +
2    assumes h1:
3    "⋀l e r. height l ≥ height r ∧ add (Node l e r) = Node (add l) e r
4          ∨ height r ≥ height l ∧ add (Node l e r) = Node l e (add r)"
5
6  lemma mining_cases:
7    fixes l e r
8    obtains (l) "height l ≥ height r ∧ add (Node l e r) = Node (add l) e r"
9          | (r) "height r ≥ height l ∧ add (Node l e r) = Node l e (add r)"
10   using h1 by auto
11
12 lemma height_add: "height (add t) = Suc (height t)"
13
14 lemma check_add[rule_format]:
15   "check n d t ⟶ check n (Suc d) (add t)"
```

4 Verification

For the verification, we fix the state of the blockchain at a particular point in time, then define all possible future progressions of the tree structure and then show that all of them preserve the common prefix property.

To this end, we introduce a new locale `blockchain` which uses the mining locale for dishonest miners and the locale for honest miners. In addition, we introduce two new parameters to provide context to the model:

- A fixed tree $t0$ which represents the blockchain at a particular point in time.
- A natural number *depth*, which determines the length of the prefix which is considered stable, i.e., which should not change in the future.

In addition to these parameters, we assume two more properties:

1. The initial tree, $t0$ passes the check function for a threshold equal to the difference between its height and the depth value (property `b1` in Listing 11).
2. The height of the initial chain is greater than the depth (property `b2` in Listing 11).

We then define the set `traces`, which contains all possible event sequences which can occur in a network with the majority of miners being honest. The set is defined inductively and each sequence can be of one of the following cases:

1. The list with one event that is honest and applies an honest add function to $t0$ (Line 9 in Listing 11).
2. The list with one event that is dishonest and applies a dishonest add function to $t0$ (Line 10 in Listing 11).
3. A list of events that is already in traces, that is prepended with an honest event (Line 11 in Listing 11).

4. A list of events that is already in traces, with less dishonest events than the sum of the number of honest events and the threshold value, prepended with a dishonest event (Line 13 in Listing 11).

Listing 11 shows the blockchain locale and traces set in Isabelle.

Listing 11 Blockchain Locale in Isabelle.

```
locale blockchain =
  honest hadd + mining dadd
  for hadd::"'a tree ⇒ 'a tree" and dadd::"'a tree ⇒ 'a tree" +
  fixes depth::nat and t0::"'a tree"
  assumes b1: "check depth (Suc (height t0 - depth)) t0"
      and b2: "height t0 > depth"

inductive_set traces :: "('a event list) set" where
  honest_base: "[(|Honest = True, State = hadd t0|)] ∈ traces"
| dishonest_base: "[(|Honest = False, State = dadd t0|)] ∈ traces"
| honest_induct: "⟦t ∈ traces⟧
  ⟹ (|Honest = True, State = hadd (State (hd t))|) # t ∈ traces"
| dishonest_induct:
  "⟦t ∈ traces; count False t < count True t + (height t0 - depth)⟧
    ⟹ (|Honest = False, State = dadd (State (hd t))|) # t ∈ traces"
```

Using traces, we create the lemmas `bounded_dishonest_mining` and `bounded_check`. `bounded_dishonest_mining` shows that given a list of events from traces, the sum of the number of honest events and the threshold value is greater than or equal to the number of dishonest events. This follows from assumption 2 of the blockchain locale. `bounded_check` states that given a list of events from traces then the most recent tree from that list passes the check function with a depth of depth and a difference value equal to the sum of number of honest events and the threshold value minus the number of dishonest events plus one. This statement is proven by induction of the tree variable for each case in traces. Listing 12 shows these lemmas in Isabelle with their proofs found in appendix D.

Listing 12 Blockchain Locale in Isabelle.

```
lemma bounded_dishonest_mining:
  fixes t assumes "t ∈ traces"
  shows "count True t + (height t0 - depth) ≥ count False t"

lemma bounded_check:
  fixes t assumes "t ∈ traces"
  shows "check depth
        (Suc (count True t + (height t0 - depth) - count False t))
        (State (hd t))"
```

We prove the consensus theorem within the blockchain locale. It states that given a list of events from traces and lists of the longest chains nodes, then the lists of these chains are the same up to index depth. This can be seen in Listing 13.

Listing 13 Consensus Theorem in Isabelle.

```
theorem consensus:
  fixes t assumes "t ∈ traces"
  and "p ∈ longest (State (hd t))"
  and "p' ∈ longest (State (hd t))"
shows "take depth p = take depth p'"
using assms(2,3)
common_prefix[of depth
  "Suc (count True t + (height t0 - depth) - count False t)"
  "(State (hd t))"]
bounded_check[OF assms(1)] by blast
```

The theorem is similar to the `common_prefix` lemma we proved earlier in Listing 5 with the key differences being that tree t is in the traces set and we use the *depth* value for parameter n in the check function. The `common_prefix` lemma was defined before we established any of our locales and so is just a lemma for the check function itself and does not consider anything related to consensus. The consensus theorem applies this lemma to the context of consensus and is essentially a proof of the common prefix property. As mentioned previously, we do not require the chain quality property under our assumption of majority honesty so the common prefix property is enough to show that consensus holds.

5 Discussion

Our verification of consensus in this model required us to make simplifications to how a blockchain network works. As mentioned in Section 3, we assume majority honesty and synchronisation of the network. Despite PoW assuming majority honesty, this is not always the case and so consensus can break down in the presence of a 51% attack. As for synchronisation, a blockchain network can become asynchronous because of dishonest behaviour or a fault within the network such as a participant having connectivity issues and not being able to update their private copy of the blockchain. As a result of this, consensus can still break down in a majority honest network.

Outside of these assumptions, we also made design choices for the model that lead to simplifications. The first of these is the use of a binary tree instead of an n-ary tree. With a binary tree, we can model forking at a node that can result in two chains. However, it is possible for there to be more than one fork at a node, resulting in three or more chains branching from a single node. A binary tree cannot be used to consider such an event, but in reality the likelihood of this event occurring is very unlikely as it would require three or more block hashes to be solved simultaneously and added to the chain at the same time. Using an n-ary tree in our verification would also make the process far more complicated as we would have to consider an arbitrary amount of branches for each node. It is for these reasons that we did not believe using a n-ary tree instead of binary tree was worth the effort required.

The main simplification of our model is that it is not probabilistic. Blocks are added to the blockchain by solving its hash which introduces a probabilistic element into the network as we cannot be absolutely sure how long such a brute force exercise will take. Because of this, majority honesty may not always hold as dishonest participants could get lucky in their hash searching whilst honest participants could get unlucky. Mining also plays a key role in the common prefix and chain quality properties described in [12], with their actual definitions being probabilistic in nature as a result. Specifically, these properties only have to hold with high probability but we show they always hold in our deterministic model.

6 Related Work

There is some work formalising traditional consensus protocols and some has even been mechanized in theorem provers. For example there exists a formalisation of Paxos [18] and Disk Paxos [15] in Isabelle, HotStuff [6] in Agda [3], and Velisarios [32] in Coq [34]. When it comes to blockchain, however, we usually do have less control over the actual network participants. Thus, verification of consensus in blockchain poses additional challenges compared to the verification of traditional BFT protocols.

There has been some early work on the verification of consensus in blockchain. In particular, blockchain in general [30, 21], the Bitcoin backbone protocol [12], general proof-of-stake [17], or a custom proof-of-work protocol known as Snow White [10]. While all these studies provide useful insides into blockchain consensus they are not mechanized and thus may contain mistakes.

More recently there has been some work on mechanizing blockchain consensus. For example there is a formalisation of Tendermint [4] using TLA+ and a formalisation of the Ethereum Beacon Chain [7] in Dafny. Moreover, there are formalisations of Casper [14], CBC Casper [27], Stellar [19], and general inter-blockchain Protocols [16] in Isabelle. Finally, there exists formalisations of Casper [29] (based on [14]), CKB [20, 5], Algorand [1], and Gasper [2] in Coq [34]. While all these works formalise various types of consensus protocols and some even verify certain properties for these protocols, none of them verify consistency in terms of common prefix which is the focus of our work.

There are three exceptions to this. First, there is the work of Pîrlea and Sergey [31] in which they formalise a general blockchain protocol in Coq. Similarly to our work, they model a blockchain as a tree (although implicitly as part of a forest). They then verify eventual ledger consistency, which is a property similar to the one we verify. The main difference to our work, however, is that they do not consider dishonest nodes in their analysis. Thus, by allowing for nodes with arbitrary behaviour, we complement their work.

In addition, Marmsoler formalised a type of general proof of work [22] in FACTum [23]. Similar to our work, the author considers honest as well as dishonest nodes in his model. Different to this work, however, a blockchain is modeled in terms of a list which does not allow to investigate forks. Thus, by modeling a blockchain as a tree instead of a list, we complement his work.

Finally, Thomsen and Spitters formalize a general, Nakamoto-Style Proof of Stake protocol in Coq [35]. They then verify a safety property similar to the common prefix property discussed in this paper. The main difference to our work lies in the type of considered consensus mechanism. While Thomsen and Spitters work is based on a general Proof of Stake consensus mechanism, our work focuses more on Proof of Work consensus.

7 Conclusion

In this paper, we have formally verified that consensus holds in a blockchain network that uses PoW and is assumed to be majority honest. This verification was carried out in Isabelle/HOL by modelling the blockchain as the longest chain in a binary tree. We have described the key functions and lemmas required to carry out this verification and outlined our assumptions of honesty and synchronisation for consensus to hold in our model. Lastly, we identify the key limitation of our model in that it is not probabilistic to account for probabilistic elements that occur within a PoW network such as mining. The successful verification of this paper serves as motivation for the implementation of the verified consensus protocol.

This paper could be expanded on in further research by aiming to address this limitation and developing a probabilistic verification of PoW using a different mathematical model such as state machines. Alternatively, a similar verification could be carried out on a different consensus protocol such as Proof of Stake (PoS) which is used by the Ethereum blockchain. However, PoS also has probabilistic elements that would need to be accounted for such as validator selection that would need to be considered or mitigated under a simplifying assumption.

References

1. Musab A Alturki, Jing Chen, Victor Luchangco, Brandon Moore, Karl Palmskog, Lucas Peña, and Grigore Roşu. Towards a verified model of the algorand consensus protocol in coq. In *Formal Methods. FM 2019 International Workshops: Porto, Portugal, October 7–11, 2019, Revised Selected Papers, Part I 3*, pages 362–367. Springer, 2020.
2. Musab A Alturki, Elaine Li, Daejun Park, Brandon Moore, Karl Palmskog, Lucas Pena, and Grigore Roşu. Verifying gasper with dynamic validator sets in coq. *Technical report*, 2020.
3. Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda–a functional language with dependent types. In *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22*, pages 73–78. Springer, 2009.
4. Sean Braithwaite, Ethan Buchman, Igor V. Konnov, Zarko Milosevic, Ilina Stoilkovska, Josef Widder, and Anca Zamfir. Formal specification and model checking of the tendermint blockchain synchronization protocol (short paper). In *FMBC@CAV*, 2020. URL: `https://api.semanticscholar.org/CorpusID:228097346`.
5. Hao Bu and Meng Sun. Towards modeling and verification of the ckb block synchronization protocol in coq. In *Formal Methods and Software Engineering: 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1–3, 2021, Proceedings 22*, pages 287–296. Springer, 2020.
6. 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*, pages 616–635, Cham, 2022. Springer International Publishing.
7. Franck Cassez, Joanne Fuller, and Aditya Asgaonkar. Formal verification of the ethereum 2.0 beacon chain. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 167–182. Springer, 2022.
8. Coindesk. How the dao hack changed ethereum and crypto, 2023. [Accessed December 2023]. URL: `https://www.coindesk.com/consensus-magazine/2023/05/09/coindesk-turns-10-how-the-dao-hack-changed-ethereum-and-crypto/`.
9. CoinMarketCap. Bitcoin market capitalization, 2023. [Accessed December 2023]. URL: `https://coinmarketcap.com/currencies/bitcoin/`.
10. Phil Daian, Rafael Pass, and Elaine Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In Ian Goldberg and Tyler Moore, editors, *Financial Cryptography and Data Security*, pages 23–41, Cham, 2019. Springer International Publishing.
11. Forkast. How ethereum classic's 51ethereum, 2020. [Accessed December 2023]. URL: `https://forkast.news/video-audio/ethereum-classic-repeat-hacks-etc-labs-ceo-terry-culver-ben-sauter/`.
12. Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 281–310. Springer, 2015.
13. Growjo. Certik revenue and competitors, 2022. [Accessed December 2023]. URL: `https://growjo.com/company/CertiK`.

14 Yoichi Hirai. A repository for pos related formal methods. https://github.com/palmskog/pos, 2018.

15 Mauro Jaskelioff and Stephan Merz. Proving the correctness of disk paxos. *Archive of Formal Proofs*, June 2005. , Formal proof development. URL: `https://isa-afp.org/entries/DiskPaxos.html`.

16 Florian Kammüller and Uwe Nestmann. Inter-Blockchain Protocols with the Isabelle Infrastructure Framework. In Bruno Bernardo and Diego Marmsoler, editors, *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*, volume 84 of *Open Access Series in Informatics (OASIcs)*, pages 11:1–11:12, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/OASIcs.FMBC.2020.11`.

17 Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual international cryptology conference*, pages 357–388. Springer, 2017.

18 Philipp Küfner, Uwe Nestmann, and Christina Rickmann. Formal verification of distributed algorithms. In Jos C. M. Baeten, Tom Ball, and Frank S. de Boer, editors, *Theoretical Computer Science*, pages 209–224, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

19 Giuliano Losa and Mike Dodds. On the Formal Verification of the Stellar Consensus Protocol. In Bruno Bernardo and Diego Marmsoler, editors, *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*, volume 84 of *Open Access Series in Informatics (OASIcs)*, pages 9:1–9:9, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/OASIcs.FMBC.2020.9`.

20 Xiaokun Luan and Meng Sun. Modeling and verification of ckb consensus protocol in coq. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 660–667. IEEE, 2021.

21 Bojan Marinković, Paola Glavan, Zoran Ognjanović, Dragan Doder, and Thomas Studer. Probabilistic consensus of the blockchain protocol. In Gabriele Kern-Isberner and Zoran Ognjanović, editors, *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, pages 469–480, Cham, 2019. Springer International Publishing.

22 Diego Marmsoler. Towards verified blockchain architectures: A case study on interactive architecture verification. In *Formal Techniques for Distributed Objects, Components, and Systems: 39th IFIP WG 6.1 International Conference, FORTE 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17–21, 2019, Proceedings*, pages 204–223, Berlin, Heidelberg, 2019. Springer-Verlag. `doi:10.1007/978-3-030-21759-4_12`.

23 Diego Marmsoler and Habtom Kashay Gidey. Interactive verification of architectural design patterns in factum. *Form. Asp. Comput.*, 31(5):541–610, November 2019. `doi:10.1007/s00165-019-00488-x`.

24 Ahmed Afif Monrat, Olov Schelén, and Karl Andersson. A survey of blockchain from the perspectives of applications, challenges, and opportunities. *IEEE Access*, 7:117134–117151, 2019. `doi:10.1109/ACCESS.2019.2936094`.

25 Neptune Mutual. Ethereum classic 51 [Accessed December 2023]. URL: `https://neptunemutual.com/blog/ethereum-classic-51-attacks/`.

26 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, 2008.

27 R. Nakamura, T. Jimba, and D. Harz. Refinement and verification of cbc casper. In *2019 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 26–38, Los Alamitos, CA, USA, June 2019. IEEE Computer Society. `doi:10.1109/CVCBT.2019.00008`.

28 Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

29 Karl Palmskog, Milos Gligoric, Lucas Pena, Brandon Moore, and Grigore Roşu. Verification of casper in the coq proof assistant. Technical report, University of Illinois at Urbana-Champaign, 2018.

30 Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, pages 643–673, Cham, 2017. Springer International Publishing.

31 George Pîrlea and Ilya Sergey. Mechanising blockchain consensus. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 78–90, 2018.

32 Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Esteves-Verissimo. Velisarios: Byzantine fault-tolerant protocols powered by coq. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 619–650, Cham, 2018. Springer International Publishing.

33 Statista. Blockchain - statistics and facts, 2023. [Accessed December 2023]. URL: `https://www.statista.com/statistics/800426/worldwide-blockchain-solutions-spending/`.

34 The Coq Development Team. The Coq reference manual – release 8.18.0. `https://coq.inria.fr/doc/V8.18.0/refman`, 2023.

35 Søren Eller Thomsen and Bas Spitters. Formalizing nakamoto-style proof of stake. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–15. IEEE, 2021.

A Functions

Listing 14 Monotonic Properties of height Function in Isabelle.

```
1  proposition height_mono_l:
2  "height r ≤ height l ⟹ height l < height l' ⟹
3   height (Node l e r) < height (Node l' e r)"
4  by (induction l; simp)
5
6  proposition height_mono_r:
7  "height l ≤ height r ⟹ height r < height r' ⟹
8   height (Node l e r) < height (Node l e r')"
9  by (induction r; simp)
```

Listing 15 Proof of the common prefix lemma in Isabelle.

```
1  lemma common_prefix[rule_format]:
2    "∀p p'. check n d t ∧ p∈longest t ∧ p'∈longest t ⟶
3    take n p = take n p'"
4  proof (induction rule: check.induct)
5    case (1 d t)
6    then show ?case by simp
7  next
8    case (2 n d)
9    then show ?case by simp
10 next
11   case (3 n d l e r)
12   show ?case
13   proof (rule+, (erule conjE)+)
14     fix p p'
15     assume a1: "check (Suc n) d (Node l e r)"
16        and a2: "p ∈ longest (Node l e r)"
17        and a3: "p' ∈ longest (Node l e r)"
18     from a1 consider (1) "d < height l - height r ∧ check n d l"
```

```
19      | (2) "d < height r - height l ∧ check n d r" by auto
20      then show "take (Suc n) p = take (Suc n) p’"
21      proof cases
22        case 1
23        then have "height r < height l" by auto
24        then have "tl p ∈ longest l" and "tl p’ ∈ longest l"
25        using a2 a3 by auto
26        then have "take n (tl p) = take n (tl p’)" using 1 3 by blast
27        moreover have "take (Suc n) p = hd p # take n (tl p)"
28        using a2 by auto
29        moreover have "take (Suc n) p’ = hd p’ # take n (tl p’)"
30        using a3 by auto
31        moreover have "hd p = hd p’" using a2 a3 by auto
32        ultimately show ?thesis by simp
33      next
34        case 2 (*symmetric*)
35      qed
36    qed
37  qed
```

Listing 16 Proofs for Adding Events to Lists of Events in both Honest and Dishonest Cases.

```
1   lemma count_true_base[simp]:
2     "count True [] = 0" unfolding count_def by simp
3   lemma count_false_base[simp]:
4     "count False [] = 0" unfolding count_def by simp
5
6   lemma count_honest_true_ind[simp]:
7     assumes "Honest e"
8       shows "count True (e#es) = Suc (count True es)"
9       unfolding count_def using assms by simp
10  lemma count_honest_false_ind[simp]:
11    assumes "Honest e"
12      shows "count False (e#es) = count False es"
13      unfolding count_def using assms by simp
14
15  lemma count_dhonest_false_ind[simp]:
16    assumes "¬ Honest e"
17    shows "count False (e#es) = Suc (count False es)"
18    unfolding count_def using assms by simp
19  lemma count_dhonest_true_ind[simp]:
20    assumes "¬ Honest e"
21    shows "count True (e#es) = count True es"
22    unfolding count_def using assms by simp
```

B Mining Locale

Listing 17 Mining *height_add* Proof in Isabelle.

```
lemma height_add:
"height (add t) = height t ∨ height (add t) = Suc (height t)"
  proof (induction t)
    case Tip
    then show ?case using m1 by auto
  next
    case (Node l e r)
    show ?case
    proof (cases rule: mining_cases)
      case l
      moreover from this have
      "height (add (Node l e r)) = height (Node (add l) e r)" by simp
      ultimately show ?thesis using Node(1) by auto
    next
      case r
      moreover from this have
      "height (add (Node l e r)) = height (Node l e (add r))" by simp
      ultimately show ?thesis using Node(2) by auto
    qed
  qed
```

Listing 18 Mining *check_add* Proof in Isabelle.

```
lemma check_add[rule_format]:
"check n (Suc d) t ⟶
height t < height (add t) ∧ check n (Suc (Suc d)) (add t)
∨ height t = height (add t) ∧ check n (Suc d) (add t)
∨ height t = height (add t) ∧ check n d (add t)"
  proof (induction rule: check.induct)
    case (1 d t)
    then show ?case using height_add by auto
  next
    case (2 n d)
    then show ?case by simp
  next
    case (3 n d l e r)
    show ?case
    proof
      assume "check (Suc n) (Suc d) (Node l e r)"
      then consider (l) "Suc d < height l - height r ∧ check n (Suc d) l"
      | (r) "Suc d < height r - height l ∧ check n (Suc d) r" by auto
      then show "height (Node l e r) < height (add (Node l e r))
      ∧ check (Suc n) (Suc (Suc d)) (add (Node l e r))
      ∨ height (Node l e r) = height (add (Node l e r))
      ∧ check (Suc n) (Suc d) (add (Node l e r))
```

```
23          ∨ height (Node l e r) = height (add (Node l e r))
24          ∧ check (Suc n) d (add (Node l e r))"
25          proof (cases)
26            case 1
27            then consider
28            "height l < height (add l) ∧ check n (Suc (Suc d)) (add l)"
29            | "height l = height (add l) ∧ check n (Suc d) (add l)"
30            | "height l = height (add l) ∧ check n d (add l)" using 3 by auto
31            then show ?thesis
32            proof cases
33              case 1
34              then show ?thesis
35              proof (cases rule: mining_cases)
36                case l2: l
37                moreover from 1 l have
38                "check (Suc n) (Suc (Suc d)) (Node (add l) e r)" by auto
39                moreover from 1 l l2 have
40                "height (Node l e r) < height (add (Node l e r))" by auto
41                ultimately show ?thesis by simp
42              next
43                case r
44                consider "height (add r) = height r"
45                | "height (add r) = Suc (height r)" using height_add by auto
46                then show ?thesis
47                proof cases
48                  case l1: 1
49                  then have "height (Node l e r) = height (add (Node l e r))"
50                  using r by simp
51                  moreover have "check (Suc n) (Suc d) (add (Node l e r))"
52                  using l l1 r by auto
53                  ultimately show ?thesis by simp
54                next
55                  case x: 2
56                  moreover have "height l > Suc (height r)" using l by auto
57                  ultimately have
58                  "height (Node l e r) = height (add (Node l e r))"
59                  using r by simp
60                  moreover have "check (Suc n) d (add (Node l e r))"
61                  proof -
62                    have "d < height l - height (add r)" using x l by auto
63                    moreover have "check n d l"
64                    using l check_weaken_distance by simp
65                    ultimately show ?thesis using r by simp
66                  qed
67                  ultimately show ?thesis by simp
68                qed
69              qed
70            next
```

```
71          case 2
72          then show ?thesis
73          proof (cases rule: mining_cases)
74            case l2: l
75            moreover have "check (Suc n) (Suc d) (Node (add l) e r)"
76            using 2 l by auto
77            moreover have
78            "height (Node l e r) = height (add (Node l e r))"
79            using 2 l2 by auto
80            ultimately show ?thesis by simp
81          next
82            case r
83            consider "height (add r) = height r"
84            | "height (add r) = Suc (height r)" using height_add by auto
85            then show ?thesis
86            proof cases
87              case l1: 1
88              then have "height (Node l e r) = height (add (Node l e r))"
89              using r by simp
90              moreover have "check (Suc n) (Suc d) (add (Node l e r))"
91              using l l1 r by auto
92              ultimately show ?thesis by simp
93            next
94              case x: 2 (*symmetric to 1*)
95            qed
96          qed
97        next
98          case 3 (*symmetric to 2*)
99        qed
100     next
101       case r (*symmetric to l*)
102     qed
103   qed
104 qed
```

C Honest Mining Locale

Listing 19 Honest Mining *height_add* Proof in Isabelle.

```
1 lemma height_add: "height (add t) = Suc (height t)"
2 proof (induction t)
3   case Tip
4   then show ?case using m1 by auto
5 next
6   case (Node l e r)
7   show ?case
8   proof (cases rule: mining_cases)
9     case l
```

```
10      moreover from this have
11      "height (add (Node l e r)) = height (Node (add l) e r)" by simp
12      ultimately show ?thesis using Node(1) by simp
13    next
14      case r
15      moreover from this have
16      "height (add (Node l e r)) = height (Node l e (add r))" by simp
17      ultimately show ?thesis using Node(2) by simp
18    qed
19  qed
```

Listing 20 Honest Mining *check_add* Proof in Isabelle.

```
1   lemma height_add: "height (add t) = Suc (height t)"
2   proof (induction t)
3     case Tip
4     then show ?case using m1 by auto
5   next
6     case (Node l e r)
7     show ?case
8     proof (cases rule: mining_cases)
9       case l
10      moreover from this have
11      "height (add (Node l e r)) = height (Node (add l) e r)" by simp
12      ultimately show ?thesis using Node(1) by simp
13    next
14      case r
15      moreover from this have
16      "height (add (Node l e r)) = height (Node l e (add r))" by simp
17      ultimately show ?thesis using Node(2) by simp
18    qed
19  qed
```

D Blockchain Locale

Listing 21 *bounded_dishonest_mining* Proof in Isabelle.

```
1   lemma bounded_dishonest_mining:
2     fixes t
3     assumes "t ∈ traces"
4     shows "count True t + (height t0 - depth) ≥ count False t"
5     using assms b2 by (induction rule:traces.induct; simp)
```

Listing 22 *bounded_check* Proof in Isabelle.

```
1   lemma bounded_check:
2     fixes t
3     assumes "t ∈ traces"
4     shows "check depth
```

```
5    (Suc (count True t + (height t0 - depth) - count False t))
6    (State (hd t))"
7    using assms
8  proof (induction rule:traces.induct)
9    case honest_base
10   define t where "t = [⦇Honest = True, State = hadd t0⦈]"
11   then have "Suc (count True t + (height t0 - depth) - count False t)
12   = Suc (Suc (height t0 - depth))" by simp
13   moreover from b1 have "check depth
14   (Suc (Suc (height t0 - depth))) (hadd t0)"
15   using honest.check_add by (simp add: honest_axioms)
16   ultimately show ?case unfolding t_def by simp
17 next
18   case dishonest_base
19   define t where "t = [⦇Honest = False, State = dadd t0⦈]"
20   then have *: "Suc (count True t + (height t0 - depth) - count False t)
21   = (height t0 - depth)" using b2 by simp
22   show ?case
23   proof (cases rule: check_add_cases[OF b1])
24     case 1
25     then have "check depth (Suc (height t0 - depth)) (dadd t0)"
26     using * unfolding t_def using check_weaken_distance by simp
27     then show ?thesis using * unfolding t_def
28     using check_weaken_distance by simp
29   next
30     case 2
31     then show ?thesis using * unfolding t_def
32     using check_weaken_distance by auto
33   next
34     case 3
35     then show ?thesis using * unfolding t_def by simp
36   qed
37 next
38   case (honest_induct t)
39   define t’ where "t’ = ⦇Honest = True, State = hadd (State (hd t))⦈ # t"
40   moreover have "Suc (count True t + (height t0 - depth)) > count False t"
41   using honest_induct bounded_dishonest_mining[OF honest_induct(1)]
42   by simp
43   ultimately have "count True t’ + (height t0 - depth) - count False t’
44   = Suc (count True t + (height t0 - depth) - count False t)" by simp
45   moreover from honest_induct have "check depth
46   (Suc (Suc (count True t + (height t0 - depth) - count False t)))
47   (hadd (State (hd t)))" using honest.check_add[OF honest_axioms]
48   by (simp add: honest_axioms)
49   ultimately show ?case unfolding t’_def
50   using check_weaken_distance by simp
51 next
52   case (dishonest_induct t)
```

```
53     define t' where "t' = ⦇Honest = False, State = dadd (State (hd t))⦈ # t"
54     then have *: "count True t' + (height t0 - depth) - count False t'
55     = count True t + (height t0 - depth) - Suc (count False t)" by simp
56     have "check depth
57     (Suc (count True t' + (height t0 - depth) - count False t'))
58     (State (hd t'))"
59     proof (cases rule: check_add_cases[OF dishonest_induct(3)])
60       case 1
61       then have "check depth
62       (Suc ((count True t + (height t0 - depth) - (count False t))))
63       (dadd (State (hd t)))" using check_weaken_distance by simp
64       then have "check depth
65       (((count True t + (height t0 - depth) - (count False t))))
66       (dadd (State (hd t)))" using check_weaken_distance by simp
67       moreover have "count True t + (height t0 - depth) > (count False t)"
68       using dishonest_induct(2) by simp
69       ultimately have "check depth
70       (Suc (count True t + (height t0 - depth) - Suc (count False t)))
71       (dadd (State (hd t)))" using Suc_diff_Suc by simp
72       then show ?thesis using * unfolding t'_def by simp
73     next
74       case 2
75       then have "check depth
76       (((count True t + (height t0 - depth) - (count False t))))
77       (dadd (State (hd t)))" using check_weaken_distance by simp
78       moreover have "count True t + (height t0 - depth) > (count False t)"
79       using dishonest_induct(2) by simp
80       ultimately have "check depth
81       (Suc (count True t + (height t0 - depth) - Suc (count False t)))
82       (dadd (State (hd t)))" using Suc_diff_Suc by simp
83       then show ?thesis using * unfolding t'_def by simp
84     next
85       case 3
86       moreover have "count True t + (height t0 - depth) > (count False t)"
87       using dishonest_induct(2) by simp
88       ultimately have "check depth
89       (Suc (count True t + (height t0 - depth) - Suc (count False t)))
90       (dadd (State (hd t)))" using Suc_diff_Suc by simp
91       then show ?thesis using * unfolding t'_def by simp
92     qed
93     then show ?case unfolding t'_def by simp
94   qed
```

Part III.

Smart Contracts

Formalizing Automated Market Makers in the Lean 4 Theorem Prover

Daniele Pusceddu
ETH Zurich, Switzerland
University of Cagliari, Italy

Massimo Bartoletti
University of Cagliari, Italy

Abstract

Automated Market Makers (AMMs) are an integral component of the decentralized finance (DeFi) ecosystem, as they allow users to exchange crypto-assets without the need for trusted authorities or external price oracles. Although these protocols are based on relatively simple mechanisms, e.g. to algorithmically determine the exchange rate between crypto-assets, they give rise to complex economic behaviours. This complexity is witnessed by the proliferation of models that study their structural and economic properties. Currently, most of theoretical results obtained on these models are supported by pen-and-paper proofs. This work proposes a formalization of constant-product AMMs in the Lean 4 Theorem Prover. To demonstrate the utility of our model, we provide mechanized proofs of key economic properties like arbitrage, that at the best of our knowledge have only been proved by pen-and-paper before.

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

Keywords and phrases Smart contracts, Ethereum, Verification, Blockchain

Digital Object Identifier 10.4230/OASIcs.FMBC.2024.5

Supplementary Material *Software*: `https://github.com/danielepusceddu/lean4-amm/tree/paper`, archived at `swh:1:dir:090eb97d11848e5615172481ca6d55537974261c`

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.

1 Introduction

Automated Market Makers (AMMs) are one of the key applications in the Decentralized Finance (DeFi) ecosystem, as they allow users to trade crypto-assets without the need for trusted intermediaries [13]. Unlike traditional order-book exchanges, where buyers and sellers must find a counterpart, AMMs enable traders to autonomously swap assets deposited in liquidity pools contributed by other users, who are incentivized to provide liquidity by a complex reward mechanism. At the time of writing, there are multiple AMM protocols controlling several billions of dollars worth of assets[1]. This has made AMMs an appealing target for attacks, resulting in losses worth billions of dollars over time[2].

The security of AMMs depends on several factors: besides the absence of traditional programming bugs, it is crucial that their economic mechanism gives rise to a rational behaviour of its users that aligns with the AMM ideal functionality, i.e. providing an algorithmic exchange rate coherent with the one given by trusted price oracles. Therefore, it

[1] `https://defillama.com/protocols/Dexes`
[2] `https://chainsec.io/defi-hacks/`

© Daniele Pusceddu and Massimo Bartoletti;
licensed under Creative Commons License CC-BY 4.0
5th International Workshop on Formal Methods for Blockchains (FMBC 2024).
Editors: Bruno Bernardo and Diego Marmsoler; Article No. 5; pp. 5:1–5:13
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

is important to obtain strong guarantees about the economic mechanisms of these protocols. While formal verification tools for smart contracts based on model-checking are useful in detecting programming bugs and even in proving some structural properties of AMMs [6, 8], they are not suitable for verifying, or even expressing more complex properties regarding the economic mechanism of AMMs. These economic mechanisms have been studied in several research works, which, in most cases, provide pen-and-paper proofs of the obtained properties. Given the complexity of the studied models, it would be desirable to also provide machine-verified proofs, so that we may rely on the proven properties beyond any reasonable doubt. To the best of our knowledge, existing mechanized formalizations [11] focus on verifying relevant structural properties of AMMs like their state consistency, and not on studying the economic mechanism of AMMs (see Section 4 for a detailed comparison).

Contributions

In this paper we formalize Automated Market Makers in the Lean 4 theorem prover. Our model is based on a slightly simplified version of the Uniswap v2 protocol (one of the leading AMMs), which was studied in [5] with a pen-and-paper formalization. We provide a Lean specification of blockchain states, abstracted from any factors that are immaterial to the study of AMMs. Then, we model the fundamental interactions that users may have with AMMs as well as the economic notions of price, networth and gain. Finally, we build machine-checked proofs of economic properties of constant-product AMMs. In particular, we derive an explicit formula for the economic gain obtained by a user after an exchange with an AMM. Building upon this formula we prove that, from a trader's perspective, aligning a constant-product AMM's internal exchange rate with the rate given by the trader's price oracle implies the optimal gain from that AMM. This results in the fundamental property of AMMs acting as price oracles themselves [1]. We then construct the optimal swap transaction that a rational user can perform to maximize their gain, solving the arbitrage problem. Our formalization and proofs[3] are made available in a public GitHub repository. At the best of our knowledge, this is the first mechanized formalization of the economic mechanism of AMMs. We finally discuss some open issues, and alternative design choices for formalizing AMMs.

2 Formalization

An Automated Market Maker implements a decentralized exchange between two different token types. The exchange rate is determined by a smart contract, which also takes care of performing the exchange itself: namely, the contract receives from a trader some amount of the input token type, and sends back the correct amount of the output token type, which is taken from the AMM reserves. A single smart contract can control many instances of AMMs (also called *AMM pairs*): we may have a pair for each possible unordered pair of token types. To create an AMM instance, a user must provide the initial liquidity for the reserves of that pair of tokens. Liquidity providers are rewarded with a type of token that specifically represents shares in that AMM's reserves: we call these *minted* token types, while any other token type will be called *atomic*.

[3] `https://github.com/danielepusceddu/lean4-amm`

Blockchain State

We begin by formalizing the blockchain state, abstracting from the details that are immaterial to the study of AMMs. Then, our model includes the users' wallets, the AMMs and their reserves (see Listing 1). We formalize the universes of users and atomic token types as the types `A` and `T`, respectively, as structures that encapsulate a natural number. Hereafter, we use $a, b, \ldots$ to denote users in `A`, and $\tau, \tau_0, \tau_1, \ldots$ to denote tokens in `T`. Minted token types are pairs of `T`. We represent the funds owned by a user by a *wallet* that maps token types to non-negative reals. To rule out wallets with infinite tokens, we use Mathlib's finitely supported functions[4]: in general, given any type α and any type M with a 0 element, $f \in \alpha \rightarrow_0 M$ if $\operatorname{supp}(f) = \{x \in \alpha \mid f(x) \neq 0\}$ is finite.

We define the type $\mathtt{W}_0$ of wallets of atomic tokens as a structure encapsulating $\mathtt{T} \rightarrow_0 \mathbb{R}_{\geq 0}$. This definition induces an element $0 \in \mathtt{W}_0$ such that $\operatorname{supp}(0) = \emptyset$: this is the *empty wallet*, which enables us to form the type $\mathtt{T} \rightarrow_0 \mathtt{W}_0$. We define the type $\mathtt{W}_1$ of wallets of minted tokens as a structure that encapsulates a function $\mathtt{bal} \in \mathtt{T} \rightarrow_0 \mathtt{W}_0$. The intuition is that $\mathtt{bal}\ \tau_0\ \tau_1$ gives the owned amount of the minted token type created by the AMM pair with tokens τ_0 and τ_1. Consistently, the function `bal` must satisfy two conditions: $\mathtt{bal}\ \tau_0\ \tau_1 = \mathtt{bal}\ \tau_1\ \tau_0$, meaning that the order of atomic tokens is irrelevant, and $\mathtt{bal}\ \tau\ \tau = 0$, meaning that the two token types in an AMM must be distinct. Our definition of $\mathtt{W}_1$ encapsulates proofs of these two properties, called `unord` and `distinct`, respectively.

We map users to their wallets with the types $\mathtt{S}_0$ and $\mathtt{S}_1$, which account for the atomic tokens and for the minted tokens, respectively. Finally, we formalize sets of AMM pairs with the type `AMMs`. The definition is strikingly similar to that of $\mathtt{W}_1$, but with a changed constraint. The intuition is that $\mathtt{res}\ \tau_0\ \tau_1$ gives the reserves of τ_0 in the AMM pair (τ_0, τ_1), while $\mathtt{res}\ \tau_1\ \tau_0$ gives the reserves of τ_1 in the same AMM pair. For uninitialized AMM pairs, the obtained reserves must be 0. The property `posres` ensures that either an AMM pair has no reserves of both token types (i.e., the AMM has not been created yet), or both token types have strictly positive reserves (i.e., one cannot deplete the reserves of a single token type in an AMM pair). We combine the previous definitions in the type Γ, which represents the state of a blockchain (note that Γ abstracts from all the details immaterial for AMMs).

Token supply

Given a blockchain state $s \in \Gamma$, we define the supply of an atomic token type τ_0 as:

$$\mathtt{atomsupply}_s(\tau_0) \quad = \sum_{a \in \operatorname{supp}(s.\mathtt{atoms})} (s.\mathtt{atoms}\ a)\ \tau_0 \ + \sum_{\tau_1 \in \operatorname{supp}(s.\mathtt{amms}\ \tau_0)} (s.\mathtt{amms}\ \tau_0)\ \tau_1$$

where the partial application $s.\mathtt{amms}\ \tau_0$ gives a map with all AMM pairs with τ_0 as one of their token types. We define the supply of a minted token type (τ_0, τ_1) as follows:

$$\mathtt{mintsupply}_s(\tau_0, \tau_1) \quad = \sum_{a \in \operatorname{supp}(s.\mathtt{mints})} (s.\mathtt{mints}\ a)\ \tau_0\ \tau_1$$

The corresponding Lean definitions (in Listing 2) have been split in order to facilitate theorem proving and, in particular, the use of Lean's simplifier.

[4] `https://leanprover-community.github.io/mathlib4_docs/Mathlib/Data/Finsupp/Defs.html`

Listing 1 Fundamental Lean definitions for the state of an AMM system.

```
structure A where
  n: ℕ

structure T where
  n: ℕ

structure W₀ where
  bal: T →₀ ℝ≥0

structure S₀ where
  map: A →₀ W₀

structure W₁ where
  bal: T →₀ W₀
  unord: ∀ (τ₀ τ₁:
    T),
    bal τ₀ τ₁ = f τ₁
    τ₀
  distinct: ∀ (τ: T),
    bal τ τ = 0

structure S₁ where
  map: A →₀ W₁

structure AMMs where
  res: T →₀ W₀
  distinct: ∀ (τ: T),
    res τ τ = 0
  posres: ∀ (τ₀ τ₁: T),
    res τ₀ τ₁ ≠ 0  ↔ f τ₁ τ₀ ≠
     0

structure Γ where
  atoms: S₀
  mints: S₁
  amms: AMMs
```

Listing 2 Supply of atomic token types and of minted token types.

```
1  noncomputable def S₀.supply (s: S₀) (τ: T): ℝ≥0 := s.map.sum (λ _ w => w τ)
2
3  noncomputable def S₁.supply (s: S₁) (τ₀ τ₁: T): ℝ≥0 :=
4      s.map.sum (λ _ w => w.get τ₀ τ₁)
5
6  noncomputable def AMMs.supply (amms: AMMs) (τ: T): ℝ≥0 :=
7      (amms.res τ).sum λ _ x => x
8
9  noncomputable def Γ.atomsupply (s: Γ) (τ: T): ℝ≥0 :=
10     (s.atoms.supply τ) + (s.amms.supply τ)
11
12 noncomputable def Γ.mintsupply (s: Γ) (τ₀ τ₁: T): ℝ≥0 := s.mints.supply τ₀ τ₁
```

AMM reserves

Given a blockchain state s and two token types τ_0, τ_1, the terms s.`amms` τ_0 τ_1 and s.`amms` τ_1 τ_0 denote, respectively, the reserves of τ_0 and τ_1 in the AMM pair (τ_0, τ_1). This way of accessing the AMM reserves is a bit impractical: when writing proofs, using s.`amms` τ_0 τ_1 carries an obligation to provide the functions `distinct` and `posres`. In particular, this requires the user to explicitly add, in any theorem using the reserves, the assumption that the reserves are strictly positive to indicate the AMM pair has been created. Furthermore, this way of accessing the reserves hides the fact that when one of the reserves is strictly positive, also the other one is such, which again should be made explicit when writing proofs.

To cope with these issues, we build a Lean API that allows for hiding these implementation details (see Listing 3). For example, given an AMM pair (τ_0, τ_1) in a state s, the expression s.`amms.r`$_0$ τ_0 τ_1 `init` gives the reserves of token τ_0 in the AMM, which are guaranteed to be strictly positive under the initialization precondition `init` $\in$ (s.`amms.init`).

Transactions

Our model encompasses all the main types of transactions supported by AMMs: creating an AMM, adding/removing liquidity, and swapping a token for another. Swaps are parameterised by a *swap rate function*, which determines the exchange rate. We use the formalization of swap transactions (Listing 4) to exemplify the scheme we used for all the transaction types.

Listing 3 Fragment of the AMM API: AMM reserves.

```
1 def AMMs.init (amms: AMMs) (τ0 τ1: T): Prop := amms.res τ0 τ1 ≠ 0
2
3 def AMMs.r0 (amms: AMMs) (τ0 τ1: T) (h: amms.init τ0 τ1): ℝ>0 := ⟨ amms.res τ0 τ1,
4     by unfold init at h; exact NNReal.neq_zero_imp_gt h ⟩
5
6 def AMMs.r1 (amms: AMMs) (τ0 τ1: T) (h: amms.init τ0 τ1): ℝ>0 := ⟨ amms.res τ1 τ0,
7     by unfold init at h; exact NNReal.neq_zero_imp_gt ((amms.posres τ0 τ1).mp h) ⟩
```

The type $\texttt{Swap}\,(sx, s, a, \tau_0, \tau_1, x)$ represents valid swap transactions in a blockchain state s, with the swap rate function sx, performed by user a to exchange x amount of the input token τ_0 for a certain amount of the output token τ_1 (Line 2). Each element of this type is a structure containing a proof of the validity of the transaction. For example, for swap transactions we must prove that the user has enough amount of τ_0 (condition `enough`), that the AMM pair with tokens τ_0 and τ_1 exists (condition `exi`), and it has enough reserves of τ_1 to give as output (condition `nodrain`). Since the type $\texttt{Swap}\,(\cdots)$ is empty when the parameters do not satisfy the above conditions, invalid transactions are not really expressible in our model. Instead, if $\texttt{Swap}\,(\cdots)$ represents a valid transaction, it will be a singleton type due to proof irrelevance (i.e., any two proofs of the same proposition are equal).

Each transaction is equipped with an `apply` function that yields the state reached by executing the transaction in the given state. For example, for $sw \in \texttt{Swap}\,(sx, s, a, \tau_0, \tau_1, x)$, `apply` sw yields a state where:

- a's atomic tokens wallet is updated by removing x units of τ_0 and adding $sw.\texttt{y}$ units of τ_1 (Line 12), where $sw.\texttt{y}$ is the amount of tokens outputted by the AMM pair;
- accordingly, the AMM reserves are updated by removing $sw.\texttt{y}$ units of τ_1 and adding x units of τ_0 (Line 14);
- the minted token wallets is unchanged (Line 13).

These definitions use functions and proofs not included in Listing 4 for brevity, such as `sub` and `sub_r1`. These are designed with the same spirit of allowing only valid operations, and so require suitable proofs. For example, `sub_r1` requires a proof of the existence of the AMM we are removing liquidity from, and a proof that the AMM pair has enough liquidity to retain a positive amount of reserves. We build these proofs inline using those contained in the structure: for instance, the parameter $sw.\texttt{exi}$ passed to `sub_r1` at line 11 is a proof that the AMM pair exists. Then, at line 16 we define the constant-product swap rate function, that is the swap rate function used by Uniswap v2. From Lemma 5 onwards, our results will focus on AMMs using this swap rate function.

Price, networth and Gain

An important aim of our model is to state and prove economic properties of AMMs related to the networth of their users. The fundamental definitions are in Listing 5. Given a wallet $w \in \texttt{W}_0$ and an atomic token price oracle $o \in \texttt{T} \to \mathbb{R}_{>0}$, we define the *value* of w in Line 2 as:

$$\texttt{value}\,(w, o) = \sum_{\tau \in \text{supp}(w)} w\,(\tau) \cdot o\,(\tau)$$

The value of a wallet of minted tokens $w \in \texttt{W}_1$ is defined similarly, except that:

- the summation ranges over $\text{supp}\,(w.\texttt{u})$, with $w.\texttt{u} \in \texttt{T}^2 \to_0 \mathbb{R}_{\geq 0}$ representing the uncurrying of w;
- the summation is divided by 2 since, if (τ_0, τ_1) is in the support of w, then also (τ_1, τ_0) is in its support.

Listing 4 Definition of the swap transaction type and of its application, as well as the constant-product swap rate function.

```
1  abbrev SX := ℝ>0 → ℝ>0 → ℝ>0 → ℝ>0
2  structure Swap (sx: SX) (s: Γ) (a: A) (τ0 τ1: T) (x: ℝ>0) where
3    enough: x ≤ s.atoms.get a τ0          -- user a has at least x τ0
4    exi: s.amms.init τ0 τ1                -- AMM pair τ0 τ1 exists in s
5    nodrain: x*(sx x (s.amms.r0 τ0 τ1 exi) (s.amms.r1 τ0 τ1 exi))
6              < (s.amms.r1 τ0 τ1 exi)   -- AMM has enough output tokens
7
8  def Swap.y (sw: Swap sx s a τ0 τ1 x): ℝ>0 :=
9      x*(sx x (s.amms.r0 τ0 τ1 sw.exi) (s.amms.r1 τ0 τ1 sw.exi))
10
11 noncomputable def Swap.apply (sw: Swap sx s a τ0 τ1 x): Γ := {
12   atoms := (s.atoms.sub a τ0 x sw.enough).add a τ1 sw.y,
13   mints := s.mints,
14   amms  := (s.amms.sub_r1 τ0 τ1 sw.exi sw.y sw.nodrain).add_r0 τ0 τ1 (by
       simp[sw.exi]) x }
15
16 noncomputable def SX.constprod: SX := λ (x r0 r1: ℝ+) => r1/(r0 + x)
```

For uniformity with the definition of value of atomic wallets, also here we assume an oracle that gives the price of (minted) tokens. However, while for pricing atomic tokens we indeed resort to an oracle, for minted tokens this oracle is instantiated to a specific function, coherently with [5]:

$$\texttt{mintedprice}_s(o, \tau_0, \tau_1) = \frac{(s.\texttt{amms.r}_0\ \tau_0\ \tau_1) \cdot o(\tau_0) + (s.\texttt{amms.r}_1\ \tau_0\ \tau_1) \cdot o(\tau_1)}{\texttt{mintsupply}_s(\tau_0, \tau_1)}$$

where we have omitted the initialization precondition `h` for brevity.

We then define the *networth* of a user as the sum of the value of their two types of wallets (Line 12). The *gain* of a user upon an update of the blockchain state is the difference between the networth in the new state and that in the old state (Line 16).

Reachable states

To formalize reachable states, we begin by defining sequences of transactions (Listing 6). $\texttt{Tx}\,(sx, s, s')$ is the type of sequences of valid transactions (of any kind) starting from state s and leading to s'. The parameter sx is the swap rate function used in swap transactions. Technically, $\texttt{Tx}\,(sx, s, s')$ is an instance of the indexed family of dependent types `Tx`, dependent on sx and s, and indexed by s'. In practice, this means that the constructors must preserve the values of sx and s (building upon the sequence of transactions does not change the swap rate function being used nor the originating state), while s' may change after each construction (the state resulting from the sequence changes with each transaction that is added to it). A state s' is *reachable* if there exists a valid sequence of transactions that reaches s' starting from a valid *initial* state s, i.e. a state with no initialized AMMs or minted token types in circulation.

Listing 5 Users' networth and gain.

```
noncomputable def W₀.value (w: W₀) (o: T → ℝ>0): ℝ≥0 :=
  w.sum (λ τ x => x*(o τ))

noncomputable def W₁.value (w: W₁) (o: T → T → ℝ≥0): ℝ≥0 :=
  (w.u.sum (λ p x => x*(o p.fst p.snd))) / 2

noncomputable def Γ.mintedprice (s: Γ) (o: T → ℝ>0) (τ₀ τ₁: T): ℝ≥0 :=
  if h:s.amms.init τ₀ τ₁ then
  ((s.amms.r₀ τ₀ τ₁ h)*(o τ₀)+(s.amms.r₁ τ₀ τ1 h)*(o τ₁)) / (s.mints.supply τ₀ τ₁)
  else 0 -- price is zero if AMM is not initialized

noncomputable def Γ.networth (s: Γ) (a: A) (o: T → ℝ>0): ℝ≥0 :=
  (W₀.value (s.atoms.get a) o) + (W₁.value (s.mints.get a) (s.mintedprice o))

noncomputable def A.gain (a: A) (o: T → ℝ>0) (s s': Γ): ℝ :=
  ((s'.networth a o): ℝ) - ((s.networth a o): ℝ)
```

Listing 6 Sequences of transactions and reachable states.

```
inductive Tx (sx: SX) (init: Γ): Γ → Type where
  | empty: Tx sx init init

  | swap (s': Γ) (rs: Tx sx init s')
         (sw: Swap sx s' a τ₀ τ₁ v0):
      Tx sx init sw.apply
  -- Other constructors omitted for brevity

def validInit (s: Γ): Prop :=
  (s.amms = AMMs.empty ∧ s.mints = S₁.empty)

def reachable (sx: SX) (s: Γ): Prop :=
  ∃ (init: Γ) (tx: Tx sx init s), validInit init
```

3 Results

We now present some noteworthy properties of AMMs that we have proven in Lean.

Proposition 1 ensures that, in any reachable state, there exists an AMM with token types τ_0 and τ_1 if and only if the minted token type (τ_0, τ_1) is in circulation, i.e. it has a strictly positive supply. This result showcases the validity of our model with regards to reasoning about reachable states. Technically, it also allows us to prove that $\texttt{mintedprice}_s(o, \tau_0, \tau_1)$ is strictly positive for any initialized AMM pair (τ_0, τ_1) in any reachable state s.

▶ **Proposition 1** (Existence of AMMs *vs.* minted token supply)**.** *Let $s' \in \Gamma$ be a reachable blockchain state. Then, for any minted token type (τ_0, τ_1), its supply in s' is strictly positive if and only if s' has an AMM with token types τ_0 and τ_1.*

Proof. By induction on the length of the sequence of transactions leading to s':

- Base case: empty transaction sequence. The proof is trivial for both directions, since a valid starting state has no initialized AMMs and no minted tokens in circulation.

- Inductive case: there are several subcases depending on the last transaction fired in the sequence. Here we consider the creation of the AMM (τ_0', τ_1') in reachable state s'. We proceed by cases on the truth of the equality $\{\tau_0, \tau_1\} = \{\tau_0', \tau_1'\}$. If it is a different token pair, then the supply remains unchanged along with the initialization status, and we can conclude by the induction hypothesis. If it is the same token pair, then we just incremented its minted token supply (which is non-negative, so after incrementing it, it must be strictly positive), and we just initialized the AMM.
- See source code for the other cases. AMMLib/Transaction/Trace.lean:275 ◀

Lemma 2 allows us to determine the change in the value of a wallet after it has been updated in some way: the resulting equality is the basis for all the subsequent proofs. To illustrate it, we briefly introduce two definitions that have been omitted before: $\texttt{drain}_0(w_0, \tau_0)$ is the atomic token wallet such that $\texttt{drain}_0(w_0, \tau_0)(\tau_0) = 0$ and $\texttt{drain}_0(w_0, \tau_0)(\tau_1) = w_0(\tau_1)$ for every other token $\tau_1 \neq \tau_0$. We define $\texttt{drain}_1(w_1, (\tau_0, \tau_0)) \in \texttt{W}_1$ similarly.

▶ **Lemma 2** (Value expansion). *Let $w_0 \in \texttt{W}_0$, $o_0 \in \texttt{T} \to \mathbb{R}_{>0}$, and $\tau_0, \tau_1 \in \texttt{T}$. Then,*

$$\texttt{value}\,(w_0, o_0) \;=\; w_0(\tau_0) \cdot o(\tau_0) + \texttt{value}\,(\texttt{drain}_0(w_0, \tau_0), o_0)$$

and, with $w_1 \in \texttt{W}_1$ and $o_1 \in \texttt{T}^2 \to \mathbb{R}_{>0}$ such that $o_1(\tau_0, \tau_1) = o_1(\tau_1, \tau_0)$,

$$\texttt{value}\,(w_1, o_1) \;=\; w_1(\tau_0, \tau_1) \cdot o_1(\tau_0, \tau_1) + \texttt{value}\,(\texttt{drain}_1(w_1, \tau_0, \tau_1), o_1)$$

Proof. By definition of value and by properties of the sum over a finite support. Full Lean proof at AMMLib/State/AtomicWall.lean:116 ◀

Lemma 3 gives an explicit formula for the gain obtained by a user upon firing a swap transaction. It is fundamental to all proofs involving gain.

▶ **Lemma 3** (Gain of a swap). *Let $sw \in \texttt{Swap}\,(sx, s, a, \tau_0, \tau_1, x)$ and let $o \in \texttt{T} \to \mathbb{R}_{>0}$. Then,*

$$\texttt{gain}\,(a, o, s, \texttt{apply}\ sw) = (sw.\texttt{y} \cdot o\,(\tau_1) - x \cdot o\,(\tau_0)) \cdot \left(1 - \frac{(s.\texttt{mints}\ a\ \tau_0\ \tau_1)}{\texttt{mintsupply}_s\,(\tau_0, \tau_1)}\right)$$

Proof. By repeated application of Lemma 2 in order to isolate the value contributed by the token types involved in the swap, and by use of Mathlib's `ring_nf` simplifier tactic. AMMLib/Transaction/Swap/Networth.lean:55 ◀

Lemma 4 establishes a correspondence between the profitability of a swap transaction (i.e., a positive or negative gain) and the order between the swap rate and the exchange rate given by the price oracle. In particular, assuming a trader a who is not a liquidity provider (i.e., a has no minted tokens for the AMM pair targeted by the swap), Lemma 4 states that:

- $\texttt{gain}\,(a, o, s, \texttt{apply}\ sw) < 0 \iff sx\,(x, r_0, r_1) < {}^{o(\tau_0)}/_{o(\tau_1)}$
- $\texttt{gain}\,(a, o, s, \texttt{apply}\ sw) = 0 \iff sx\,(x, r_0, r_1) = {}^{o(\tau_0)}/_{o(\tau_1)}$
- $\texttt{gain}\,(a, o, s, \texttt{apply}\ sw) > 0 \iff sx\,(x, r_0, r_1) > {}^{o(\tau_0)}/_{o(\tau_1)}$

Technically, to formalize this result it is convenient to use Mathlib's `cmp`[5], which gives the order between the two parameters.

---

[5] https://leanprover-community.github.io/mathlib4_docs/Mathlib/Init/Data/Ordering/Basic.html#cmp

▶ **Lemma 4** (Swap rate *vs.* exchange rate)**.** *Let* $sw \in \texttt{Swap}\,(sx, s, a, \tau_0, \tau_1, x)$ *be a swap transaction, and let* $o \in \texttt{T} \to \mathbb{R}_{>0}$ *be a price oracle. For* $i \in \{0,1\}$*, let* $r_i = s.\texttt{amms}.\texttt{r}_i\,(\tau_0, \tau_1)$*. If* $s.\texttt{mints}\,(a)\,(\tau_0, \tau_1) = 0$*, then*

$$\texttt{cmp}\,(\texttt{gain}\,(a, o, s, \texttt{apply}\ sw)\,, 0) \;=\; \texttt{cmp}\left(sx\,(x, r_0, r_1)\,, \frac{o\,(\tau_0)}{o\,(\tau_1)}\right)$$

Proof. By term manipulation and Lemma 3. AMMLib/Transaction/Swap/Networth.lean:86 ◀

Lemma 5 establishes that, in a constant-product AMM, there exists only one profitable direction for a swap. Namely, if swapping τ_0 for τ_1 gives a positive gain, then swapping in the other direction (i.e., τ_1 for τ_0) will give a negative gain. Note that the inverse does not hold: a negative gain in a direction does not imply a positive gain in the other direction.

▶ **Lemma 5** (Unique direction for swap gain)**.** *Let* $sw \in \texttt{Swap}\,(\texttt{constprod}, s, a, \tau_0, \tau_1, x)$ *and* $sw' \in \texttt{Swap}\,(\texttt{constprod}, s, a, \tau_1, \tau_0, x')$ *be two swap transactions in opposite directions, and let* $o \in \texttt{T} \to \mathbb{R}_{>0}$*. If* $\texttt{gain}\,(a, o, s, \texttt{apply}\ sw) > 0$*, then* $\texttt{gain}\,(a, o, s, \texttt{apply}\ sw) < 0$*.*

Proof. By Lemma 4. AMMLib/Transaction/Swap/Constprod.lean:160 ◀

▶ **Example 6.** Consider a blockchain state s and atomic tokens τ_0 and τ_1, and assume that:
- a is a trader with no minted tokens, i.e. $(s.\texttt{mints}\ a)\ \tau_0\ \tau_1 = 0$;
- the AMM pair for (τ_0, τ_1) has been initialized in s and has reserves $r_0 = (s.\texttt{amms}\ \tau_0\ \tau_1) = 18$ and $r_1 = (s.\texttt{amms}\ \tau_1\ \tau_0) = 6$;
- all the AMMs in s use the constant-product swap rate function;
- o is a price oracle such that $o\ \tau_0 = 3$ and $o\ \tau_1 = 4$.

Assume that a wants to sell 6 units of τ_1 for some units of τ_0 with the swap transaction $sw \in \texttt{Swap}(\texttt{constprod}, s, a, \tau_1, \tau_0, 6)$. Then, by Lemma 3, a's gain is given by $9{\cdot}3 - 24 = 3 > 0$. Coherently with Lemma 5, any swap in the opposite direction would give a negative gain: e.g., if a sells 6 units of τ_0, her gain would be $3/2 \cdot 4 - 18 = -12$.

We say that a swap transaction $sw \in \texttt{Swap}\,(\texttt{constprod}, s, a, \tau_0, \tau_1, x)$ is *optimal* for a given price oracle o when, for all $sw' \in \texttt{Swap}\,(\texttt{constprod}, s, a, \tau_0, \tau_1, x')$ with $x' \neq x$ we have

$$\texttt{gain}\,(a, o, s, \texttt{apply}\ sw') < \texttt{gain}\,(a, o, s, \texttt{apply}\ sw)$$

Theorem 7 gives a sufficient condition for the optimality of swaps in constant-product AMMs: it suffices that the ratio of the AMM's reserves after the swap is equal to the exchange rate given by the price oracle. Intuitively, the condition in Theorem 7 means that the exchange rate between the two token types induced by the AMM (i.e., the ration between the token reserves) is aligned with the exchange rate given by the price oracle, and so further swaps would yield a negative gain. Note that, by definition, if a swap is optimal then it is also unique, i.e. swapping any other amount would yield a suboptimal gain.

▶ **Theorem 7** (Sufficient condition for optimal swaps)**.** *Let* $sw \in \texttt{Swap}\,(\texttt{constprod}, s, a, \tau_0, \tau_1, x)$ *be a swap transaction on a constant-product AMM, and let* $o \in \texttt{T} \to \mathbb{R}_{>0}$ *be a price oracle. For* $i \in \{0,1\}$*, let* $r'_i = (\texttt{apply}\ sw)\,.\texttt{amms}.\texttt{r}_i\,(\tau_0, \tau_1)$ *be the AMM reserves after the swap. If* ${r'_1}/{r'_0} = {o(\tau_0)}/{o(\tau_1)}$*, then* sw *is optimal.*

Proof. By cases on $x < x'$ and by application of Lemma 4. AMMLib/Transaction/Swap/-Constprod.lean:184 ◀

▶ **Example 8.** Under the assumptions of Example 6, the exchange rate between τ_1 and τ_0 given by the price oracle is $(o\ \tau_0)/(o\ \tau_1) = 3/4$, while the exchange rate induced by the AMM (i.e., the ratio between the reserves) is $r_1/r_0 = 1/3$. Hence, to satisfy the equality in Theorem 7 we must perform a swap that increases r_1 and decreases r_0, i.e. a must sell units of τ_1 to buy units of τ_0, coherently with the necessary condition for a positive gain in Example 6. Theorem 9 below will establish exactly how many units of τ_1 must be traded.

Theorem 9 gives an explicit formula for the input amount x that yields an optimal swap transaction for a given AMM pair, under the assumption that the user firing the transaction does not hold the AMM's minted token type. The other implicit assumption is that the user a firing the swap has the needed amount x of units of the sold token, i.e. $x \leq (s.\texttt{atoms}\ a)\ \tau_1$. In practice, this assumption can always be satisfied with *flash loans*, which allow a to borrow the amount x, perform the swap, and then return the loan in a single, atomic transaction.

▶ **Theorem 9** (Arbitrage for constant-product AMMs). *Let* $sw \in \texttt{Swap}\,(\texttt{constprod}, s, a, \tau_0, \tau_1, x)$ *be a swap transaction on a constant-product AMM, and let* $o \in \texttt{T} \to \mathbb{R}_{>0}$ *be a price oracle. For* $i \in \{0,1\}$*, let* $r_i = s.\texttt{amms}.\texttt{r}_i\,(\tau_0, \tau_1)$ *be the AMM reserves. If* a *has no minted tokens (i.e.,* $s.\texttt{mints}\,(a)\,(\tau_0, \tau_1) = 0$*) then* sw *is optimal if the amount of traded units of* τ_0 *is:*

$$x \;=\; \sqrt{\frac{o(\tau_1) \cdot r_0 \cdot r_1}{o(\tau_0)}} - r_0$$

Proof. By algebraic manipulation and Theorem 7. AMMLib/Transaction/Swap/Constprod.lean:316 ◀

▶ **Example 10.** Under the assumptions of Example 8, we know that to perform a profitable swap the trader must sell units of τ_1, i.e. fire a transaction $\texttt{Swap}\,(\texttt{constprod}, s, a, \tau_1, \tau_0, x)$. Theorem 9 gives the optimal input value x, i.e., the number of sold units of τ_1:

$$x \;=\; \sqrt{\frac{3 \cdot 6 \cdot 18}{4}} \;-\; 6 \;=\; 3$$

Then, the output amount is given by $sw.\texttt{y} = 3 \cdot 18/(6+3) = 6$ and, by Lemma 3, the gain of a is $6 \cdot 3 - 3 \cdot 4 = 6$, which maximizes it. Note that, in the new state, the exchange rate given by the AMM coincides with that given by the price oracle: $(r_1 + x)(r_0 - sw.\texttt{y}) = (o\ \tau_0)/(o\ \tau_1)$.

4 Related work

The closest work to ours is [11], which proposes a methodology for developing and verifying AMMs in the Coq proof assistant. In this approach, an AMM is decomposed into multiple interacting smart contract: e.g., each minted token is modelled as a single smart contract, following the way fungible tokens are encoded in blockchains platforms that do not provide custom tokens natively, as e.g. in Ethereum and Tezos. These smart contracts are then implemented as Coq functions on top of ConCert [3], a generic model of blockchain platforms and smart contracts mechanized in Coq. Concert is used to verify behavioural properties of smart contracts, either in isolation or composed with other smart contracts: this is fundamental for [11], where AMMs are specified as compositions of multiple contracts.

More specifically, [11] applies the proposed methodology to the Dexter2 protocol, which implements a constant-product AMM based on Uniswap v1 on the Tezos blockchain[6]. The main properties of AMMs proved in [11] are correspondences between the state of AMMs and the sequences of transactions executed on the blockchain. In particular, they prove that:

- the balance recorded in the main AMM contract is coherent with the actual balance resulting from the execution of the sequence of transactions;
- the supply of the minted token recorded in the main AMM contract is equal to the actual supply resulting from the execution;
- the state of the minted token contract is coherent with the execution.

Furthermore, starting from the Coq specification of the Dexter2 AMM, [11] extracts verified CamlLigo code, which is directly deployable on the Tezos blockchain.

Although both our work and [11] involve AMM formalizations within a proof assistant, the ultimate goals are quite different. The formalization in [11] closely follows the concrete implementation of a particular AMM instance (Dexter2) and produces a deployable implementation that is provably coherent with the proposed Coq specification. By contrast, we start from a more abstract specification of AMM, with the goal of studying the properties that must be satisfied by any implementation coherent with the specification. An advantage of our approach over [11] is that it provides a suitable level of abstraction where proving properties about the economic mechanisms of AMMs, i.e. properties about the gain of users and of equilibria among users' strategies. In particular, Theorem 9 establishes a paradigmatic property of AMMs, which explains the economic mechanism underlying their design.

Besides these main differences, our AMM model and that in [11] have several differences. A notable difference is that our model is based on Uniswap v2, while [11] is based on Uniswap v1. In particular, this means that our AMMs can handle arbitrary token pairs, while in Dexter2 any AMM pairs a token with the blockchain native crypto-currency.

At the best of our knowledge, [11] and ours are currently the only mechanized formalizations of AMMs in a proof assistant. Many other works study economic properties of AMMs that go beyond those proved in this paper. The works [2] and [1] study, respectively, an AMM model based on Uniswap similar to ours, and a generalization of the model where the AMM is parameterised over a *trading function* of the AMM reserves, which must remain constant before and after any swap transactions (in Uniswap v1 and v2, the trading function is just the product between the AMM reserves). The work [5] studies another generalization of Uniswap v2, where the relation between input and output tokens of swap transactions is determined by an arbitrary *swap rate function*, studying the properties of this function that give rise to a sound economic mechanism of AMMs. While both [1] and [5] share the common goal of providing general models of AMMs wherein to study their economic behaviour, they largely diverge on the formalization: [1] is based on concepts related to convex optimization problems, while [5] borrows formalization and reasoning techniques from concurrency theory. The Lean 4 model proposed in this paper follows the formalization in [5], which has the advantage of requiring far less mathematical dependencies: although Mathlib is equipped to reason about convex sets and functions[7], it is currently lacking in advanced convex optimization definitions and results as those used in [1].

Our AMM model is based on Uniswap v2, one of the most successful AMMs so far. We briefly discuss some alternative AMM constructions. Balancer [4] generalizes the constant-product function used by Uniswap to a constant (weighted geometric) mean $f(r_1, \cdots, r_n) =$

[6] https://gitlab.com/dexter2tz/dexter2tz/-/tree/master/
[7] https://leanprover-community.github.io/mathlib4_docs/Mathlib/Analysis/Convex/Function

$\prod_{i=1}^{n} r_i^{w_i}$, where the weight w_i reflects the relevance of a token τ_i in a tuple $(\tau_1, \cdots, \tau_n)$. Curve [7] mixes constant-sum and constant-product functions, aiming at a swap rate with small fluctuations for large amounts of swapped tokens. The work [9] studies a variant of the constant-product swap rate invariant, where the rate adjusts dynamically based on oracle prices, with the goal of reducing the need for arbitrage transactions. Other approaches aiming at the same goal are studies in [12, 9], and implemented in [10]. Extending our Lean formalization and results to these alternative AMM designs would require a substantial reworking of our model and proofs.

5 Conclusions

In this work we have provided a formalization of AMMs in Lean. Blockchain states are represented as structures containing wallets and AMMs, and transactions as dependent types equipped with a function that defines the state resulting from firing the transaction. Based on this, we have modeled the key economic notions of price, networth and gain. We have then focused on the economic properties of AMMs, constructing machine-checkable proofs. In Lemma 3 we have given an explicit formula for the economic gain of a user after firing a swap transaction. In Theorem 7 we have proved that the rational strategy for traders leads to the alignment between the AMM internal exchange rate and that given by price oracles. Finally, in Theorem 9 we have derived the amount of tokens that a trader should sell to maximize the gain from a constant-product AMM.

Design choices

Before coming up with our Lean formalization, we have experimented with a few alternative definitions. Currently, we use the two pairs of atomic token types (τ_0, τ_1) and (τ_1, τ_0) to represent the same minted token type. Initially, we used the type M of sets of atomic tokens of cardinality 2, and modeled wallets of minted tokens by the type $M \to_0 \mathbb{R}_{\geq 0}$. However, using M in the definition of AMMs turned out not to be as easy. The type $M \to_0 \mathbb{R}^2_{\geq 0}$ would obviously not work since we would not know which value corresponds to the reserve of which token. On the other hand, the dependent type $(m : M) \to_0 \texttt{Option}\,(m \to \mathbb{R}_{>0})$ would work after defining $0 := \texttt{None}$, at the cost of losing the straightforward definition for supply of atomic tokens. We have opted for the custom subtype $\mathbb{R}_{>0}$ to represent the positive reals, since they simplify writing certain definitions and proof passages (e.g., avoiding the use of garbage outputs in the definitions where negative inputs would not make sense). This choice however turned out to have some cons, since it makes using Mathlib more complex, and in some cases we have to coerce back to the reals anyway (e.g. when reasoning about the gain). Using Mathlib's reals would perhaps lead to a smoother treatment.

Limitations

Compared to real-world AMM implementations, our Lean formalization introduces a few simplifications, that overall contribute to keeping our proofs manageable. Bridging the gap with real AMMs would require several extensions, which we discuss below as directions for future work. AMMs typically implement a trading fee $\phi \in [0, 1]$ that represents the portion of the swap amount kept by the AMM. While modeling the fee would be easy (ϕ would be an additional parameter to `SX` functions, to `Swap` types, and to transactions `Tx`), it would require a major reworking of all the results that deal with swaps. Our swaps have *zero-slippage*, in that either a swap gives exactly the amount of tokens required by a user, or they are aborted.

While on the one hand this is desirable (e.g., it rules out sandwich attacks), on the other hand it has drawbacks related to liveness, since a user may need to repeatedly send swap transactions until one is accepted. Real-world AMM implementations allow users to specify a slippage tolerance in the form of the minimum amount of tokens they expect from a swap. Extending our model to encompass slippage tolerance would require to add a parameter to each transaction type, and a minor reworking of the results. Some AMM implementations allow users to create AMMs pairs involving *minted* token types. Consequently, the tokens minted by these AMMs in general are "nestings" of token types. Extending our model in this direction would require to replace price oracles in our results with a suitable price function.

References

1. Guillermo Angeris and Tarun Chitra. Improved price oracles: Constant function market makers. In *ACM Conference on Advances in Financial Technologies (AFT)*, pages 80–91. ACM, 2020. doi:10.1145/3419614.3423251.
2. Guillermo Angeris, Hsien-Tang Kao, Rei Chiang, Charlie Noyes, and Tarun Chitra. An analysis of Uniswap markets. *Cryptoeconomic Systems*, 1(1), 2021. doi:10.21428/58320208.c9738e64.
3. Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. Concert: a smart contract certification framework in Coq. In *ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pages 215–228. ACM, 2020. doi:10.1145/3372885.3373829.
4. Balancer whitepaper, 2019. URL: https://balancer.finance/whitepaper/.
5. Massimo Bartoletti, James Hsin yu Chiang, and Alberto Lluch-Lafuente. A theory of automated market makers in DeFi. *Logical Methods in Computer Science*, Volume 18, Issue 4, December 2022. doi:10.46298/lmcs-18(4:12)2022.
6. Certora. Formal verification of Compound's open-oracle with Uniswap anchor. https://files.safe.de.fi/safe/files/audit/pdf/CompoundUniswapAnchoredOpenOracleAug2020.pdf, 2020.
7. Michael Egorov. Stableswap - efficient mechanism for stablecoin, 2019. URL: https://curve.fi/files/stableswap-paper.pdf.
8. Uri Kirstein. Detecting corner cases in Compound V3 with formal specifications. https://medium.com/certora/detecting-corner-cases-in-compound-v3-with-formal-specifications-b7abf137fb15, 2022.
9. Bhaskar Krishnamachari, Qi Feng, and Eugenio Grippo. Dynamic curves for decentralized autonomous cryptocurrency exchanges. In *International Symposium on Foundations and Applications of Blockchain (FAB)*, volume 92 of *OASIcs*, pages 5:1–5:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/OASIcs.FAB.2021.5.
10. Mooniswap whitepaper, 2020. URL: https://mooniswap.exchange/docs/MooniswapWhitePaper-v1.0.pdf.
11. 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.
12. Improving frontrunning resistance of x*y=k market makers, 2018. URL: https://ethresear.ch/t/improving-front-running-resistance-of-x-y-k-market-makers/1281.
13. Jiahua Xu, Krzysztof Paruch, Simon Cousaert, and Yebo Feng. Sok: Decentralized exchanges (DEX) with automated market maker (AMM) protocols. *ACM Comput. Surv.*, 55(11):238:1–238:50, 2023. doi:10.1145/3570639.

Towards Benchmarking of Solidity Verification Tools

Massimo Bartoletti
University of Cagliari, Italy

Fabio Fioravanti
University of Chieti-Pescara, Italy

Giulia Matricardi
University of Chieti-Pescara, Italy

Roberto Pettinau
Technical University of Denmark, Lyngby, Denmark

Franco Sainas
EPFL, Lausanne, Switzerland

Abstract

Formal verification of smart contracts has become a hot topic in academic and industrial research, given the growing value of assets managed by decentralized applications and the consequent incentive for adversaries to tamper with them. Most of the current research on the verification of contracts revolves around Solidity, the main high-level language supported by Ethereum and other leading blockchains. Although bug detection tools for Solidity have been proliferating almost since the inception of Ethereum, only in the last few years we have seen verification tools capable of proving that a contract respects some desirable properties. An open issue is how to evaluate and compare the effectiveness of these tools: indeed, the existing benchmarks for general-purpose programming languages cannot be adapted to Solidity, given substantial differences in the programming model and in the desirable properties. We address this problem by proposing an open benchmark for Solidity verification tools. By exploiting our benchmark, we compare two leading tools, SolCMC and Certora, discussing their completeness, soundness and expressiveness limitations.

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

Keywords and phrases Smart contracts, Ethereum, Verification, Blockchain

Digital Object Identifier 10.4230/OASIcs.FMBC.2024.6

Supplementary Material *Software*: `https://github.com/fsainas/contracts-verification-benchmark`, archived at `swh:1:dir:5452a36f0c58435ad4b7dadb7e7563653ffc25b6`

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.

1 Introduction

The rapid growth of decentralized applications based on blockchain technologies have emphasized the importance of ensuring the security of smart contracts – the basic building blocks of these applications. The research on smart contracts security has been proliferating since 2016, leading on the one side to the discovery of a variety of attacks, and on the other side to the development of several tools to detect vulnerabilities of smart contracts before they are deployed. Despite the increasing breadth and precision of these analysis tools, attacks to smart contracts have caused financial losses worth several billions of dollars so far, and are unlikely to be eradicated anytime soon.

© Massimo Bartoletti, Fabio Fioravanti, Giulia Matricardi, Roberto Pettinau, and Franco Sainas; licensed under Creative Commons License CC-BY 4.0

5th International Workshop on Formal Methods for Blockchains (FMBC 2024).
Editors: Bruno Bernardo and Diego Marmsoler; Article No. 6; pp. 6:1–6:15
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

A large class of analysis tools for smart contracts are focussed on detecting known vulnerability patterns in contracts code. Even though tools of this type can detect many nefarious bugs, statistically the vast majority of the losses due to real-world attacks are caused by logic errors in the contract code, which cannot be prevented by only checking for fixed vulnerability patterns [14]. In this context, a contract can be considered secure when its executions are coherent with some ideal behaviour, even in the presence of adversaries trying to subvert it. Only a few tools support this kind of security analysis, allowing developers to specify the ideal properties the contract is expected to satisfy. In this work we focus on SolCMC and Certora, two leading verification tools for contracts written in Solidity, the main smart contract language for Ethereum and EVM-compatible blockchains. Both tools allow the developer to specify desirable contract properties, and use SMT solvers to verify whether the contract satisfies them, showing a counterexample when detecting a violation. Although both tools have been independently tested by their developers [1, 10], no public comparison exists so far to assess their effectiveness and limitations in practice.

Our long-term goal is a comprehensive, publicly available benchmark to evaluate the effectiveness of verification tools for Solidity contracts. As an initial step towards this goal, in this paper we present a benchmark comprising 323 verification tasks, each one made of a Solidity contract and a property it is expected to satisfy.[1] A crucial component of our benchmark is a manually crafted ground truth of the verification tasks, encompassing multiple versions of each smart contract in order to cover different ways of satisfying or violating its associated properties. To foster the reproducibility of the results, we make available a toolchain that automatises the construction of the verification tasks, their processing with SolCMC and Certora, and the summarisation of the results. Based on these artifacts, we present a preliminary evaluation of SolCMC and Certora, comparing their completeness, soundness, and expressiveness. Finally, we introduce a scoring scheme for Solidity verification tools, which is inspired by schemes used in software verification competitions [7], but taking into account the peculiarities of the smart contracts context.

2 Background and related work

Over the years, several dozens of tools have been developed to analyse Ethereum contracts (see e.g. [26, 23, 21] for systematic surveys). The vast majority of these tools focus on specific types of contract vulnerabilities, such as reentrancy, integer overflow and underflow, mishandled exceptions, transaction ordering dependence, *etc.* [19]. Some tools focus on runtime verification of contracts [4], to force failures when some property violation is detected at run-time. More recent tools give users more control on the properties to be verified, in principle enabling the verification of contract implementations against an ideal, abstract description of their behaviour.

A prominent tool in this category is SolCMC [2], a symbolic model checker integrated in the Solidity compiler since 2019. Specifying properties in SolCMC requires developers to instrument the contract code with `assert` statements, which are treated as verification targets. Failure of an `assert` means that the desired property is not satisfied by the contract. For example, consider a method `deposit` that receives ETH from any user, recording the sent amount in a `balances` mapping. The property "after a successful `deposit`, the balance entry of `msg.sender` is increased by `msg.value`" can be encoded as the function in Listing 1:

[1] https://github.com/fsainas/contracts-verification-benchmark

Listing 1 SolCMC encoding of a safety property.

```
function deposit_user_balance() public payable {
  uint old_user_balance = balances[msg.sender];
  deposit();
  uint new_user_balance = balances[msg.sender];
  assert(new_user_balance == old_user_balance + msg.value);
}
```

This defines a contract invariant that must be true for any reachable contract state: the balance of a user after a successful call is equal to the previous balance plus the deposit. SolCMC transforms the instrumented contract into a set of Constrained Horn Clauses (CHC) [8, 16] which is fed to a CHC satisfiability solver (Spacer [25], integrated in Z3 [17], or Eldarica [22]) to check if any assert can fail. If so, it produces a trace witnessing the violation.

Certora [24, 6] is another leading formal verification tool for Solidity. Unlike SolCMC, it decouples the specification of the properties from the contract code. Properties, written in the Certora Verification Language (CVL), roughly can take the form of `assert` statements ("for all contract runs, the condition holds") or `satisfy` statements ("there exists a run where the condition holds"). For instance, the CVL specification of the `deposit` property seen before is shown in Listing 2. Certora compiles the Solidity contract and its associated properties into a logical formula, and sends it to an SMT solver. Another key difference between Certora and SolCMC is that in SolCMC verification is done locally (since it is part of the Solidity compiler stack), while in Certora it is executed remotely on a cloud service.

Listing 2 Certora encoding of a safety property.

```
rule deposit_user_balance {
  env e; // an arbitrary transaction and context
  address sender = e.msg.sender;
  mathint old_user_balance = getBalanceEntry(sender);
  deposit(e); // calls deposit with context e
  mathint new_user_balance = getBalanceEntry(sender);
  mathint deposit_amount = to_mathint(e.msg.value);
  assert new_user_balance == old_user_balance + deposit_amount;
}
```

Besides SolCMC and Certora, other tools for verifying user-defined properties of Solidity contracts have been proposed (see [2] for a comparison). VerX [29] models properties in a variant of past linear temporal logic. This allows to verify safety properties of contracts, while liveness properties are not expressible. SmartACE [34] verifies properties written in Scribble [15]: contracts are annotated with Scribble annotations (i.e., contract invariants and method postconditions). Scribble transforms the annotated contract into a contract with `assert`s, which are used as verification targets. SmartACE uses local bundle abstractions to reduce the state explosion caused by having to deal with many users interacting with the contract, factorising users into a representative few. It models each contract in LLVM-IR and integrates existing analysers such as SeaHorn and Klee to facilitate verification. Notably, SmartAce has been applied to verify some contracts from the OpenZeppelin library [35].

Comparing verification tools

A primary source of comparison among different verification tools is given by the research papers where these tools were introduced [20, 33, 30]. A problem here is that each comparison is based on an ad-hoc dataset of contracts and properties, which makes it difficult to compare

the effectiveness of different tools. The work [3] provides a unifying view of these datasets, by collecting their verification tasks and judgements, and mapping them to a uniform scheme based on the Smart Contract Weakness Classification [19]. A main difference between this dataset and ours is in the nature of the properties in the verification tasks: the ones in [3] are specific vulnerabilities (e.g., reentrancy, overflows, *etc.*), while ours are ideal properties of the analysed contract (e.g., “after calling `foo`, the sender receives 1 ETH”).

A few works compare different analysis tools without introducing their own. The work [14] evaluates five tools based on their ability to identify vulnerabilities that have been actually exploited by attacks in the wild. Perhaps surprisingly, the conclusion is that tools that detect specific vulnerability patterns are ineffective against real attacks, being able to counter only $\sim$12% to the economic damage in the considered dataset, while offering no protection against the remaining part of the damage, which exceeds 2 billion dollars. This is a strong motivation for research on analysis techniques and tools that can also detect logic-related bugs, which are the focus of our benchmark. The work [18] proposes a vulnerability classification scheme that extends [19], and evaluates the effectiveness of three bug detection tools. We note that both works [18, 14] focus on tools that detect specific vulnerabilities: at the best of our knowledge, ours is the first comparison between general verification tools for Solidity. Another main difference between our work and [18] is that the comparison in [18] is based on a quantitative evaluation of the tool outcomes (in the form of a confusion matrix), while we also devise a qualitative comparison that explains the reason behind these results, and in particular the causes of unsoundness (false positives) and incompleteness (false negatives).

3 Our benchmark

The benchmark is logically organized in the following components:

- a collection of informal specifications of use cases for smart contracts, each accompanied by a set of desirable properties. We deliberately choose *not* to use a formal language to write the smart contract specifications and the associated properties, since we want to be free to express properties that go beyond those expressible by current verification tools.
- Solidity implementations of the use cases and specifications of their properties in the languages supported by SolCMC and Certora. For each use case we provide multiple Solidity implementations, either respecting the given properties or violating them (in obvious or subtle ways). A *verification task* comprises the implementation of a use case and that of a related property.
- a *ground truth* that assesses, for each verification task, whether the implementation satisfies the associated property or not.

Our toolchain processes these data to construct the verification task and runs SolCMC and Certora on each of them.

The way we construct the verification tasks is tool-specific:

- for SolCMC, each property is encoded in Solidity within the associated smart contract. Although, in general, these asserts can be scattered throughout the contract code, in our benchmark we keep the definitions of the properties separated from the contracts, in order to automatize the verification of multiple properties on multiple version of the contract. Accordingly, we provide two ways to write a property:
  - as a function that is added to the contract. This function may assert invariants on the contract state, and may call other contract methods as a property wrapper.
  - as a set of fragments of ghost code that are injected in the contract methods.

In this way, whoever extends the benchmark can write these properties without affecting the behaviour of the original contract, so that the instrumented contract satisfies the considered properties if and only if the original one does. In practice, this can be achieved by preventing ghost code from writing the state of the original contract and from changing its control flow except to signal the violation of the desired property. Future versions of the toolchain will give warnings when detecting potential discrepancies.

- for Certora, we write properties in the Certora Verification Language (CVL) [9]. The syntax of CVL extends Solidity with a set of meta-programming primitives that allow to express complex contract properties. We encode a contract property as a CVL *invariant* when the property involves facts about the state of the smart contract that should be true in any execution, while we encode it as a *rule* when the property concerns the expected behavior of calling one or more contract methods. In general, a rule is a sequence of commands that describe an execution trace of the contract, together with preconditions (`require`) and postconditions. There are two kinds of postconditions: `assert`, which *must* hold for any trace, and `satisfy`, which *can* be satisfied (i.e., it is possible to find a trace that makes them true). Unlike SolCMC, in Certora ghost code can be encoded directly within the properties, without altering the contract being verified.

Scoring the results of the tools

After constructing the verification tasks, the toolchain runs SolCMC (locally) and Certora (remotely) on each of them. The execution outcome on a verification task is summarised and scored according to the schema reported in Table 1. The overall design goal is that a tool that does nothing will have a null score, a tool that provides correct answers when verifying or detecting violation of the properties will have a positive score, and a tool that tricks the user into believing false results (e.g., claiming that a property holds when it is not the case, or viceversa) will have a negative score. We assign a null score in three cases: when the property is not expressible in the tool, when the tool fails to provide an output (e.g., because of aborts, timeouts, or memory exhaustion), and when the tool does not provide a definite answer about the validity of a property. The ratio for assigning the same score here is that it would be easy to make the property expressible by a tool that always diverges.

Our viewpoint is that tools are aimed at certifying that desirable properties are satisfied by a given Solidity implementation. Therefore, our scoring schema privileges soundness over completeness, as a false positive may create much bigger problems to users, as they will be convinced that their contract satisfies a property that in practice does not hold, while a false negative will only make the user doubt of the correctness of the contract.

The basis of our scoring schema is standard: we have two cases (P/N) depending on whether the property in the verification task is satisfied or not, and two cases (T/F) depending on whether the tool answers correctly to the task or not. We slightly deviate from this standard classification, in that we additionally classify the outputs of a tool as *strong* claims (e.g., “the property holds”, “the property is violated”) or *weak* claims (e.g., “the property *might* hold”, “the property *might* be violated”). More specifically, we use the following criteria to distinguish between weak and strong claims of the tools at hand:

- in SolCMC, when verification terminates, the output has one of the following forms:
  - “Assertion violation check is safe!”. We consider this as a strong claim that the tool has verified the property, hence we classify the output as a P!
  - “Assertion violation happens here”. In this case, the tool outputs the line of code where the asserted property is violated, and shows a sequence of method calls that lead to the violation. Hence, we consider this as a strong claim, and classify it as an N!

Table 1 Scoring schema for Solidity verification tools.

Result	Points	Description
ND	0	Property not expressible in the tool
UNK	0	Timeout / Memory exhaustion
TN!	2	Property violated, tool claims violation
TN	1	Property violated, tool conjectures violation
FN!	-8	Property holds, tool claims violation
FN	0	Property holds, tool conjectures violation
TP!	2	Property holds, tool claims correctness
TP	1	Property holds, tool conjectures correctness
FP!	-16	Property violated, tool claims correctness
FP	-1	Property violated, tool conjectures correctness

  - "Assertion violation might happen here". Here the tool has not been able to verify neither the violation, nor the correctness of the property, and (to stay on the safe side) it states that a violation is possible. We classify this output as an N.
- in Certora the classification depends both on the tool output and how the property is modelled in CVL. For an `assert`, the output is classified as P! when Certora returns ok, and N when it rejects the property. We do not put the "!" in the negative case as Certora may reject the property because of an unreachable counterexample, while we put the "!" in the positive case because it has been able to prove that no state (reachable or not) leads to a violation. The case `satisfy` is dealt with dually: we classify as P when Certora returns ok, and N! when it rejects the property.[2] Since our benchmark only admits rules that do not use both `assert` and `satisfy`, this criterion is always applicable.

The scoring schema is displayed in Table 1. Coherently with our design choices, we assign FP! the lowest score, because the tool is falsely claiming that a desirable property is true. Strong false negatives FN! (i.e., false alarms) are considered "half as dangerous" than false positives. False weak accepts (FP) have only a mildly negative score, since we treat them as mere conjectures: in fact, when conjecturing a result, the tool is just conveying the fact that it is not convinced of the truth of the opposite result. The asymmetry between FP! and FN! is mimicked on weak judgements by assigning false weak rejects (FN) a null score.

4 Evaluation: SolCMC *vs.* Certora

We discuss in this section what we have learnt by using SolCMC and Certora in the design and application of our benchmark. Here we focus on completeness, soundness and expressiveness of the two tools: their scoring as per Table 1 is presented at the end of the section. As a disclaimer, we note that our evaluation is based on the current versions of the tools[3]. Since they are moving targets, with multiple updates released during the writing of this paper, it is likely that some of the weaknesses discussed below may be fixed in future releases.

Completeness

SolCMC and Certora share some sources of incompleteness, i.e. properties that are true but that the provers do not manage to prove. This is the case e.g. for contracts containing external calls, i.e. calls from the analysed contract to another account [32] (e.g., the Deposit/ERC20

[2] It is possible to limit the set of considered starting states by refining the implementation with a set of invariants. We do this in a best effort manner.
[3] Versions: solc v0.8.24, Eldarica v2.0.9, Z3 v4.12.2, certora-cli v6.3.1

use case). By default, called contracts are considered untrusted by SolCMC and Certora, and accordingly these tools over-approximate their behaviour (even when their code is known). This basically makes the provers fail to verify any property that depends on the behaviour of the called contract, so leading to false negatives. For example, the `assert` in Listing 3 always passes, but SolCMC detects a possible violation (the same happens with Certora). SolCMC has an option to change the default behaviour by considering external calls to be trusted, but a known drawback of this option is a substantial computational overhead. For instance, checking the invalidity of the assertion `c.n()==0` in the contract `C2` of Listing 3 takes ~ 8 minutes using the Z3 solver in our experimental setting, despite the contract being just a few lines of code (by contrast, it takes a few seconds with the untrusted option). In general, even with the default option about untrusted calls, non-termination is not uncommon (see Table 2). This is a general problem related to Z3, which can be easily misled to divergence even with apparently harmless sets of constraints. Even in real-world use cases not specifically crafted to make verification burdensome, computation times sometimes occur to explode unexpectedly.

Listing 3 SolCMC: untrusted external calls.

```
contract C1 {
  uint n;
  constructor() { n=0; }

  function set() external { n=1; }

  // n could be either 0 or 1
}
```

```
contract C2 {
  C1 public c;
  constructor() { c=new C1(); }

  function inv() public view {
    assert(c.n()<=1);
  }
}
```

Many desirable properties of accounts, like e.g. that the balance is updated according to certain rules, are often broken on *contract* accounts. For instance, consider a contract with a method that allows the sender to withdraw funds, and the property "after a successful call to `withdraw`, the balance of `msg.sender` has increased". This property may be violated when the sender is a contract account, which fallbacks on the `withdraw` by giving away all its balance. While properties of this kind do not hold, in general, for contract accounts, they are expected to hold for externally-owned accounts (EOAs), which do not have code (e.g., `withdraw-sender-rcv-EOA` in the bank use case). In general, properties of EOAs are not even expressible, since it is not possible to discriminate EOAs from contract accounts (see the discussion below about expressiveness). A property about EOAs is expressible only if the account under scrutiny is the `msg.sender`. In this case, it is possible to tell that the account is an EOA by comparing it to `tx.origin` (the transaction originator): namely, the two addresses are equal iff `msg.sender` is an EOA. The property above can then be refined as "if `msg.sender` is an EOA, then after a call to `withdraw`, the balance of `msg.sender` has increased". Both SolCMC and Certora fail however to verify that the amended property is satisfied. No alternative encodings of EOAs seem to exist that allow the provers to successfully verify non-trivial properties about them.

Further cases of incompleteness include map invariants (e.g., the sum of a map is preserved), which are unlikely to be proved by both tools, and the over-approximation of the possible environments. E.g., Certora includes the contract in the approximation for `msg.sender`, even if the contract has no calls (e.g., `deposit-contract-balance` in the bank).

Listing 4 Unsoundness in SolCMC: *selfdestruct*.

```
contract CallWrapper is
    ReentrancyGuard {
  function callwrap(address called)
       public nonReentrant {
    called.call("");
  }
  ...
}
```

```
// SolCMC invariant
function inv(address a) public {
  uint b = address(this).balance;
  callwrap(a);

  // contract balance is preserved
  assert(b==address(this).balance);
}
```

Soundness

Analysing the results of our benchmark, we have spotted a few sources of unsoundness (i.e., the property does not hold but the tool falsely claims it is true) for both SolCMC and Certora. In SolCMC, false positives may happen when reasoning about the contract balance in case of external calls and reentrancy guards, as shown in Listing 4 (see the `bal` property in the CallWrapper use case). The contract `CallWrapper` has a single method, which performs a low-level call to an arbitrary address; the `nonReentrant` modifier by OpenZeppelin ensures that this call is non-reentrant. Now, consider the property: "the contract balance is preserved by `callwrap`". Apparently, it might seem to hold, because the contract has no `payable` nor `receive` methods. However, there are other asynchronous events that can make the contract balance increase: e.g., it can receive ETH from a *coinbase* transaction (i.e., the first transaction in a block, which collects the block reward), or from a *selfdestruct* (i.e., an action performed by a contract to destroy itself and transfer the remaining ETH to another account) [31]. Therefore, the property does *not* hold, since the address called by `callwrap` can be a contract that triggers a *selfdestruct*. SolCMC here produces a false positive, claiming that the invariant `inv` is always satisfied. We conjecture that this output derives from an under-approximation of SolCMC, which believes that the absence of reentrant methods implies that the call cannot affect the contract state, including the balance. Note instead that, when removing the `nonReentrant` modifier, SolCMC correctly detects that the invariant may be violated. Certora instead correctly classifies the property as false, but it produces a false positive on an extension of `CallWrapper` with a variable `s` that can be updated by the method `set`, and the property "`s` is preserved by `callwrap`" (see Listing 5 and the `stor` property in the CallWrapper use case). This property is false, since the account called by `callwrap` can perform a reentrant call to `set`. Certora fails to understand that a call to an address may lead to additional code execution, including a further call to one of the methods of the contract. Hence, Certora claims that the property holds, hereby being unsound [13].

Listing 5 Unsoundness in Certora: untracked reentrant calls.

```
contract CallWrapper {
  uint s;
  ...
  function set(uint snew) public {
    s = snew;
  }
}
```

```
rule P(address a) {
  env e;
  uint s0 = currentContract.s;
  callwrap(e, a);
  uint s1 = currentContract.s;
  assert s1 == s0;
}
```

Certora has some further documented under-approximations that may lead to unsoundness [11, 12]. When dealing with invariants, Certora checks that the invariant is preserved after the execution of each contract method, neglecting the effect of *selfdestruct* or *coinbase*

transactions. These transactions may increase the ETH balance of the analysed contract, which has no way of preventing this unexpected incoming ETH: therefore, if the invariant depends on the contract balance, it may be broken at any time. As an example, consider the contract `DoNothing` in Listing 6 , which just saves its initial balance in a variable `bal0`, and does nothing afterwards. The associated CVL property is an invariant checking that the contract balance is always equal to the stored initial balance. Certora claims that the invariant holds, since it holds at creation and it is preserved after the execution of each method (`balanceOf`). This is unsound, since e.g. a *coinbase* transaction can send ETH to the contract, making the actual balance exceed `bal0`.

Listing 6 Unsoundness in Certora: untracked *selfdestruct* and *coinbase* transactions.

```
contract DoNothing {
  uint bal0;
  constructor() {
    bal0 = address(this).balance;
  }
  function balanceOf(address a)
      public view returns (uint) {
    return a.balance;
  }
}
```

```
// Certora invariant specification

invariant inv()
  balanceOf(currentContract)
  ==
  currentContract.bal0;
```

Besides the artificial example in Listing 5, false positives may occur in real-world use cases when reasoning about the state after a low-level call. For instance, in our benchmark this is the case for a simple bank contract allowing users to deposit and withdraw ETH, and the property requiring that after a successful `withdraw`, the balance entry of the sender is decreased of the right amount. Apart from these glitches, both SolCMC Certora perform comparably well regarding false positives on our benchmark (see Table 2).

Expressiveness limitations

One of the main categories of properties that cannot be encoded in SolCMC are liveness properties (e.g., `wd-fin-before` in the vault use case). For a minimal example, consider Listing 7 and the property "`foo` never reverts". Intuitively, we can call `foo` inside of a `try-catch` statement, making sure that the method does not revert by checking that the `catch` is not reachable. However, this encoding is unsound, since the invariant is satisfied by some implementations of `foo` that actually revert. Specifically, this is the case of implementations of `foo` that never revert when the method is called by the contract itself, like the one in Listing 7, left. The invariant passes, but if foo is called from any account different from `Liveness`, then `foo` reverts. This is not the only way to trick SolCMC into verifying a false liveness property, as any assumption made by having the contract call itself can be used (e.g., reentrancy). In general, when expressing a property in SolCMC, any external call made in the invariant does not faithfully capture the intended property, as it does not model a call made by an arbitrary user, but only by the contract itself. The same problem exists when we use a low-level call instead of an external call within a `try-catch`. Attempting to encode the liveness property as the success of the invariant itself does not work either: any command in the body of the invariant cannot ensure that a certain line of it is always reached.

Even restricting to safety, expressing properties that involve two or more method calls in sequence is tricky, and in particular it cannot be done by using invariants, only. E.g., the invariant in Listing 8 (right) is a seemingly reasonable (but unsound) encoding of the property "`bar` cannot be called twice in a row" (see also the property `wd-twice` in the vault).

Listing 7 Completeness in SolCMC: a wrong encoding of a liveness property.

```
contract Liveness {
  function foo() {
    require(msg.sender==
            address(this));
  }
  ...
}
```

```
// wrong encoding of "foo succeeds"

function inv() public {
  try this.foo() {} catch {
    assert(false);
  }
}
```

Consider e.g. the implementation of `bar` in the contract `Sequence`: here, two consecutive calls to `bar` are possible whenever the callers are distinct: hence, the property is violated. Notice instead that the invariant is satisfied, because the sender of both calls to `bar` is the same, since coincides with the sender of `inv`. A sound encoding is still possible, but at the cost of instrumenting the contract with ghost code, which although supported by our toolchain, is a complex and error-prone operation in general. By contrast, Certora can express smoothly this kind of properties as CVL rules, whenever the number of calls in the sequence is fixed. Properties involving *unbounded* sequences are instead not expressible even in Certora (see e.g. the property `always-wd-all-many` in the tokenless bank use case).

Listing 8 Completeness in SolCMC: multiple sequential calls.

```
contract Sequence {
  address last;
  function foo() { ... }
  function bar() {
    require(msg.sender!=last);
    last = msg.sender;
  }
  ...
}
```

```
function inv() public {
  bar();
  bar();
  assert(false);
}
```

The property specification language supported by Certora is quite powerful, allowing us to express most of the properties in our benchmark. Still, there are interesting classes of properties that cannot be expressed. This is the case e.g. for properties of the form "for all reachable states, some user can do something that eventually produces some desirable effect". For example, consider the `Deposit` contract in Listing 9. The contract allows anyone to withdraw any fraction of its balance through the method `pay`, unless the variable `frozen` is true. Now, `frozen` is controlled by the contract owner through the method `freeze`: hence, the owner at any point can freeze the contract balance, preventing anyone from withdrawing. A desirable property contracts, in general, is that the funds stored in the contract cannot be frozen forever, a property often referred to as *liquidity* [33, 5, 27]. A tentative formalization of the liquidity property is the CVL rule in Listing 9. The rule is satisfied if there exists some starting state such that, for all `sender` address and value `v`, `sender` can fire a transaction `pay(v)` that increases their balance by `v`. This however is not a correct way to encode our liquidity property: indeed, Certora says that the property is satisfied, since there *exists* a trace that makes the condition in the `satisfy` statement true (this is the trace where the owner has not set `frozen`). Note that an alternative formalization where the `satisfy` is replaced by an `assert` does not work too. In this case, we would require that, for all reachable states, a transaction `pay(v)` is never reverted, *for all* choices of the amount `v`. Certora would correctly state that the property is false, because there are some values `v` that make the transaction fail (e.g., when `v` exceeds the contract balance). Although in this

simple case it would be easy to fix the property by requiring that the transaction is not reverted for all values `v` less than the contract balance and when `frozen` is false, in general for liquidity we would like to know if there *exist* parameters that make the desirable property true, which is not expressible in CVL.

Listing 9 Expressiveness of Certora: along all paths, eventually.

```
contract Deposit {
  address owner;
  bool frozen;
  constructor () payable {
    owner = msg.sender;
  }
  function freeze() public {
    require (msg.sender == owner);
    frozen = true;
  }
  function pay(uint v) public {
    require(!frozen);
    (bool succ,) = msg.sender.call{
        value: v}("");
    require(succ);
  }
}
```

```
// Certora rule specification

rule P(address sender, uint v) {
  // sender initial balance
  mathint b0 = bal(sender);
  env e;

  require e.msg.sender == sender;

  pay(e, v);

  // sender balance after pay(v)
  mathint b1 = bal(sender);

  // looking for a positive example
  satisfy(b1 == b0 + v);
}
```

Another category of properties that are not easily expressible, are those that reason about transfers of funds from the contract. For example, consider the property "the method `pay` calls the sender, transferring 10 wei from the contract to it", which is trivially satisfied by the contract in Listing 10 (see also `arbitrate-send` in the escrow use case). It might seem reasonable to express this property as a simple check on the balance of the contract and of the sender, as the invariant in Listing 10 (right). This however would not be correct: in fact, a contract that sends 10 wei to a middle man who forwards the sum to the sender would satisfy the invariant but violate the property. Indeed, we do not believe that properties of this kind are expressible in SolCMC. In CVL instead we can express a property very similar to the above, in which the required called contract is not the sender, but a specific account. This is possible with the use of hooks (this would require using hooks). It is unclear whether the exact property above is expressible in Certora.

As mentioned before, neither SolCMC nor Certora can, in general, express properties that are specific to EOAs. This derives from the fact that EOAs are not always discernible from contract accounts [28]. Other classes of properties that seem beyond the expressiveness boundaries of existing Solidity verification tools are those about game-theoretic interactions between users and adversaries, and fairness and probabilistic properties (see, respectively, the properties `rkey-no-wd` and `okey-rkey-private-wd` in the vault use case).

Scoring

We display in Table 2 the results obtained by running SolCMC and Certora on the verification tasks in our benchmark. As expected, Certora is able to express more properties than SolCMC; despite that, SolCMC's score is close to Certora's, which is penalized by its seemingly better behaviour with respect to soundness, the causes of which are exactly those discussed before in Listings 5 and 6. Regarding the two CHC solvers used by SolCMC, we note that Z3 has several UNK entries more than Eldarica due to its propensity to timeout. This explains the difference in score between Z3 and Eldarica: the latter performs worse because of its

Listing 10 Expressiveness of SolCMC: a wrong attempt of reasoning about received ETH.

```
contract Deposit {
  // pay 10 wei to the sender
  function pay() public {
    (bool succ,) = msg.sender.call
                     {value: 10}("");
    require(succ);
  }
}
```

```
function invariant() public {
  uint s0 = msg.sender.balance;
  uint c0 = address(this).balance;

  pay();

  uint s1 = msg.sender.balance;
  uint c1 = address(this).balance;
  assert(s1==s0+10 && c1==c0-10);
}
```

tendency to terminate with the wrong answer (getting -16 for the FP!), whereas Z3 just diverges (getting 0 for the UNK). Besides that, we do not note significant differences between these two CHC solvers. We highlight the absence of weak false positives (FP) in our results: SolCMC never claims that a property holds unless it is sure of it, while Certora only does so when the properties are encoded using `satisfy` statements (which have not been used in our current use cases).

Table 2 Scoring of SolCMC and Certora (323 verification tasks).

	ND	UNK	TP(!)	TN(!)	FP(!)	FN(!)	Score
Certora	60	0	97(97)	94(0)	7(7)	65(0)	176
SolCMC (Z3)	153	22	86(86)	49(48)	2(2)	11(9)	165
SolCMC (Eldarica)	153	5	88(88)	54(52)	7(7)	16(10)	90

5 Conclusions

We have discussed an ongoing collaborative effort towards the construction of a benchmark for comparing formal verification tools for Solidity. Once completed, the benchmark will serve as a valuable resource for developers, providing guidance in choosing the appropriate tool based on project requirements and contract complexity. In the meanwhile, we have given a first qualitative evaluation of SolCMC and Certora by discussing their soundness, completeness and expressiveness limitations based on our experience on developing the benchmark.

Constructing the ground truth was perhaps the most difficult task we encountered while developing the benchmark. In particular, while it is often easy to tell that an intentionally bugged contract violates a property, ensuring that a property is satisfied is error-prone. Indeed, Solidity has a few semantical quirks that make manual reasoning about contracts quite burdensome (reentrancy, in particular, may be very tricky). For this reason, it could make sense to simplify the construction of the ground truth by reducing the number of verification tasks for each use case: for instance, we could provide just a contract version that satisfies all the desirable properties, and just one version witnessing the violation of each property. While we are not able to certify the correctness of our ground truth beyond any reasonable doubt, we expect that the open-source nature of our benchmark can foster the collaboration with the community of experts. To consolidate our ground truth (at least, for negative properties), we plan to add, in future versions of the benchmark, references to actual contracts in the Ethereum testnet that violate the property. In this regard, we note that the counterexamples outputted by SolCMC and Certora when they find a violation are

rarely informative about its causes. Incorporating other datasets of ground truth, like e.g. [3], into ours, would also allow us to extend the comparison between SolCMC and Certora, at least for the kind of specific contract weaknesses considered in these datasets.

In the choice of the use cases in our benchmark, one of our primary criteria was simplicity: this is clearly motivated by the need of manually constructing the ground truth. One meaningful criterion to extend the benchmark could be to find use cases that depend on corner cases of the semantics of Solidity, to test which approximations are made in these cases by the verification tools. Extending the benchmark with more complex use cases, including more convoluted ways to violate properties, and experimenting it on other verification tools beyond SolCMC and Certora are also viable directions for future work.

References

1 Leonardo Alt, Martin Blicha, Antti E. J. Hyvärinen, and Natasha Sharygina. Solcmc: Cav 2022 artifact. `https://github.com/leonardoalt/cav_2022_artifact/tree/main`, 2022.

2 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`.

3 Monika Di Angelo and Gernot Salzer. Consolidation of ground truth sets for weakness detection in smart contracts. In *Financial Cryptography Workshops*, volume 13953 of *LNCS*, pages 439–455. Springer, 2023. `doi:10.1007/978-3-031-48806-1_28`.

4 Shaun Azzopardi, Joshua Ellul, and Gordon J. Pace. Monitoring smart contracts: ContractLarva and open challenges beyond. In *Runtime Verification*, volume 11237 of *LNCS*, pages 113–137. Springer, 2018. `doi:10.1007/978-3-030-03769-7_8`.

5 Massimo Bartoletti, Stefano Lande, Maurizio Murgia, and Roberto Zunino. Verifying liquidity of recursive Bitcoin contracts. *Log. Methods Comput. Sci.*, 18(1), 2022. `doi:10.46298/LMCS-18(1:22)2022`.

6 Thomas Bernardi, Nurit Dor, Anastasia Fedotov, Shelly Grossman, Neil Immerman, Daniel Jackson, Alexander Nutz, Lior Oppenheim, Or Pistiner, Noam Rinetzky, Mooly Sagiv, Marcelo Taube, John A. Toman, and James R. Wilcox. WIP: Finding bugs automatically in smart contracts with parameterized invariants. `https://groups.csail.mit.edu/sdg/pubs/2020/sbc2020.pdf`, 2020.

7 Dirk Beyer. Competition on software verification and witness validation: SV-COMP 2023. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 13994 of *LNCS*, pages 495–522. Springer, 2023. `doi:10.1007/978-3-031-30820-8_29`.

8 N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation* (II), volume 9300 of *LNCS*, pages 24–51. Springer, 2015. `doi:10.1007/978-3-319-23534-9_2`.

9 Certora. The Certora Verification Language. `https://docs.certora.com/en/latest/docs/cvl/index.html`, 2022.

10 Certora. Formal verification of OpenZeppelin (may-june 2022). `https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/certora/reports/2022-05.pdf`, 2022.

11 Certora. Certora prover documentation: invariants. `https://docs.certora.com/en/latest/docs/cvl/invariants.html`, 2023.

12 Certora. Certora prover documentation: Prover approximations. `https://docs.certora.com/en/latest/docs/prover/approx/index.html`, 2023.

13 Certora report: CallWrapper. `https://prover.certora.com/output/95211/e265c818d176463cbaa6f53e4d0fe394?anonymousKey=7d67af6ba7eedc4f099a133a04f4d36953f377dc`, 2024.

14 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 *ICSE*, 2024. To appear.

15 Consensys. Write smart contract specifications using Scribble. https://consensys.io/diligence/scribble/, 2023.

16 E. De Angelis, F. Fioravanti, J. P. Gallagher, M. V. Hermenegildo, A. Pettorossi, and M. Proietti. Analysis and transformation of constrained Horn clauses for program verification. *Theory Pract. Log. Program.*, 22(6):974–1042, 2022. doi:10.1017/S1471068421000211.

17 L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

18 Bruno Dias, Naghmeh Ivaki, and Nuno Laranjeiro. An empirical evaluation of the effectiveness of smart contract verification tools. In *Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 17–26, 2021. doi:10.1109/PRDC53464.2021.00013.

19 Enterprise Ethereum Alliance. Smart contract weakness classification (SWC). https://swcregistry.io/, 2020.

20 Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15. IEEE/ACM, 2019. doi:10.1109/WETSEB.2019.00008.

21 Ikram Garfatta, Kais Klai, Walid Gaaloul, and Mohamed Graiet. A survey on formal verification for Solidity smart contracts. In *Australasian Computer Science Week Multiconference*, pages 3:1–3:10. ACM, 2021. doi:10.1145/3437378.3437879.

22 Hossein Hojjat and Philipp Rümmer. The ELDARICA Horn solver. *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–7, 2018. doi:10.23919/FMCAD.2018.8603013.

23 Nikolay Ivanov, Chenning Li, Qiben Yan, Zhiyuan Sun, Zhichao Cao, and Xiapu Luo. Security threat mitigation for smart contracts: A comprehensive survey. *ACM Comput. Surv.*, 55(14s):326:1–326:37, 2023. doi:10.1145/3593293.

24 Daniel Jackson, Chandrakana Nandi, and Mooly Sagiv. Certora technology white paper. https://docs.certora.com/en/latest/docs/whitepaper/index.html, 2022.

25 Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-based Model Checking for Recursive Programs. *Formal Methods in System Design*, pages 175–225, 2016.

26 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.

27 Cosimo Laneve. Liquidity analysis in resource-aware programming. *J. Log. Algebraic Methods Program.*, 135:100889, 2023. doi:10.1016/J.JLAMP.2023.100889.

28 OpenZeppelin. Utilities / address. https://docs.openzeppelin.com/contracts/4.x/api/utils#Address, 2024.

29 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.

30 Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. eThor: Practical and provably sound static analysis of Ethereum smart contracts. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 621–640. ACM, 2020. doi:10.1145/3372297.3417250.

31 The Solidity Authors. SMTChecker and formal verification: contract balance. https://docs.soliditylang.org/en/v0.8.24/smtchecker.html#contract-balance, 2023.

32 The Solidity Authors. SMTChecker and formal verification: untrusted calls and reentrancy. https://docs.soliditylang.org/en/latest/smtchecker.html#trusted-external-calls, 2023.

33 Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. Securify: Practical security analysis of smart contracts. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 67–82. ACM, 2018. doi:10.1145/3243734.3243780.

34 Scott Wesley, Maria Christakis, Jorge A. Navas, Richard J. Trefler, Valentin Wüstholz, 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`.

35 Scott Wesley and Valentin Wüstholz. Verify OpenZeppelin. `https://github.com/contract-ace/verify-openzeppelin`, 2022.

Towards Formally Specifying and Verifying Smart Contract Upgrades in Coq

Derek Sorensen
Department of Computer Science and Technology, University of Cambridge, UK

Abstract

Smart contract upgrades are costly from a verification perspective and can be a meaningful source of vulnerabilities when done incorrectly. Unfortunately, there is no established, formal framework through which one can reason about contracts as they undergo upgrades, though much work has been done to verify standalone smart contracts. Instead, one must repeat the full verification process for each contract upgrade, something which relies heavily on fallible intuition, can lead to unexpected vulnerabilities, and drives up the cost of formally verifying smart contracts. We propose a formal framework for contract upgrades in ConCert, a Coq-based smart contract verification tool. Central to this framework is our notion of a *contract morphism*, a theoretical tool which we introduce to formally encode structural relationships between smart contracts, and with which we can formally specify and verify an upgraded contract relative to its previous versions. We argue that ours is a natural framework for specifying and verifying contract upgrades, and hope to offer a first step towards rigorous, efficient specification and verification of contract upgrades.

2012 ACM Subject Classification Theory of computation → Program verification

Keywords and phrases smart contract verification, formal methods, interactive theorem prover, smart contract upgrades

Digital Object Identifier 10.4230/OASIcs.FMBC.2024.7

Supplementary Material *Software*: `https://github.com/differentialderek/FinCert`
archived at `swh:1:dir:9ab2f53114db2da1e2ad01b23f4213265efa9702`

1 Introduction

Faulty upgrades are a meaningful source of smart contract vulnerabilities. Costly attacks such as those on Uranium Finance (2021) [8], NowSwap (2021) [4], and Nomad (2022) [7, 9], totaling 241 million USD in lost assets, are a few of many examples of contracts attacked after an erroneous upgrade. Furthermore, because verifying software is time, labor, and resource intensive, it can be difficult to justify formally verifying software which may be upgraded quickly or frequently – a problem shared with other verified software, *e.g.* [16, 21]. Both of these factors limit the effectiveness of formal methods to address security issues in real-world software, inhibiting verification as business and security propositions [18].

What is needed is a practical and formal framework through which to specify and verify contract upgrades. As it stands we have no such framework apart from repeating the formal specification and verification process on a new contract version. Not only are upgrades costly from a verification perspective, as we have no good way of reusing much of the verification work on previous contract versions, but incorrect specifications are themselves a meaningful source of contract vulnerabilities [19]. Thus each time a specification is made from scratch we risk introducing errors of incorrect specification.

To mitigate these issues we introduce a formal framework for specifying and verifying contract upgrades, through which we can reuse formal specification and proof on previous contract versions. This framework relies on the notion of a *contract morphism*, a theoretical tool we introduce that formally encodes structural relationships between smart contracts, and with which we can specify and reason about the structure and behavior of an upgraded

© Derek Sorensen;
licensed under Creative Commons License CC-BY 4.0

contract relative to its previous versions. We argue that this is a natural framework for specifying and verifying contract upgrades, one which could decrease the cost of formally verifying contract upgrades as well as the risk of introducing vulnerabilities due to incorrect specification.

We proceed as follows. In §2, we survey related work. In §3, we introduce *contract morphisms* as a formal tool to specify and verify contract upgrades. In §4 we give two examples of formally specifying a contract upgrade with contract morphisms. In §5 we discuss formal verification with contract morphisms. We conclude in §6.

2 Related Work

In the realm of smart contracts there is limited formal work on formal reasoning about contract upgrades. Previous work [3, 6] proposes paradigm-shifting methods to either attach formal proofs to smart contracts and their upgrades, which are verified by the chain, or to trust a canonical third party to verify all contract upgrades before deployment. Unfortunately this work is likely impractical, as both solutions require substantial paradigm shifts or re-engineering of blockchain ecosystems. The latter also arguably contradicts the permissionless ethos of blockchain ecosystems by mandating a trusted third party.

In the context of software more generally, much work has gone into ensuring that software upgrades are carried out safely with formal methods [10, 12, 21]. Recent work has begun to address the issue of adapting formal proofs in a proof assistant to changes in software in order to lower the cost of formally verified software which may undergo regular upgrades [16]. This problem is complicated by the computable nature of proofs in proof assistants like Coq; chosen data types strongly influence the structure of proofs, making adaptation difficult [11]. A notable contribution to this work is Ringer *et al.*'s work on *proof repair* [14, 15], which seeks to relate a new program version to the old – by type equivalences or by comparing inductive structures – and thereby reuse previously-completed proofs on the updated code.

Drawing on this previous work, particularly Ringer *et al.*'s idea of reusing formal proofs by way of structural similarities between programs, our goal is to provide a framework for using formal methods to formally specify and verify smart contract upgrades. Contract morphisms (§3) will be our primary theoretical tool for specifying and verifying contract upgrades. Their purpose is to formally encode a structural relationship between smart contracts which can be used for both formal specification and proof reuse. With contract morphisms we address the problem of formal reasoning about contract upgrades, but in contrast to previous work on the subject our proposed framework does not require the paradigm-shifting reengineering of blockchain systems in order to be used.

Finally, we note that for smart contracts there is a distinction between contract upgrades and contract *upgradeability*. Some contracts come with a predefined logic to handle upgrades and avoid hard forks, the most popular of these on Ethereum being the Diamond framework [13]. However, they are complicated contracts as their specifications include the upgradeability functionality and governance, as well as the functionality of a given version of the contract. We will only consider upgrades via hard forks in this paper, leaving the question of rigorous formal specification and verification of upgradeable contracts to future work.

3 Contract Morphisms

In what follows we define *contract morphisms*, a theoretical tool which codifies formal relationships between smart contracts. In later sections we use them to formally specify and verify contract upgrades. We argue that this provides our desired formal framework.

3.1 Morphisms of Pure Functions

Before focusing on the specific case of smart contracts, we consider the more general case of programs formalized as pure functions. Take types A, A' and B, B', and two functions $p : A \to B$ and $q : A' \to B'$. A *morphism* from p to q is a or a pair of functions f_i and f_o which form a commutative square,

$$\begin{array}{ccc} A & \xrightarrow{f_i} & A' \\ {\scriptstyle p}\downarrow & /\!\!/ & \downarrow{\scriptstyle q} \\ B & \xrightarrow{f_o} & B' \end{array}$$

i.e. for which

$q \circ f_i = f_o \circ p.$

Together, we call f_i and f_o the morphism

$f : p \to q.$

Via f_i and f_o, the commutative square like the above maps inputs and outputs of p to inputs and outputs of q. If p and q are programs (in particular, pure functions), we can also interpret this as execution traces of p to execution traces of q, such that transforming the inputs of p into those of q with f_i, and then applying q is the same as applying p first and then transforming the outputs over f_o.

We can define composition of morphisms easily as the composition of commutative squares. That is, given functions p, q, and r, and morphisms

$f' : p \to q$ and $f'' : q \to r$,

we can define a morphism $f := f'' \circ f' : p \to r$ by the outer square of the following diagram,

$$\begin{array}{ccccc} A & \xrightarrow{f'_i} & A' & \xrightarrow{f''_i} & A'' \\ {\scriptstyle p}\downarrow & /\!\!/ & \downarrow{\scriptstyle q} & /\!\!/ & \downarrow{\scriptstyle r} \\ B & \xrightarrow{f'_o} & B' & \xrightarrow{f''_o} & B'' \end{array}$$

which is commutative if each of the inner squares are commutative. Note that composition is associative, assuming the underlying functions are associative, and that we have the obvious identity morphism $f_{\text{id}} : p \to p$ given by $f_i, f_o := id$,

$$\begin{array}{ccc} A & \xrightarrow{\text{id}} & A \\ {\scriptstyle p}\downarrow & /\!\!/ & \downarrow{\scriptstyle p} \\ B & \xrightarrow{\text{id}} & B \end{array}$$

which commutes trivially. Thus given a well-defined class of functions, which in our case will be smart contracts modeled in Coq by pure functions, we have a category on those functions with morphisms given by commutative squares on those pure functions.

In the coming sections, given a morphism $f : p \to q$, we might consider the case that q is an upgraded version of p. Because f relates execution traces of q to those of p, we will see this can be used to reason formally about q in terms of p, both in specification and verification.

3.2 Contract Morphisms in ConCert

In ConCert, a Coq-based tool for smart contract verification which models the execution semantics of third-generation blockchains [2] and features verified extraction to various blockchains [1], smart contracts are formalized with a `Contract` type as a pair of pure, stateful 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 `(init C)` and `(receive C)` are given as follows, where the types `Chain` and `ContractCallContext` are ConCert-specific types used to model the underlying blockchain and context.

Listing 1 Type signature of the `init` and `receive` functions, respectively, of a smart contract in ConCert.

```
init C : Chain → ContractCallContext → Setup → result State Error.

receive C : Chain → ContractCallContext → State → option Msg →
            result (State * list ActionBody) Error.
```

Now consider contracts `C1` and `C2`,

```
C1 :  Contract Setup1 Msg1 State1 Error1

C2 :  Contract Setup2 Msg2 State2 Error2.
```

We define a data type of *morphisms* between contracts `C1` and `C2`,

```
ContractMorphism C1 C2.
```

This data type consists firstly of four *component functions* between the contract types of `C1` and `C2` – the `Setup`, `Msg`, `State`, and `Error` types respectively.

- `setup_morph :  Setup1 -> Setup2`
- `msg_morph   :  Msg1   -> Msg2`
- `state_morph :  State1 -> State2`
- `error_morph :  Error1 -> Error2.`

We can use these component functions to make commutative squares like those we saw in §3.1 for each of the `init` and `receive` functions. For `init`, the horizontal arrows of the squares are given by the functions `mA_init` and `mB_init`. For `receive`, the horizontal arrows are given by the functions `mA_recv` and `mB_recv`. See Listing 2 for the definition of these functions in terms of the four component functions given above.

$$\begin{array}{ccc} A_{\texttt{init}} & \xrightarrow{\texttt{mA_init}} & A'_{\texttt{init}} \\ {\scriptstyle\texttt{init}}\downarrow & /\!/ & \downarrow{\scriptstyle\texttt{init}'} \\ B_{\texttt{init}} & \xrightarrow{\texttt{mB_init}} & B'_{\texttt{init}} \end{array} \qquad \begin{array}{ccc} A_{\texttt{recv}} & \xrightarrow{\texttt{mA_recv}} & A'_{\texttt{recv}} \\ {\scriptstyle\texttt{receive}}\downarrow & /\!/ & \downarrow{\scriptstyle\texttt{receive}'} \\ B_{\texttt{recv}} & \xrightarrow{\texttt{mB_recv}} & B'_{\texttt{recv}} \end{array}$$

The functions defined above give us squares, but to finish the definition of contract morphisms we need these squares to commute. Thus our definition includes two coherence conditions, one for the `init` square and one for the `receive` square, which are given as follows.

Listing 2 The functions which we use for the horizontal arrows of a pair of commutative squares `f_init : init C1 -> init C2` and `f_recv : receive C1 -> receive C2`, respectively, in the definition of a contract morphism.

```
(* functions to form a commutative square on init *)
mA_init :=
    fun (c : Chain) (ctx : ContractCallContext) (s : Setup) =>
    (c, ctx, setup_morph s).
mB_init := fun (res : result State Error) =>
    match res with
    | Ok init_st => Ok (state_morph init_st)
    | Err e => Err (error_morph e)
    end.

(* functions to form a commutative square on receive *)
mA_recv := fun (c : Chain) (ctx : ContractCallContext)
    (st : State) (op_msg : option Msg) =>
    (c, ctx, state_morph st, option_map msg_morph op_msg).
mB_recv := fun (res : result (State * list ActionBody) Error) =>
    match res with
    | Ok (init_st, nacts) => Ok (state_morph init_st, nacts)
    | Err e => Err (error_morph e)
    end.
```

```
(* The coherence condition that makes the init square commute *)
init_coherence: forall c ctx s,
(match (init C1 c ctx s) with
    | Ok init_st => Ok (state_morph init_st)
    | Err e => Err (error_morph e)
    end) =
(init C2 c ctx (setup_morph s)).

(* The coherence condition that makes the receive square commute *)
recv_coherence : forall c ctx st op_msg,
(match (receive C1 c ctx st op_msg) with
    | Ok (new_st, new_acts) => Ok (state_morph new_st, new_acts)
    | Err e => Err (error_morph e)
    end) =
(receive C2 c ctx (state_morph st) (option_map msg_morph op_msg)).
```

Thus a contract morphism

`m : ContractMorphism C1 C2`

is defined as a pair of commutative squares, each of which are morphisms between the respective `init` and `receive` functions of each contract. We give the formal definition of a contract morphism in Listing 3.

As the name *morphism* suggests, we should expect contract morphisms to behave like morphisms in a well-defined category. That is, we should have an associative composition operation on morphisms, and for every contract `C` should have an identity morphism

`id_C : ContractMorphism C C`

with which composition is trivial.

Listing 3 The formal definition of a contract morphism in ConCert, consisting of four component functions and two coherence conditions, which together give a pair of commutative squares.

```
Record ContractMorphism
    (C1 : Contract Setup1 Msg1 State1 Error1)
    (C2 : Contract Setup2 Msg2 State2 Error2) :=
    build_contract_morphism {
        (* the components of a morphism *)
        setup_morph : Setup1 → Setup2 ;
        msg_morph  : Msg1   → Msg2  ;
        state_morph : State1 → State2 ;
        error_morph : Error1 → Error2 ;
        (* coherence conditions *)
        init_coherence : forall c ctx s,
            result_functor state_morph error_morph (init C1 c ctx s) =
            init C2 c ctx (setup_morph s) ;
        recv_coherence : forall c ctx st op_msg,
            result_functor (fun '(st, l) ⇒ (state_morph st, l))
                error_morph
                (receive C1 c ctx st op_msg) =
            receive C2 c ctx (state_morph st)
                (option_map msg_morph op_msg) ;
}.
```

Indeed, this is the case. We can compose morphisms by composing the morphism component functions. We have two results,

`compose_init_coh` and `compose_recv_coh`,

which show that coherence of the composed morphism follows from the coherence conditions of each individual morphism. These results simply show that commutative squares compose, as we saw in §3.1, giving us a well-defined composition function `compose_cm`.

```
compose_cm : forall C1 C2 C3
    (g : ContractMorphism C2 C3) (f : ContractMorphism C1 C2) : ContractMorphism C1 C3.
```

We also have a proof that composition is associative, drawing on the associativity of component functions, and we have the obvious identity morphism, given by four identity component functions, such that composition with the identity is trivial.

```
Definition id_cm (C : Contract Setup Msg State Error) :
  ContractMorphism C C := {|
      (* components *)
      setup_morph := id ;
      msg_morph   := id ;
      state_morph := id ;
      error_morph := id ;
      (* coherence conditions *)
      init_coherence := init_coherence_id C ;
      recv_coherence := recv_coherence_id C ;
  |}.
```

This gives us a well-defined category **Contracts** of smart contracts, with objects given by the `Contract` type and morphisms given by the `ContractMorphism` type.

Note that in many categories, *e.g.* the categories of sets, topological spaces, or groups, morphisms are structure-preserving functions. So too for us. The existence of a morphism

```
f : ContractMorphism C1 C2
```

indicates a structural and mathematical relationship between contracts `C1` and `C2`, in particular relating their execution traces via the four component morphisms. As we will see, this relationship can be exploited to prove theorems about one contract in terms of another contract, something which we will do here in the case of contract upgrades and upgradeability.

In many categories there are also different classes of morphisms, including injections (embeddings, monomorphisms), surjections (quotients, epimorphisms), and isomorphisms. Injections, or embeddings, typically preserve the structure of the domain faithfully within the codomain, essentially identifying a copy of the domain within the codomain. Surjections typically represent a compression of some kind, and the information lost in the compression can frequently be described by a kernel object. As we will see, we also have injective and surjective contract morphisms, which are given when the four component functions are, respectively, injective or surjective, and which follow analogous intuitions.

4 Morphisms to Formally Specify and Verify Contract Upgrades

Our goal now is to use contract morphisms as a tool to formally specify and verify contract upgrades in ConCert. Consider a contract upgrade from the perspective of a formal specification. Contracts are usually upgraded with a goal that relates the new to the previous contract version, whether it be to patch a bug, add functionality, or improve contract features. Thus the new specification relates to the old – it should eliminate a vulnerability but preserve all other functionality, be backwards compatible while adding functionality, or make improvements such as greater gas-efficiency without deviating from the behavior of the previous contract version. Of course, in practice an upgraded contract is not formally specified in relation to an older version, but rather by altering the old specification into the new, or simply starting from scratch and writing a new specification by hand. As discussed in §1, this can be a source of vulnerabilities.

In this section, we will formally specify contract upgrades in two examples using contract morphisms. The advantage of using morphisms is that we are able to clearly articulate the intent of an upgrade in the formal specification by way of a morphism in such a way that formal verification consists of producing a morphism between the updated contract implementation and a previous version which meets the required specification.

▶ **Example 1** (Swap Contract Upgrade)**.** Consider a smart contract `C1` that prices and executes trades, *e.g.* a decentralized exchange (DEX) or an automated market maker (AMM) [22]. Suppose that we wish to upgrade `C1` to a contract `C2` so that it calculates trades at higher precision by a factor of ten, meaning that the internal token balances in storage have one more decimal place, and the trade calculation is able to calculate at one decimal place greater in precision. Then in ConCert our contract `C1` will have a storage type which keeps track of internal token balances, exposed by a function `get_bal`.

```
Context { storage : Type } { get_bal : storage → N }.
```

It will also have a `TRADE` entrypoint which accepts a payload of some type, `trade_data`, characterized by an entrypoint type, `entrypoint`, and an associated typeclass, `Msg_Spec`.

Listing 4 We assume an entrypoit type `entrypoint`, characterized by a typeclass `Msg_Spec`, which includes a trade function `trade`.

```
Class Msg_Spec (T : Type) := {
    (* the trade entrypoint *)
    trade : trade_data → T ;
    (* for any other entrypoint types *)
    other : other_entrypoint → option T ;
}.

(* We assume an entrypoint conforming to Msg_Spec *)
Context { entrypoint : Type } `{ e_msg : Msg_Spec entrypoint }.
```

Now assume that `C1` has some internal function `calculate_trade` that it uses to calculate how many tokens will be traded out for a given contract call to the `TRADE` entrypoint. The trade quantity, internal token balances, and the `calculate_trade` function will all be accurate up to some decimal place, commonly 9 in the wild, formalized in the following specification, `spec_trade`, of `C1`.

Listing 5 The formalized proposition that `C1` uses `calculate_trade` to price trades.

```
(* the specification of C1's trading functionality with regards to the
    calculate_trade function *)
Definition spec_trade : Prop :=
    forall cstate chain ctx trade_data cstate' acts,
    (* for any successful call to C1's trade entrypoint, *)
    receive C1 chain ctx cstate (Some (trade trade_data)) =
    Ok(cstate', acts) →
    (* the balance in storage updates according to the
        calculate_trade function *)
    get_bal cstate' =
    get_bal cstate + calculate_trade (trade_qty trade_data).
```

The property of Listing 5, `spec_trade`, is a specification with regards to which `C1` is assumed to be correct.

Now we wish to upgrade `C1` to a new contract `C2` such that `C2` calculates trades and keeps balances at one decimal place higher of accuracy. We will first have a specification for `C2` which is analogous to `spec_trade` in Listing 5, which says that `C2` uses some new function, `calc_trade_precise`, to calculate its trades.

Listing 6 The formalized proposition that `C2` uses `calculate_trade_precise` to price trades.

```
(* The specification of C2's trading functionality with regards to the
    calculate_trade_precise function. This is analogous to spec_trade *)
Definition spec_trade_precise : Prop :=
    forall cstate chain ctx trade_data cstate' acts,
    (* for a successful call to C2's trade entrypoint, *)
    receive C2 chain ctx cstate (Some (trade trade_data)) = Ok (cstate', acts) →
    (* the balance in storage updates according to the
        calculate_trade_precise function *)
    get_bal cstate' =
    get_bal cstate +
    calculate_trade_precise (trade_qty trade_data).
```

Our goal now is to use a contract morphism to complete the formal specification of `C2` in terms of `C1`. Our specification is this: A correct implementation of the upgraded contract `C2` must satisfy `spec_trade_precise` and be accompanied by a contract morphism

```
f :  ContractMorphism C2 C1
```

with the following five properties, stated formally in Listing 7:

1. `msg_morph f` rounds down the precision of messages to `trade` by a factor of `10`
2. `msg_morph f` is the identity morphism on all messages aside from messages to `trade`
3. `state_morph f` rounds down on the balances kept in storage exposed by `get_bal`
4. `error_morph f` and `setup_morph f` are the respective identity functions

Listing 7 The formal specification of the upgrade from `C1` to `C2`.

```
(* FORMAL SPECIFICATION:
    An upgrade C2 must admit a morphism
    f : ContractMorphism C2 C1
    with the following properties: *)

(* 1. msg_morph f rounds trades down when it maps inputs of the receive function *)
Definition f_recv_input_rounds_down
    (f : ContractMorphism C2 C1) : Prop :=
    forall t', exists t,
    (msg_morph C2 C1 f) (trade t') = trade t ∧
    trade_qty t = (trade_qty t') / 10.

(* 2. msg_morph f only affects the trade entrypoint *)
Definition f_recv_input_other_equal
    (f : ContractMorphism C2 C1) : Prop :=
    forall msg o,
    (* for calls to all other entrypoints, *)
    msg = other o →
    (* f is the identity *)
    option_map (msg_morph C2 C1 f) (other o) = other o.

(* 3. state_morph f rounds down on the storage *)
Definition f_state_morph (f : ContractMorphism C2 C1) : Prop :=
    forall st, get_bal (state_morph C2 C1 f st) = (get_bal st) / 10.

(* 4. error_morph f and setup_morph f are the identity functions *)
Definition f_recv_output_err (f : ContractMorphism C2 C1) : Prop :=
    (error_morph C2 C1 f) = id.

Definition f_init_id (f : ContractMorphism C2 C1) : Prop :=
    (setup_morph C2 C1 f) = id.
```

The meaning of a morphism `f` satisfying the above conditions, as a specification, is in the *coherence conditions* of `f`. We know that every possible execution trace of `C2` has a corresponding execution trace in `C1`, and we know that the input messages are identical except that `C2` accepts trades at a higher level of precision. The coherence conditions also tell us that the state of `C2` is always related to the analogous state of `C1`, expressed in the function `state_morph`. With regards to the trading functionality of our new contract `C2`, we know that the balance kept in the storage of `C2`, which is affected by trades, will always be identical to the analogous balance of `C1` after rounding down, which we can formally prove.

Listing 8 All reachable states of `C2` round down to their corresponding states in `C1`.

```
Theorem rounding_down_invariant bstate caddr
    (trace : ChainTrace empty_state bstate):
    (* Forall reachable states with contract at caddr, *)
    env_contracts bstate caddr = Some (C2 : WeakContract) →
    (* cstate is the state of the contract AND *)
    exists (cstate' cstate : storage),
    contract_state bstate caddr = Some cstate' ∧
    (* cstate is contract-reachable for C1 AND *)
    cstate_reachable C1 cstate ∧
    (* such that for cstate, the state of C1 in bstate,
       the balance in cstate is rounded-down from the
       balance of cstate' *)
    get_bal cstate = (get_bal cstate') / 10.
```

Most importantly, `f` guarantees a relationship between the trading functionality of `C2` and that of `C1`: `C2` emulates the exact same trading behavior as `C1` after rounding down one decimal place in precision. This means that `C2` does not introduce any novel vulnerabilities relating to trades and balances not extant to `C1`. In particular, a proof of this fact would have prevented the attacks on Uranium Finance [8], NowSwap [4], and Nomad [7].

Moving on, note that `f` of Example 1 was directed from `C2` to `C1`. The coherence conditions of `f` forced all execution traces of `C2` to conform to a pattern set by `C1`, which is precisely what lets us make the claim that we haven't introduced any new behaviors regarding trading functionality to `C2` aside from the increase in precision. Morphisms directed in the opposite direction can also be used in specification. Rather than classifying all possible execution traces of the upgrade, in this case a morphism proves that certain desired behavior exists within the contract. We illustrate with an example of specifying backwards compatibility.

▶ **Example 2** (Backwards Compatibility)**.** Consider contracts `C1` and `C2`, where `C2` is again an upgrade of `C1`, and suppose that we wish to show that `C2` is backwards compatible with `C1`. The intent of this upgrade is that the full functionality of `C1` be present within `C2`. We show this by embedding `C1` into `C2` via an injective contract morphism.

We illustate with a simple example of a counter contract `C1` which keeps some `n : N` in storage and has one entrypoint `incr` that increments the natural number in storage by `1`. `C1` is upgraded to `C2`, which in addition to an entrypoint to increment the natural number in storage also includes a `decr` entrypoint to decrement the natural number in storage by `1`.

Listing 9 The entrypoint types of `C1` and `C2`, respectively.

```
Inductive entrypoint1 := | incr (u : unit).
Inductive entrypoint2 := | incr' (u : unit) | decr (u : unit).
```

We prove that `C2` is backwards compatible with `C1` by defining a contract morphism

`f : ContractMorphism C1 C2`

with the following component functions.

```
Definition msg_morph (e : entrypoint1) : entrypoint2 :=
    match e with | incr _ ⇒ incr' tt end.
Definition setup_morph : setup → setup  := id.
Definition state_morph : storage → storage := id.
Definition error_morph : error → error  := id.
```

These component functions do the obvious thing – send calls to the increment entrypoint of `C1` to the increment entrypoint of `C2` with the same payload, and do nothing otherwise. And `f` is an embedding since each of its component functions are manifestly injective, which we can formally prove.

```
Lemma f_is_embedding : is_inj_cm f.
```

Again, the meaning of `f` as a specification is in its coherence conditions. Any reachable state of `C1` necessarily has an analagous reachable state of `C2` which is fully structure preserving: if we were to only use the functionality of `C2` which it inherits from `C1`, we would get identical contract behavior to `C1`. We have a formal proof of this result.

Listing 10 `C2` is backwards compatible with `C1` via the embedding `f`.

```
Theorem injection_invariant bstate caddr
    (trace : ChainTrace empty_state bstate):
    env_contracts bstate caddr = Some (C1 : WeakContract) →
    (* Forall reachable states cstate of C1,
       there's a corresponding reachable state
       cstate' of C2, related by the injection *)
    exists (cstate' cstate : storage),
    contract_state bstate caddr = Some cstate ∧
    (* cstate' is a contract-reachable state of C2 *)
    cstate_reachable C2 cstate' ∧
    (* .. equal to cstate *)
    cstate' = cstate.
```

This is a toy example, but in practice specifying a new contract which is backwards compatible to the old in this strong sense may not be straightforward. Via contract embeddings, contract morphisms give us a way of formally specifying and verifying backwards compatibility.

5 Further Applications of Morphisms in Formal Verification

Contract morphisms establish a relationship between contracts which makes them suitable for specifying and verifying upgrades. For that same reason, contract morphisms may also have applications in proof reuse, or proof *transport*, more generally. The special case of contract *isomorphism* may also provide a stronger relationship between formal specification and proof on the associated contracts.

5.1 Hoare Properties and Contract Morphisms

First we consider properties that *transport* over a morphism, in particular those that we can pull back over a morphism. Hoare properties are a particularly strong example: they relate pre-conditions to post-conditions, which is relevant to morphisms because morphisms relate inputs and outputs of contract executions. As contracts are formalized in ConCert, constraints on on inputs amount to pre-conditions, and constraints on outputs amount to post-conditions. Thus for contracts `C1` and `C2` and a morphism `f : ContractMorphism C1 C2`, we might expect to be able to transport Hoare properties of one contract over `f` to the other.

Indeed, any Hoare property proved for `C2` will always have an analogous result on `C1`, mediated by `f`. We proved this in two results which relate all reachable states of `C1` to those of `C2`, and those of `C2` to those of `C1`, via the `state_morph` component of `f`. These results, `left_cm_induction` and `right_cm_induction`, are collectively called morphism induction, as

they allow us to induct along the execution trace of one contract in relation to that of another. In particular, morphism induction says that properties of the state of `C2` which are invariant over `state_morph` must hold for all states of `C1`.

As a toy example of this relationship, suppose that we can prove that if a certain boolean in the storage of `C2` is set at `true`, a given entrypoint `e2` of `C2` can be successfully called, and that it fails otherwise. Suppose further that the `msg_morph` component of `f` sends all calls to an entrypoint `e1` of `C1` to calls to `e2`, and that the `state_morph` component of `f` sends a state of `C1` with an analogous boolean set at `true` to one of `C2` with the boolean set at `false`, and visa versa. Then by morphism induction on the trace of `C1`, we get for free that calls to `e1` succeed only when the analogous boolean in the state of `C1` is set at `false`, rather than `true`. The relationship encoded by `f` between contracts `C1` and `C2` shows that `C1` and `C2` use opposing, but predictably related, logic for execution, which allows us to reuse proofs on `C2` to prove analogous results on `C1`.

5.2 Isomorphisms and Propositional Indistinguishability

This relationship between contracts strengthens when we have a pair of morphisms

```
f : ContractMorphism C1 C2   and   g : ContractMorphism C2 C1
```

such that `compose_cm g f = id_cm C1` and `compose_cm f g = id_cm C2`. This is an *isomorphism* of contracts. Isomorphisms of contracts are particularly strong; the component functions are equivalences of types and they induce a bisimulation of contracts in ConCert.

Since bisimulation is a strong and mathematically stable notion of equivalence [17], future work could investigate proof transport over contract isomorphisms, building on recent work in Coq-based formal methods. For example, we may wish to prove results on a contract optimized for formal reasoning, and transport those onto a bisimlar, performant contract, similar to the work of Cohen *et al.* [5]. This might include altering certain data types while maintaining an equivalence; chosen data types have a strong influence on the structure of proofs and can be nontrivial to transport [11, 15, 20].

6 Conclusion

Our goal in this paper was to provide a formal framework for formally specifying and verifying smart contract upgrades in Coq. To do so we introduced the notion of a contract morphism, which encodes a formal relationship between execution traces of two contracts. We argued that this was a suitable, formal notion with which to reason about contract upgrades and provided examples of contract upgrades which can be specified and verified with contract morphisms. To our knowledge, this is the first time that the intent of an upgrade has been articulated explicitly in formal specification, and is the first formal attempt at reasoning explicitly about contract upgrades in a formal setting.

This work is intended to be a preliminary framework for reasoning about contract upgrades in Coq. As such, there are practical questions to be asked, such as whether these tools are even feasible on gas-optimized code, which can be difficult to formally reason about. Even so we are optimistic, as the previously-mentioned work by Ringer *et al.* in proof repair is practically useful and resembles our framework from a theoretical standpoint. Since the status quo is to simply update the formal specification of a previous version into the specification of the new, we hope that contract morphisms will be a strong start to efficient and rigorous verification of contract upgrades.

References

1. Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. Extracting smart contracts tested and verified in Coq. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021, pages 105–121, New York, NY, USA, January 2021. Association for Computing Machinery. doi:10.1145/3437992.3439934.
2. 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.
3. Pedro Antonino, Juliandson Ferreira, Augusto Sampaio, and A. W. Roscoe. Specification is Law: Safe Creation and Upgrade of Ethereum Smart Contracts. In *Software Engineering and Formal Methods: 20th International Conference, SEFM 2022, Berlin, Germany, September 26–30, 2022, Proceedings*, pages 227–243. Springer, 2022.
4. Rob Behnke. Explained: The NowSwap Protocol Hack. https://halborn.com/explained-the-nowswap-protocol-hack-september-2021/, September 2021. Accessed January 2024.
5. Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In *International Conference on Certified Programs and Proofs*, pages 147–162. Springer, 2013.
6. Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, Vikram Saraph, and Eric Koskinen. Proof-Carrying Smart Contracts. In Aviv Zohar, Ittay Eyal, Vanessa Teague, Jeremy Clark, Andrea Bracciali, Federico Pintore, and Massimiliano Sala, editors, *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 325–338, Berlin, Heidelberg, 2019. Springer. doi:10.1007/978-3-662-58820-8_22.
7. etherscan.io. Nomad Bridge Exploit. Transaction 0xa5fe9d044e4f3e5aa5bc4c0709333cd2190cba0f4e7f16bcf73f49f83e4a5460, 2022.
8. Uranium Finance. Uranium Finance Exploit. https://uraniumfinance.medium.com/exploit-d3a88921531c, April 2021. Accessed January 2024.
9. Immunefi. Hack Analysis: Nomad Bridge, August 2022. https://medium.com/immunefi/hack-analysis-nomad-bridge-august-2022-5aa63d53814a, January 2023.
10. Oussama Jebbar, Ferhat Khendek, and Maria Toeroe. Upgrade of highly available systems: Formal methods at the rescue. In *2017 IEEE International Conference on Information Reuse and Integration (IRI)*, pages 270–274. IEEE, 2017.
11. Nicolas Magaud and Yves Bertot. Changing data structures in type theory: A study of natural numbers. In *International Workshop on Types for Proofs and Programs*, pages 181–196. Springer, 2000.
12. Stephen McCamant and Michael D Ernst. Predicting problems caused by component upgrades. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 287–296, 2003.
13. Nick Mudge. EIP-2535: Diamonds, Multi-Facet Proxy. https://eips.ethereum.org/EIPS/eip-2535. Accessed January 2024.
14. Talia Ringer. *Proof Repair*. University of Washington, 2021.
15. 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.
16. Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Adapting proof automation to adapt proofs. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 115–129, 2018.
17. Davide Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8(5):447–479, October 1998. doi:10.1017/S0960129598002527.
18. Amritraj Singh, Reza M Parizi, Qi Zhang, Kim-Kwang Raymond Choo, and Ali Dehghantanha. Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities. *Computers & Security*, 88:101654, 2020.

19 Derek Sorensen. (In)Correct Smart Contract Specifications. *IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2024.

20 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.

21 Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, pages 154–165, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2854065.2854081.

22 Jiahua Xu, Krzysztof Paruch, Simon Cousaert, and Yebo Feng. SoK: Decentralized Exchanges (DEX) with Automated Market Maker (AMM) Protocols. *ACM Computing Surveys*, 55(11):1–50, 2023.

A Practical Notion of Liveness in Smart Contract Applications

Jonas Schiffl[1]

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

Smart contracts are programs which manage resources in blockchain networks in an automated fashion. In this context, liveness properties are often essential: If I transfer money to a contract, will I eventually get it back?

This kind of property can be hard to specify and verify, in particular because application-specific fairness assumptions w.r.t. function invocation and the behavior of other parties are usually necessary for any liveness proof to succeed. In this work, we analyze smart contract liveness properties discussed in the literature. We find that the smart contract paradigm of decentralization and trustlessness implies that "real" liveness properties do not usually occur. The properties that have been classified as liveness can be more aptly described as *enabledness*, i.e., the ability of an agent to induce a state change, such as a transfer of resources.

Our contribution in this work is a specification language based on LTL to capture this kind of property. It features some special constructs to describe common properties in smart contracts, such as transfers or ownership of cryptocurrency. We show how often-used examples of liveness properties can be succinctly specified in our language. Moreover, we show how our notion of liveness can simplify formal verification compared to existing approaches.

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.2024.8

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

Smart contracts are programs which run in conjunction with blockchains. They typically manage resources, especially cryptocurrency. Abstractly, a smart contract application can be viewed as a set of functions and state variables. Actors in the network can call the functions and thereby change an application's state. Function calls are executed atomically in no pre-defined order.

Due to their unique characteristics, it is very important that smart contracts are correct upon deployment. In this work, we propose a novel perspective on an important and challenging class of correctness properties, namely *liveness*, in the context of smart contracts. In general, liveness properties can take many forms, depending on the application domain. One classic example is termination: Given a function, we may ask whether it always finishes

[1] Corresponding author

© Jonas Schiffl and Bernhard Beckert;
licensed under Creative Commons License CC-BY 4.0

5th International Workshop on Formal Methods for Blockchains (FMBC 2024).
Editors: Bruno Bernardo and Diego Marmsoler; Article No. 8; pp. 8:1–8:13
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

execution. In other domains, especially in distributed or parallel settings, deadlock freedom is essential: Is there always a way to continue execution, or is it possible to reach a situation where no progress can be made?

In this work, we argue that in the domain of smart contracts, liveness properties typically require that a certain functionality is (or becomes) accessible to an actor. In Section 2, we substantiate this intuition by analyzing the examples given in existing literature on smart contract liveness verification. In Section 3, we formalize our notion and develop a specification language for smart contract application liveness properties, based on a subset of LTL. We also sketch possibilities for verification. In Section 4, we demonstrate the use of the language on some examples from literature.

1.1 Related Work

Formal verification of smart contracts is a very active field of research. Most of the work targets smart contracts on the Ethereum platform. Many approaches focus on the detection of pre-defined vulnerabilities, which are detected by various kinds of static analysis. Recent overviews are given by He et al. [5] and Munir et al. [9].

Other tools also allow specification and verification of user-defined correctness properties. Recent versions of the Solidity compiler itself include an SMT-based tool that checks the validity of assertions [7], hinting at the high importance of formal verification for smart contract development. Other tools allow even more powerful properties to be specified and verified. In solc-verify[4] and Celestial [2], developers can specify invariants and function contracts consisting of first-order logic pre- and postconditions as well as frame conditions. VerX [12] introduces temporal operators and verification of safety properties.

Nam and Kil [10] present an approach for model-checking ATL properties of smart contracts by translating Solidity to the language of the MCMAS model checker. Their approach focuses on strategies and behaviors of actors, highlighting the need to consider different agents and their interests.

In the approach of Godoy et. al. [3], Solidity smart contracts are manually translated to the Alloy modelling language, providing an abstract, state transition based view of the smart contract application for visualization and auditing purposes. The approach works with the concept of enabledness of smart contract functions in the form of *enabledness preserving abstractions*. `requires` statements in the code determine whether a function is enabled, relating it to the notion of enabledness we develop in this work.

Furthermore, to our knowledge, there are two tools for specification and verification of liveness properties. Both are specific to the Solidity programming language. VeriSolid [8] is a tool for developing Solidity smart contracts through modeling them as state transition systems. The state of an application is modeled explicitly. Transitions are written directly in the supported Solidity subset. This model is translated into a BIP (Behavior, Interaction, Priority) model, which can be model-checked, e.g., for safety and liveness properties like deadlock freedom. VeriSolid allows specification of liveness properties in CTL. However, the properties that can be proven are concerned with successful termination after a function is called. There is no notion of fairness assumptions or the ability of an actor to effect a transfer.

SmartPulse [14] is a tool for checking safety and liveness properties of a given Solidity smart contract. Properties are specified in SmartLTL, which contains primitives for functions being called, functions finishing execution, reverting, and sending Ether (Ethereum's built-in cryptocurrency). Fairness assumptions can be specified if necessary to prove liveness properties. In addition to source code and property specifications, an environment is

specified, consisting of an attacker model (e.g., bounds on the number of re-entrant calls) and a blockchain model (e.g., gas costs of function calls). The SmartPulse paper provides a number of example liveness properties which we will discuss in the next section.

2 Liveness Properties in Smart Contracts

Smart contract platforms have several characteristics that influence what kind of liveness properties are important in a smart contract application. First, smart contracts exist in an open world: As a matter of principle, anyone can call any function and thereby trigger a transaction. Furthermore, smart contract platforms specifically exist for use cases where participants in the network do not necessarily trust each other. Therefore, participants generally cannot be assumed to behave in any particular way, at least in the absence of incentives.

A second defining characteristic of smart contracts is money: Most smart contract platforms have some form of cryptocurrency built in, and transferring currency or tokens is a part of almost all real-world smart contract applications. This means that there are usually financial incentives.

These characteristics lead to a special kind of liveness property which is highly common in smart contracts: “If I transfer money to a smart contract, will I get it back?” Or, more generally: Will some desired state change eventually happen? We elaborate on this kind of liveness property via a few simple examples, and demonstrate its pervasiveness by a brief review of example liveness properties in the literature.

2.1 Simple Bank Example

An example often used to showcase basic functionality is a simple smart contract version of a bank, which allows other accounts to deposit money, logs the balance of each account, and enables withdrawing funds according to the caller’s balance (which is stored in a mapping `bals`). There are only two public functions, `deposit` and `withdraw`. They both have no precondition. The postcondition is that money is transferred from the caller to the bank contract (or vice versa for `withdraw`) and that the `bals` mapping is updated accordingly.

One major correctness property of this application can be viewed as a liveness property: If I deposit money in the bank, I will eventually get it back.

2.2 Escrow Example

Another common example is an escrow, where a smart contract application acts as an intermediary for a purchase[2]. For a successful purchase, the application proceeds through a succession of predefined states, according to the actions of buyer and seller. There are several liveness properties integral to the correctness of the application, depending on the exact implementation. For example, after the buyer confirms they received the purchased item, the seller should eventually be refunded their deposit.

[2] `https://docs.soliditylang.org/en/latest/solidity-by-example.html#safe-remote-purchase`

2.3 Auction Example

Another example[3] is a smart contract implementing an auction. Here, we consider an application consisting of a single contract which has three state variables: a boolean variable `state` which records the state of the auction (open, closed, or finalized), a mapping `bids` which records the bids made by each actor resp. account, and `highestBidder` which stores the current leader of the auction. Furthermore, the contract has four functions: `bid()` transfers some amount of currency (specified by the caller) from the caller to the auction contract. If the amount is higher than the current leading bid, the `highestBidder` variable is overwritten with the caller's address, and their bid is recorded in `bids`. The function `close()` sets the `state` variable to `closed` and then assigns ownership of the auction item to the current highest bidder. The function `claim` requires that the auction is closed, but not finalized. It transfers the amount of the winning bid to the auctioneer. Finally, `withdraw()` can be called by all losing bidders. It transfers the corresponding amount (as recorded in `bids`) to the caller.

Like in the bank example, a crucial correctness property of this application is a liveness property: If an actor makes a bid, that actor will eventually either win the auction and be assigned ownership of the desired item, or they will get their money back.

2.4 Examples from Literature

The SmartPulse paper [14] lists 23 safety and liveness properties of 10 applications that can be verified with their tool. Of these, 13 are liveness properties, signified by the *eventually* keyword. All of them fall into one of two categories: In the first category are properties that represent postconditions of a function, like the following: *"If a user withdraws funds after refunds are enabled, they will eventually be sent the sum of their deposits."* The paper on VeriSolid [11] only gives a single example of a liveness property, which also falls into this category.

The second category is of the type described in the beginning of this section, stating that some desired action will happen eventually. One of the examples in this category is an auction smart contract exactly like the one described above. Another example is the following statement about a crowdfunding application: *"If the campaign fails, the backers can eventually get their refund."*

In SmartPulse, liveness properties can be specified in a variant of LTL. Properties of the second category require a *fairness assumption* about the actors' behavior in order for verification to succeed: If a withdraw functionality is available, but never called, then losing bidders will not get their money back, although they could! The fairness assumption in this auction scenario is that any losing bidder will eventually call the `withdraw` function.

2.5 Observations

From the examples above, we note several important points. First, many liveness properties in smart contract applications can be reduced to postconditions and termination of a single function. Since there are several tools for specifying and verifying function postconditions, we focus on the other kind of liveness property, which states that a desired state change will happen eventually.

[3] used, e.g., in the documentation of the Solidity programming language:
`https://docs.soliditylang.org/en/stable/solidity-by-example.html#simple-open-auction`

Concerning this kind of "real" liveness property, we observe that the crucial point about a desirable state change is whether an actor is able to effect it. There is a subtle difference: Liveness in smart contracts is not about whether something will definitely happen, but about whether someone can make it happen. In the auction example, what we want to prove is not whether every losing bidder actually gets their money back, it is that they *can* get it back (if they take the appropriate action).

This phrasing leads to the insight that in the context of smart contracts, it should be possible to specify liveness properties without having to specify assumptions regarding behavior, at all. What should be specified is *enabledness*, i.e., the ability to effect a desired result. This ability often pertains to a specific actor (or set of actors). Furthermore, on some examples, the desired change can only be effected after some fixed amount of time has elapsed, or if some other condition is met.

One last observation is that liveness is usually connected to resources, e.g., the built-in cryptocurrency of a blockchain network. Often, liveness corresponds to ownership: If an actor is able to effect a transfer of an amount of currency from a smart contract to themselves, then they own this amount, even though it is not stored in their own account.

3 Formalization of Smart Contract Liveness

In this section, we formalize the insights from the previous section and propose a practical way of specifying liveness properties for smart contract applications.

3.1 A Model of Smart Contract Applications

Our goal is a specification language that is independent of a concrete smart contract platform. Therefore, we assume a very generic model of smart contract applications (model elements in italics): First, we have a notion of *Accounts* identified by a name (e.g., an address). These can be either *External Accounts* (representing human actors) or *Contracts*. Each account has an integer-valued balance.

An application is a set of *Contracts*. Each *Contract* consists of a set of *Functions* and a set of *State Variables*. *State Variables* have a name and a type. The overall state of an application is determined by the values of all state variables (including account balances). The state only changes as a consequence of a function call, after a function has executed successfully.

Functions consist of a name, a set of parameters, and a set of return values. Each function call happens within a call context. For our purposes, this context consists of the account making the call, the amount of money transferred, and the list of parameter values at call time.

Crucially, each function also has a function contract consisting of a *Precondition* and a *Postcondition*, which are first-order predicates over the execution context and the application state. The intended semantics is as follows: When a function is called in a context that does not satisfy the precondition, the function reverts and no state change occurs. When a function is called in a state and context that satisfies its precondition, it terminates in a state which satisfies the postcondition[4].

[4] Note how this differs from preconditions e.g. in JML [6] or ACSL [1], where a function contract does not specify the behavior if the precondition is not fulfilled. These semantics would not make sense in the open, adversarial smart contracts setting.

For a given smart contract application a, we say that $\mathcal{F}$ is the set of all functions of all contracts in a, $\mathcal{V}$ the set of all state variables (qualified with the name of the contract where they are defined), and *Vals* the set of all possible values of the variables. Then the state $\mathcal{S} : \mathcal{V} \to \mathit{Vals}$ is a function which assigns each state variable a value.

For a function $f \in \mathcal{F}$, P_f is the set of all possible concrete parameter lists for f. Furthermore, we say that $\mathcal{A}$ is the set of all accounts, and the set $\mathit{Ctx} = \mathcal{A} \times P_f \times \mathbb{N}$ is the set of all call contexts. A call context $c \in \mathit{Ctx}$ is a triple consisting of the calling account, parameters, and amount of currency transferred. We write $c.\mathit{params}$, $c.\mathit{caller}$, and $c.\mathit{amt}$, respectively.

Function $\mathit{pre}_f : \mathcal{S} \times \mathit{Ctx} \to \mathbb{B}$ is the precondition of f, a predicate over the application state, the caller, and the parameters. Likewise, $\mathit{post}_f : \mathcal{S} \times \mathcal{S} \times \mathit{Ctx} \to \mathbb{B}$ is a predicate over the state before and after the execution of a function, as well as the caller and the parameters of the call.

3.2 Specification Language

In this section, we develop a specification language, based on a subset of Linear Temporal Logic, which captures our main observation about smart contract liveness properties: That is, in the context of smart contract applications, it does not make sense to specify that something "will eventually happen". Rather, one should specify that a desired functionality or state change is enabled.

Abstractly, the execution of a smart contract application can be viewed as a trace of transactions, each initiated by an account calling a function with some parameters in a way that fulfills the function's precondition. Our language enables developers to write down properties that are expected to be true in all possible execution traces.

3.2.1 LTL

Linear temporal logic (LTL, first introduced in 1977 by Pnueli [13]) is a widely used logic for describing and verifying properties of execution traces. In its original form, LTL formulas consist of a set of propositional variables, the standard boolean connectors, and some temporal operators that enable statements about traces.

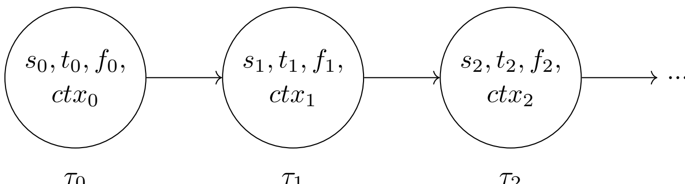
The *Next* operator $\mathbf{X}\phi$ states that some formula ϕ holds in the next step of the trace. The *Until* operator $\phi \,\mathbf{U}\, \psi$ states that eventually, ψ will hold, and ϕ must hold in every step until that point. $\Diamond$ ("eventually") with the meaning $\Diamond\phi ::= \mathit{true}\,\mathbf{U}\,\phi$ and $\Box$("globally", with $\Box\phi ::= \neg\Diamond\neg\phi$) are commonly used derivations. Furthermore, the *Weak Until* operator $\phi \,\mathbf{U_W}\, \psi$ states that ϕ must be true until a state is reached where ψ holds, but unlike $\mathbf{U}$, ψ does not necessarily have to become true at some point. The *Releases* operator $\mathbf{R}$ is the dual of $\mathbf{U}$ with $\phi \,\mathbf{R}\, \psi ::= \neg(\neg\phi \,\mathbf{U}\, \neg\psi)$. Note that both $\mathbf{U}$ and $\mathbf{R}$, in combination with negation, form a basis for the other LTL operators.

Every LTL formula can be transformed into Negation Normal Form (NNF), where the only operators are $\mathbf{U}$, $\mathbf{R}$, and $\mathbf{X}$, and where only atomic formulas are negated [15].

3.2.2 Restrictions

We adapt LTL as follows: Instead of atomic propositions, we allow predicates in first-order logic with arithmetic over the state of an application. This includes quantification over arrays and mappings, as well as over sets of accounts.

Figure 1 Our trace model of a single smart contract application execution. Each node τ_i contains the application state (s_i), the system time t_i when this state was reached, and the transaction that led to it (with each transaction consisting of the called function f_i, and the call context ctx_i).



As for the modal operators, we restrict our language in two ways: First, we omit the *Next* operator, which expresses that some condition holds in the following state. In the context of a smart contract network, it is never possible to predict which transaction is going to be executed next, as no single entity has control over this. Therefore, properties which require that something must happen exactly in the next state (as opposed to some other time) do not make sense in this domain.

Furthermore, we allow only formulas which are *Until*-free in NNF. This fragment of LTL has been called Safety LTL ([15]). Intuitively, every formula in Safety LTL rules out traces based on some finite prefix which violates the formula. This syntactic restriction captures our intuition that for smart contracts, "classical" liveness conditions (as expressed by the $\Diamond$ and $\mathbf{U}$ operators) do not make sense.

3.2.3 Domain-specific Constructs

For this restricted form of LTL, we now introduce some specification constructs which capture important "liveness" properties relevant in the smart contract domain.

We view an execution of a smart contract application as a trace (cf. Figure 1). For our purposes, each node τ_i of a trace τ consists of the application state s_i as well as the transaction which led to τ_i and the time of the transaction t_i. The transaction description consists of the name of the called function as well as the call context (caller, parameters, and transferred amount) with which it was called (we write f_i and ctx_i, respectively).

We define enabledness as a specification construct that is evaluated in one state of an execution. For a function f (with precondition pre_f over the state and the execution context) and an account a, we define that `enabled[a, par, amt](f)` is true in a node τ_i iff a can call f successfully with parameters par and amount amt in the state represented by that node:

$$\tau_i \models \texttt{enabled[a, par, amt](f)} \quad :\Leftrightarrow \quad pre_f(s_i, (a, par, amt))$$

The context, or parts of it, can be left out to indicate universal quantification, i.e., enabledness for all callers regardless of the parameters and the amount transferred with the call:

$$\tau_i \models \texttt{enabled[](f)} \quad :\Leftrightarrow \quad \forall a \in \mathcal{A}\ \forall par \in P_f : pre_f(s_i, (a, par, amt))$$

We extend this notion to predicates over the state: For a predicate over the application state p, we say that p is enabled in node τ_i if there exists a function that is enabled, and which results in a state that implies the desired state:

$$\tau_i \models \texttt{enabled[a, par,amt](c)}$$
$$:\Leftrightarrow \quad \exists f \in \mathcal{F} : \tau_i \models \texttt{enabled[a, par, amt](f)} \wedge post_f(s_{i-1}, s_i, (a, par, amt)) \rightarrow c$$

This construct includes two-state predicates, i.e., predicates which relate two states of the application by expressing a condition over the new state in terms of the previous state. Since function postconditions can also reference the state before the function was executed, the semantics are exactly the same as for enabledness of one-state predicates.

The transfer of currency from one account to another can be described in terms of a two-state predicate. This is so predominant in the smart contract domain that we create a special abbreviation.

For accounts *from* and *to*, `transfer(from, to, amt)` is true in node τ_i iff the balances of *from* and *to* changed accordingly in the step from τ_{i-1} to τ_i:

$$\begin{aligned}\tau_i \models \texttt{transfer(from, to, amt)} \quad :\Leftrightarrow \\ & s_i(from.balance) = s_{i-1}(from.balance) - amt \\ & \wedge \;\; s_i(to.balance) = s_{i-1}(to.balance) + amt\end{aligned}$$

Since liveness properties in smart contract applications are often about resource ownership, we introduce a special construct to express that an account a owns some amount of currency. We formalize ownership as the ability of an account to effect a transfer of currency to itself from the contract where the property is specified (`this`):

$$\tau_i \models \texttt{owns(a, amt)} \quad :\Leftrightarrow \quad \tau_i \models \Box\texttt{enabled[a]()transfer(this, a, amt)}$$

This property only makes sense when the `amt` expression refers to a variable which stores the amount, and which is updated in case of a transfer. One example is the mapping storing the balances in the Bank contract. We think this pattern is prevalent enough to justify the `owns` abbreviation.

Furthermore, we introduce a way to express that a transaction happened in a given step τ_i:

$$\tau_i \models \texttt{f[ctx]} \quad :\Leftrightarrow \quad \texttt{f} = f_i \wedge \texttt{ctx} = ctx_i$$

As with `enabled[]()`, the calling account and the parameters can be left out: `tx[]` is true in τ_i iff $\texttt{tx} = f_i$. This is useful for example when specifying that a certain condition always holds after some function was called.

Lastly, we provide a construct which describes that something (e.g., a transaction or state change) is always possible at least until it actually happens:

$$\tau_i \models \texttt{enabledUntil[ctx](f)} \quad :\Leftrightarrow \quad \tau_i \models \texttt{enabled[ctx](f)}\;\mathbf{U_W}\;\texttt{f[ctx]}$$

As above, the calling account and the parameters can be left out to indicate universal quantification, and the construct can also be used with a predicate instead of a transaction.

Thus, if p is a predicate over the application state, $f \in \mathcal{F}$ a function of the application, $par \in P_f$ a list of parameters for f, amt an integer expression, and $a, b \in \mathcal{A}$ accounts, the syntax of our language is as follows:

$$\begin{aligned}\phi ::= \; & true \mid false \\ & \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \rightarrow \phi_2 \\ & \mid \phi_1 \,\mathbf{R}\, \phi_2 \mid \phi_1 \,\mathbf{U_W}\, \phi_2 \mid \Box\phi \\ & \mid \texttt{enabled[}a?\ par?\ amt?\texttt{](}[p \mid f]\texttt{)} \\ & \mid \texttt{enabledU[}a?\ par?\ amt?\texttt{](}[p \mid f]\texttt{)} \\ & \mid \texttt{transfer(}a\texttt{,}b\texttt{,}amt\texttt{)} \\ & \mid \texttt{f[}a?\ par?\texttt{]} \\ & \mid \texttt{owns(}a, amt\texttt{)}\end{aligned}$$

Note that for the `enabled` and `enabledU` expressions, the account and the parameter list are optional, and the argument can be either a transaction or a predicate.

Our choice of LTL operators hints at our intentions to only care about Safety LTL formulas. As described above, we only admit formulas which are *Until*-free in NNF.

3.3 Verification

While the main focus of this work is on specification, we also want to point out that viewing smart contract liveness as a matter of enabledness can simplify the verification task.

For each property specified in an application model, the verification goal is to show that all possible executions of this application fulfill the property. For real liveness properties, as expressed by the $\Diamond$ operator, this requires additional assumptions, e.g., about the behavior of the actors in an applications. Because we restrict our specification language to safety formulas, we can, in principle, prove them by showing that the desired property is an inductive invariant.

The enabledness of a function in some context is equivalent to whether its precondition is fulfilled by that context. Likewise, the enabledness of a state or state change is equivalent to whether an enabled function exists that leads to the desired state or state change. To show that a function is always enabled, it is sufficient to show that it is enabled in the application's initial state, and that every possible call again results in a state where the function is enabled. By extension, to show that a state or state change is always enabled, it suffices to show that a function which leads to the desired state or state change is always enabled.

4 Prototypical Implementation and Evaluation

We have implemented a metamodel of smart contract applications which conforms to the model assumed for this work (cf. Section 3.1). The metamodel can be used by developers to model an application, and specify and verify the kind of liveness properties introduced in this work on the model level. Verification of application liveness properties is based on the function contracts.

Then, code skeletons consisting of function headers and function contracts (pre- and postconditions) can be generated for a specific smart contract platform (and a verification tool for this platform). After the developer completes the implementation in a way that each function conforms to its contract, the application automatically also conforms to the liveness specification.

While this model-driven approach is not the main topic of this work, we use it for a light-weight evaluation of our approach.

4.1 Implementation of Model-driven Approach

We developed a small XML-like language to describe smart contract applications consisting of several contracts. Each contract, in turn, consists of a set of state variables and functions. Our type system is inspired by Solidity, which we envision to be the main target language for our approach, but it is flexible enough to accommodate other languages as well. The type system comprises the primitive types Account, Integer (signed and unsigned), Boolean, and String. Furthermore, there are arrays and key-value mappings. Lastly, there are also the user-defined Struct and Enum types.

Each function defined by the developer has a name, a list of typed parameters, and a return type. Furthermore, we also developed a language for writing function contracts, consisting of a precondition, a postcondition, and a frame condition, which specifies which part of the application state a function may modify.

From a model written in this language, we can translate into smart contract programming languages. For evaluation, we translate into Solidity. Translating the contracts, state variables and function headers is straightforward, because the structure of our metamodel fits the structure of a Solidity application, and all of our types have an equivalent in Solidity.

The function contracts are translated to the specification language of SOLC-VERIFY[4], a tool for deductive functional verification of Solidity smart contracts. The developer provides an implementation of the generated function headers. If SOLC-VERIFYis able to prove the implementation correct against the generated formal specification, we can transfer this result back to the model, and reasonably assume that any property we can verify based on the model's function contracts is also true of the implementation.

The verification of liveness properties is not yet implemented in an automated fashion. We have developed a translation from the function contracts to an SMT encoding. For simple examples, our prototypical approach suffices to prove that a function is enabled in a given state (e.g., after initialization or after a given transaction), and that it will remain enabled, by proving that its precondition is an invariant.

4.2 Evaluation

In this section, we describe how to specify the liveness properties of the examples discussed in Section 2. We also sketch how the properties can be verified and discuss the limitations and advantages of our approach in general.

4.2.1 Bank

The main correctness property of the simple bank application (cf. Section 2.1) is that every customer can withdraw all their funds whenever they want. The customer balances are stored in a key-value mapping `bals`.

$\forall a \in \mathcal{A} : \Box$`enabled[a](a.balance ==  old{a.balance} + old{bals[a]})`

In this case, we can also specify where the money comes from, and use the `transfer` shorthand:

$\forall a \in \mathcal{A} : \Box$`enabled[a](transfer(this, a, bals[a])`

For this, we can also use the `owns` abbreviation and simply write

$\forall a \in \mathcal{A} :$ `owns(a, balances[a])`

Verification is straightforward: The `withdraw` function does not have a precondition. It is therefore always enabled for every caller, and the postcondition matches the desired property exactly.

4.2.2 Escrow

In the escrow example (Section 2.2), one liveness property is that the seller can get their deposit back after the buyer confirms the reception of the item. In our approach, this can be modeled as follows:

`confirmPurchase[buyer]` $\rightarrow$ `enabledUntil[seller](transfer(this, seller, deposit))`

This means that after the `confirmPurchase` method is called successfully, the seller is able to effect a refund of their deposit - of course, only until this actually happens.

This can be verified by showing that the only function that is enabled after the successful call to `confirmPurchase` is `refundSeller`, and that its postcondition implies the desired effect.

4.2.3 Auction

For the auction example (Section 2.3), we consider two properties: Bidders must be refunded if they do not win, and the seller should be able to claim the winning bid after the auction closes.

For the losing bidders, the property is similar to that of the bank, with the difference being that the current highest bidder can not withdraw:

$$\forall a \in \mathcal{A} : \Box(\texttt{a == highestBidder} \lor \texttt{enabled[a](transfer(this, a, bals[a]))}$$

Note that in this example, we cannot use the `owns` shortcut, because it includes a $\Box$ operator, so that the resulting property might actually not be true: After all, a losing bidder might increase their bid to become the highest bidder again.

Verification is also similar to the bank example. The `withdraw` function has only one precondition, which is that the caller must not be the current highest bidder. Therefore, it is always enabled for all other accounts. From this, it follows that the desired property is indeed an invariant.

For the seller in the auction, the desired property is that after the auction is closed, they get paid. This can be specified in two steps. First, after the auction ends, it can be closed:

$$\texttt{time > endTime} \rightarrow \Box\texttt{enabled[](close)}$$

Second, after the auction is closed, the seller can call the `claim` function to be paid the auction price from the contract:

$$\texttt{close[]} \rightarrow \texttt{enabledUntil[seller](claim())}$$

Another correct formalization of this second property can be expressed with the application state instead of a transaction expression:

$$\texttt{state == closed} \rightarrow \texttt{enabledUntil[seller](claim())}$$

Other possible formalizations could express the ability to effect a transfer, instead of the enabledness of the claim function. In this example, there are many different reasonable ways of specifying the desired property.

Verification relies on the fact that after the `close()` function is successfully executed, the `state` variable is in the state `closed` (as implied by the postcondition). This means that the `claim()` function is enabled for the seller. Since no other function is enabled, we derive that the enabledness of the `claim()` function is an invariant until it is actually called.

At first glance, it seems that the seller property it would be easier to specify with a $\Diamond$ operator, like this:

$$\Diamond\texttt{enabled[seller](claim)}$$

This property holds, but additional assumptions about the seller's behavior would have to be given in order to be able to prove this.

4.2.4 General Remarks

Our specification language can be used to express all properties yielded by our literature research on smart contract liveness properties. This shows that the restrictions in our language (no *Next*, and only *Until*-free formulas) are actually not needed to specify properties which are commonly perceived to be liveness properties.

In some cases, our restrictions force the specification to be explicit about how a desirable state can be reached: e.g., in the auction, the specification cannot just state $\Diamond$`enabled[](close)`, but has to show the way: `time > end` $\rightarrow$ `enabledUntil[](close)`. This not only simplifies verification, but also forces clarity in the specification. If a complex sequence of function calls is necessary to reach some goal, this might point to an overly complex implementation and possible simplification. At the very least, our approach will force the developer to document the necessary steps.

There are plausible scenarios where our notion of liveness fails to express all relevant properties. One example would be a vote with a quorum: Some desirable action will be taken according to a vote, but only after a fixed percentage of those entitled to vote have cast their vote. Will the action be taken eventually? Whether or not the participants are incentivized to vote depends on the specifics of the application. If they are sufficiently incentivized, this would constitute a case where a fairness condition makes sense, and our simpler notion would not be sufficient to specify and verify that any action will be taken. However, cases like this do not seem to be common in the smart contract world, and deciding whether a fairness assumption is plausible can be very challenging. We leave this kind of question to future research.

Our model-driven approach for specification and verification enables developers to specify liveness properties on a level where the implementation of the functions is abstracted via function contracts. Therefore, we cannot rely on the implementation itself for verification. Working on the abstraction means that, in general, the properties that can be proven in our approach are a subset of the properties that would be provable directly on the implementation. However, since verification in our approach is straightforward for all example liveness properties we could find in the literature, we argue that this limitation hopefully does not matter much in practice.

5 Conclusion and Future Work

In this paper, we analyze the concept of liveness properties for smart contract applications. We find that all properties commonly perceived as liveness in the literature are not classical liveness, but can be expressed as an actor's access to some functionality. Based on this finding, we suggest a specification language based on a subset of LTL, which contains concise constructs for specifying typical properties. We also sketch how this perspective simplifies the verification task, and evaluate our approach on some typical examples.

In the future, we will look to automate the verification task. Furthermore, we will develop processes for different platforms to achieve implementations which adhere to the liveness properties specified in the model. Conversely it would be possible to translate an annotated smart contract implementation to a model, and use our approach to specify and verify liveness properties on it.

References

1 Patrick Baudin, Vincent Prevosto, Jean-Christophe Filliâtre, Yannick Moy, Benjamin Monate, and Claude Marché. ANSI/ISO C Specification Language (version 1.4), 2008.

2 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.

3 Javier Godoy, Juan Pablo Galeotti, Diego Garbervetsky, and Sebastian Uchitel. Predicate abstractions for smart contract validation. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, MODELS '22, pages 289–299, New York, NY, USA, 2022. Association for Computing Machinery. `doi:10.1145/3550355.3552462`.

4 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.

5 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.

6 Gary T Leavens, Albert L Baker, and Clyde Ruby. JML: A Java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA'98)*, pages 404–420, 1998.

7 Matteo Marescotti, Rodrigo Otoni, Leonardo Alt, Patrick Eugster, Antti E. J. Hyvärinen, and Natasha Sharygina. Accurate Smart Contract Verification Through Direct Modelling. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, LNCS, pages 178–194, Cham, 2020.

8 Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. VeriSolid: Correct-by-Design Smart Contracts for Ethereum. In Ian Goldberg and Tyler Moore, editors, *Financial Cryptography and Data Security*, pages 446–465, Cham, 2019.

9 Sundas Munir and Walid Taha. Pre-deployment Analysis of Smart Contracts – A Survey, January 2023. `arXiv:2301.06079`.

10 Wonhong Nam and Hyunyoung Kil. Formal verification of blockchain smart contracts via atl model checking. *IEEE Access*, 10:8151–8162, 2022. `doi:10.1109/ACCESS.2022.3143145`.

11 Keerthi Nelaturu, Anastasia Mavridou, Andreas Veneris, and Aron Laszka. Verified Development and Deployment of Multiple Interacting Smart Contracts with VeriSolid. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9, May 2020.

12 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.

13 Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. ieee, 1977.

14 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.

15 Shufang Zhu, Lucas M Tabajara, Jianwen Li, Geguang Pu, and Moshe Y Vardi. A symbolic approach to safety ltl synthesis. In *Hardware and Software: Verification and Testing: 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings 13*, pages 147–162. Springer, 2017.

Securing Aptos Framework with Formal Verification

Junkil Park ✉
Aptos Labs, Santa Clara, CA, USA

Teng Zhang ✉
Aptos Labs, Santa Clara, CA, USA

Wolfgang Grieskamp ✉
Aptos Labs, Santa Clara, CA, USA

Meng Xu ✉
University of Waterloo, Canada

Gerardo Di Giacomo ✉
Aptos Labs, Santa Clara, CA, USA

Kundu Chen ✉
MoveBit, Hong Kong

Yi Lu ✉
Bitslab, Singapore, Singapore

Robert Chen ✉
OtterSec, USA

Abstract

The Aptos Framework is a collection of smart contracts written in the Move language that define standard and common on-chain actions for the Aptos Network. As the security and safety of the Aptos Framework is of utmost importance, it has continuously undergone rigorous testing and comprehensive auditing. To further increase the level of assurance, we have formally verified its security and correctness. This involves identifying critical security requirements for each module, creating formal specifications, and subsequently verifying them using the Move Prover. To the best of our knowledge, this represents one of the first instances of formal verification being applied on such a large scale in a smart contract framework. This paper discusses how this rigorous effort ensures a high level of quality assurance for the Aptos Framework.

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

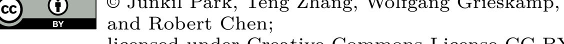
Keywords and phrases Formal verification, Smart contracts, Aptos Network, The Move language, The Move Prover

Digital Object Identifier 10.4230/OASIcs.FMBC.2024.9

1 Introduction

The Aptos Network [1] is a safe, scalable, and upgradeable layer-1 blockchain with built-in support for the Move language designed for fast and secure transaction execution. The Aptos Framework[1], similar to an operating system for a computer, serves as the foundational platform for the Aptos Network, defining its core functionalities, managing on-chain resources, and offering a standardized environment for the development of user smart contracts. It comprises a suite of Move smart contract modules that define standard and common on-chain actions for the Aptos Network, such as prologue and epilogue of transactions, the staking mechanism, and Aptos Digital Asset Standard. It is imperative to ensure the correctness and security of the Aptos Framework because the unexpected behavior of such foundational Move modules could cause substantial asset loss or disrupt the normal functioning of the network. For this reason, the Aptos Framework has continuously undergone rigorous testing and comprehensive auditing. To further increase the level of assurance, we have formally verified its security and correctness against formal specifications derived from our comprehensive and systematic methodology. We identified critical security requirements for each module and created formal specifications for large parts of the Aptos Framework, which were then verified with the Move Prover.[2]

[1] `https://github.com/aptos-labs/aptos-core/tree/fmbc-24/aptos-move/framework`
[2] Notice that a preliminary summary of this paper has been published on Medium [2]



© Junkil Park, Teng Zhang, Wolfgang Grieskamp, Meng Xu, Gerardo Di Giacomo, Kundu Chen, Yi Lu, and Robert Chen;
licensed under Creative Commons License CC-BY 4.0
5th International Workshop on Formal Methods for Blockchains (FMBC 2024).
Editors: Bruno Bernardo and Diego Marmsoler; Article No. 9; pp. 9:1–9:16

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

This rigorous approach has significantly enhanced the security and correctness assurance of the Aptos Framework. For instance, `block_prologue` is a critical Move function since it is executed at the beginning of each block. More importantly, `block_prologue` should never abort because any abort will effectively halt the network by preventing it from generating new blocks. As part of the liveness assurance of Aptos Network, we formally proved the absence of abort in `block_prologue`, despite that this function involves complex execution across 96 Move functions in 22 different Move modules. During this process, we identified and fixed some potential arithmetic overflows that may potentially trigger an abort in `block_prologue` (e.g., the `calculate_rewards_amount` function in the `stake` module).

Moreover, specifications form an integral part of the Aptos Framework *documentation*, which is in fact automatically generated based on the specifications and hence, provides an unambiguous and detailed account of the expected behavior for each function and module. Finally, integrating the Move Prover into the Continuous Integration (CI) process significantly reduces the time and cost associated with code auditing – once the specifications are pinned down, problematic code changes will likely trigger verification failure in the CI before manual auditing happens.

This paper will explain how we have secured the Aptos Framework through formal verification using the Move Prover. Section 2 introduces the Move language, Move Prover, and Aptos Framework. In Section 3, we will explain our methodology to verification of the Aptos Framework. Section 4 will discuss important findings and lessons learned from this effort. After describing the related work in Section 5, we will conclude in Section 6.

2 Background

2.1 Move as A Programming Language for Smart Contracts

The Aptos Network natively supports Move as its smart contract language. A Move program is essentially a sequence of updates that try to evolve a *global persistent memory state*, which we just call the *(global) memory*. Similar to other blockchains, updates are a series of atomic transactions. All runtime errors result in a transaction abort, which does not change the blockchain state except to transfer some currency ("gas") from the account that sent the transaction to pay for the cost of executing the transaction.

The global memory is organized as a collection of resources described by Move structures (i.e., data types). A resource in memory is indexed by a ⟨type, address⟩ pair. An address is a unique identifier in the Aptos Network that typically represents the address of a user account. For instance, the expression `exists<Coin<USD>>(addr)` will be true if there is a value of type `Coin<USD>` stored at `addr`. As seen in this example, Move uses type generics, and working with generic functions and types is rather idiomatic for Move.

A Move package consists of a set of *modules*. Each module defines a set of Move functions. These functions update the global memory and may emit events. The execution of these functions can abort explicitly because of an abort instruction (failure of an `assert`) or implicitly because of a runtime error such as an out-of-bounds vector read or integer overflows. For instance, the `coin` module provides the foundation for coins on Aptos. As one of the core public APIs defined in it, the function `deposit` is shown in Listing 1:

1. checks whether the coin with type `CoinType` is registered under the recipient's account with the address `account_addr`;
2. retrieves a mutable reference to the corresponding resource `CoinStore` from the account;
3. if the store is not frozen, deposits the input `coin` to the `CoinStore` by calling the `merge` function and emits an `Deposit` event.

If the function executes successfully, the borrowed global resource CoinStore in account_addr will be updated after the transaction is committed. Otherwise, the transaction will abort without any changes to the global memory.

Listing 1 The deposit function of the coin module.

```
public fun deposit<CoinType>(
  account_addr:address,
  coin:Coin<CoinType>
) acquires CoinStore {
  assert!(
    is_account_registered<CoinType>(account_addr),
    error::not_found(ECOIN_STORE_NOT_PUBLISHED)
  );
  let coin_store = borrow_global_mut<CoinStore<CoinType>>(account_addr);
  assert!(!coin_store.frozen, error::permission_denied(EFROZEN));
  event::emit(
    Deposit<CoinType> { account: account_addr, amount: coin.value }
  );
  merge(&mut coin_store.coin, coin);
}
```

2.2 Move Prover

The Move Prover (MVP for short) [21] is a formal verification tool for smart contracts that are written in the Move language. The Move language is tightly coupled and integrated with MVP because they have been developed and are evolving together. Move features an expressive specification language designed to define the intended behaviors of a Move smart contract. The architecture of MVP is shown in Figure 1. Move code (with the specification) is given as input to the tool chain, which produces two artifacts: an abstract syntax tree (AST) of the specifications, and the generated bytecode. They are merged into a unified object model and input to the Prover Compiler. After a series of bytecode transformations such as reference elimination, specification instrumentation and monomorphization, Boogie [14] IR is generated which is further lowered into the SMT language and subsequently fed to an SMT solver such as Z3 [6] or CVC5 [18]. MVP checks whether the code satisfies the user-given specification for all possible program variable assignments. If not, MVP generates a counterexample, that is an assignment to program variables such that the specification does not hold. The Move Prover takes great care of translating the counter-example back into the Move representation, hiding the intrinsic details of the SMT solver. MVP is fast and reliable [8], and can be used routinely during smart contract development, making the experience of running MVP similar to the experience of running compilers, linters, type checkers, and other development tools.

2.3 Move Specification Language

The Move specification language allows developers to specify the properties of their smart contracts, leveraging MVP to guarantee they behave as specified without adding any runtime cost on-chain. In the specification language, developers can provide pre- and post-conditions for functions, which include conditions over input parameters and global memory. Developers may also provide invariants over data structures as well as the contents of the global memory. The language also supports universal and existential quantification over bounded domains, such as the indices of a vector, as well as effectively unbounded domains, such as memory addresses and integers (e.g., `forall a: address: P(a)`, and `exists i: u64: Q(i)`, for some predicates P and Q). While quantifiers can render the verification problem undecidable

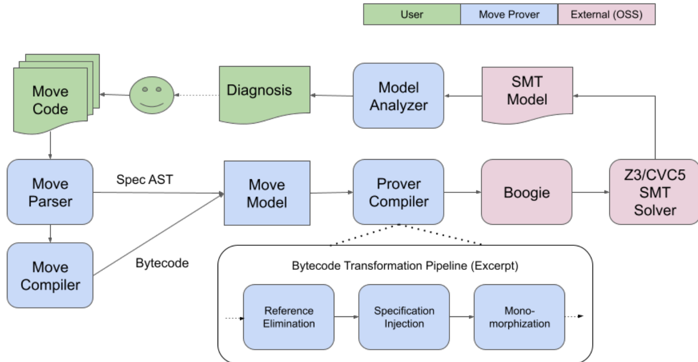


Figure 1 Architecture of the Move Prover.

and lead to timeout or an "unknown" response from SMT solvers, they offer a practical advantage: they allow for a more direct formalization of various properties, enhancing the clarity of specifications.

As an example, the spec block in Listing 2 shows the specification of the `deposit` function described in Section 2.1, which is the mathematical representation of the expected behavior of the function. Two `aborts_if` clauses specify that the function aborts if and only if at least one of the following conditions are satisfied: (1) the `CoinStore` resource for the coin with the type `CoinType` does not exist under the account `account_addr`; (2) the `CoinStore` resource is frozen. The `ensures` clause specifies that the value of the coin stored under `account_addr` is increased by the value of the input `coin` after execution. MVP guarantees that function implementation satisfies this specification for all input values and all coin types. MVP 's formal verification contrasts with testing, where a single test case only covers a specific instance of input and coin type. Moreover, once a specification for MVP is defined, it enables MVP to automatically check the specification thereafter (through CI). This automation significantly reduces the costs associated with repetitive manual audits for every modification of the smart contract.

Listing 2 The spec of the deposit function.

```
spec deposit<CoinType>(
  account_addr: address,
  coin: Coin<CoinType>
) {
  aborts_if !exists<CoinStore<CoinType>>(account_addr);
  aborts_if global<CoinStore<CoinType>>(account_addr).frozen;
  ensures global<CoinStore<CoinType>>(account_addr)).coin.value ==
    old(global<CoinStore<CoinType>>(account_addr)).coin.value +
    coin.value;
}
```

2.4 Aptos Framework

The Aptos Framework defines standard actions performed on-chain. For instance, the `genesis` module defines operations to be executed during genesis such as initializing the core account and core modules on chain. The `block_prologue` function in the `block` module defines the actions to execute before each transaction, updating the current block's metadata and the on-chain performance scores of validators. Beyond system-related actions, the framework establishes standards for coins and staking, math libraries, efficient data structures (e.g.,

smart vectors which adapt depending on their size), and cryptographic algorithms (e.g., ed25519). Given its essential role in the Aptos Network, security and safety of the framework are of utmost importance: bugs can cause network disruptions or lead to significant financial losses. We conducted thorough testing and auditing of the Aptos Framework, however, it is well-known that these measures cannot guarantee the complete absence of bugs [7]. Formal verification, in contrast, can provide rigorous proofs of critical properties. In the next section, we will present how to apply this technique effectively to the Aptos Framework.

3 Formal Verification of the Aptos Framework

Formal verification has received significant attention in the blockchain industry due to the critical importance of smart contract assurance [15, 19]. However, applying formal verification to a smart contract framework is challenging, especially when it encompasses tens of thousands of lines of code and undergoes constant evolution. Moreover, a formally verified smart contract is only as correct as its specifications. Therefore, how to devise a comprehensive set of specifications for large and evolving codebase is the key to maximize the return on investment (ROI) in formal methods.

To address this challenge, we have adopted two approaches to devising specification from distinct directions. First, we adopted a top-down approach, starting from high-level requirements to detailed specifications. Inspired by prior work of the authors [10], we established a traceability framework that enables the tracking of how high-level requirements are tested, audited, and verified. The high-level requirements and the traceability information are thoroughly documented within the Move spec files, including the links between high-level requirements and their respective formal specs. This ensures that all critical safety properties are covered in specifications with provenance. In addition, we pursued a bottom-up approach – systematically deriving specifications from individual functions and modules. This approach allowed us to uncover, verify, and document functional properties of the Aptos Framework. This section explains this combined effort in more detail.

3.1 From Security Requirement to Verification

In collaboration with our audit firm, we identified critical security requirements for each module within the Aptos Framework. These requirements were systematically documented with details covering their definition, criticality, implementation approach, and enforcement methods. The enforcement methods include audit, test, and, where appropriate, formal verification. For each requirement that can be enforced by formal verification, we created the corresponding formal specifications and verified them with MVP.

For example, a high-critical security requirement for the `coin` module is shown in Table 1. It states that the supply of a coin can be changed only by certain operations, such as `burn` and `mint`. This requirement is in place to ensure that coins cannot be created arbitrarily, which could potentially result in significant financial loss. The implementation has been audited manually, and the property has been further specified and verified.

Shown in Listing 3, this requirement is encoded as a post-condition to be applied to all functions in the `coin` module except for the `mint` and `burn` functions.

Listing 3 Post condition to enforce the high-level requirement in Table 1.

```
spec module {
  apply TotalSupplyNoChange<CoinType> to *<CoinType>
    except mint, burn, burn_from;
}
```

Table 1 A high-level requirement of the coin module.

No.	Requirement	Criticality	Implementation	Enforcement
4	The supply of a coin is only affected by burn and mint operations.	High	Only `mint` and `burn` operations on a coin alter the total supply of coins.	Formally verified in `TotalSupplyNoChange`

The definition of the post-condition `TotalSupplyNoChange` is given in Listing 4. The condition explicitly states that the `supply` value remains unchanged when comparing its values before and after execution. It uses two concepts not seen so far: *specification functions* which allow to work on state and appear in `old(..)` expressions, as well as *specification schemas* which allow to group and later inject properties:

Listing 4 Definition of TotalSupplyNoChange.

```
spec fun supply<CoinType>(addr: address): u128 {
  option::spec_borrow(global<CoinType>(addr)).supply
}
spec schema TotalSupplyNoChange<CoinType> {
  coin_type_address: address;
  ensures option::is_some(global<CoinType>(coin_type_address)) ==> supply
      (coin_address) == old(supply(coin_address))
}
```

We have authored the high-level requirements for over 40 modules, and most of the requirements are formally verified (see Appendix A). We also integrated these high-level requirement artifacts into the Aptos Framework reference documentation. Additionally, we've implemented a traceability framework that establishes connections between high-level requirements and their associated formal specifications, facilitating the mapping of formal specifications back to their originating high-level requirements. For example, the high-level requirements of the coin module can be found in the link[3]. For other modules, please see Appendix A.

3.2 Systematic functional specification

By verifying high-level requirements, we ensure that the Aptos Framework satisfies the critical security properties identified during the audit. It is also important to ensure the functionally correct behavior of the Aptos Framework. To ensure functional correctness, we have systematically inferred the specifications from the code and comments of Move functions and modules through a thorough manual process. This involves examining each Move function to identify its abort conditions and post-conditions. Additionally, we meticulously examined every struct to determine its data invariants and, by synthesizing these findings, also established global invariants for each module.

3.2.1 Abort conditions

`aborts_if` specifications cover important classes of properties, such as access control checks, input validation, and state validation. Move functions are normally designed to abort when they are called (1) by an account without permission, (2) with an input argument outside of an expected range, or (3) on an unexpected global state. For example,

[3] https://github.com/aptos-labs/aptos-core/blob/fmbc-24/aptos-move/framework/aptos-framework/doc/coin.md#high-level-req

in the Aptos Framework's staking config contract, only the `aptos_framework` account (i.e., `0x1`) can call `update_recurring_lockup_duration_secs`. Also, the input parameter `new_recurring_lockup_duration_secs` should be non-zero. It should be called only when the resource `StakingConfig` is published under the `aptos_framework` address. These expected behaviors are captured by the specification in Listing 5.

Listing 5 Specification of update_recurring_lockup_duration_secs.

```
spec update_recurring_lockup_duration_secs(
    aptos_framework: &signer,
    new_recurring_lockup_duration_secs: u64
){
    aborts_if signer::address_of(aptos_framework) != @aptos_framework;
    aborts_if new_recurring_lockup_duration_secs == 0;
    aborts_if !exists<StakingConfig>(@aptos_framework);
    ...
}
```

Given this `aborts_if` specification, MVP verifies two things. First, it verifies that the function indeed aborts when any one of the conditions holds. Second, MVP verifies that the function does not abort on any other condition. This verification is important because it allows developers to understand the complete set of conditions under which the function can abort, thus the specifications also serve as precise documentation. Notice that abort condition verification works very smoothly with MVP in practice, as finding a particular program point that aborts is a simpler problem for the solver than general post-conditions. In virtually any case we have encountered, a missed abort is quickly identified and pointed to in the Move source.

For certain functions, it is critical that they do not abort. For instance, the `block_prologue` function must never abort since it is executed with each block, and a malfunction can bring the entire network down. The block prologue complex execution involves 96 Move functions in 22 different Move modules. We formally specified all these modules and proved that the block prologue execution would never fail (or abort) in an unexpected condition (see Appendix A). The top-level specification of this function can be found in Listing 6, with some schemas included which contain further details. The clause "`aborts_if false`" means that this function should never abort, which can be proven if the `requires` conditions over the input and the global state hold. For instance, the third `requires` condition says the proposer (one of the input arguments) of creating the block must be either a reserved address (`@vm_reserved`) or an active validator, which needs to be checked by retrieving a global resource in the `stake` module. It is worth noting that this function is directly called by the VM, so conditions in the specification were manually audited at the call site.

Listing 6 Specification of block_prologue.

```
spec block_prologue {
    requires chain_status::is_operating();
    requires system_addresses::is_vm(vm);
    requires proposer == @vm_reserved
             || stake::spec_is_current_epoch_validator(proposer);
    requires timestamp >= reconfiguration::last_reconfiguration_time();
    requires (proposer == @vm_reserved)
             ==> (timestamp::spec_now_microseconds() == timestamp);
    requires (proposer != @vm_reserved)
              ==> (timestamp::spec_now_microseconds() < timestamp);
    requires exists<stake::ValidatorFees>(@aptos_framework);
    requires exists<CoinInfo<AptosCoin>>(@aptos_framework);
```

```
    include transaction_fee::
        RequiresCollectedFeesPerValueLeqBlockAptosSupply;
    include staking_config::StakingRewardsConfigRequirement;
    aborts_if false; // can never abort
}
```

3.2.2 Struct invariants

Struct invariants define the properties that the data within a struct must consistently satisfy. When specifying the Aptos Framework, we observed many cases where the invariants of a struct were implicitly present. These invariants were often documented within code comments or manifested as assertion statements, and related functions were developed while observing these implicit invariants. We have explicitly specified the invariants and verified them using MVP. For example, the struct `GasCurve` (shown in Listing 7) represents a gas curve to be used to adjust the global storage gas. It is an Eulerian approximation of an exponential curve. The fields `min_gas` and `max_gas` are the minimum and maximum gas charges respectively, and `points` is a vector of (x, y) pairs that represent the basis points of the curve where the `x`-coordinate is the utilization ratio in the curve and `y`-coordinate is the utilization multiplier in the curve.

Listing 7 Definition of GasCurve and Point.

```
struct GasCurve has copy, drop, store {
  min_gas: u64,
  max_gas: u64,
  points: vector<Point>
}
struct Point has copy, drop, store {
  x: u64,
  y: u64
}
```

For each time period, the storage gas is recalculated by interpolation into the curve that `GasCurve` defines. An implicit invariant of `GasCurve` was documented within code comments to ensure correct linear interpolation. It says that every instance of `GasCurve` is well-formed, representing a properly structured curve. Otherwise, interpolation becomes impossible, leading to an abortion of the process. We formally specified the invariant to ensure it is enforced consistently in all places. Listing 8 shows the specification of the invariant. For each point instance, the (x,y) pair must not exceed the basis point denomination. For each gas curve instance, 1) the minimum gas charge does not exceed the maximum gas charge; 2) the maximum gas charge is capped by `MAX_U64` scaled down by the basis point denomination; and 3) the gas curve is a monotonically increasing function. MVP ensures that those invariants hold everywhere in the code, that is, whenever of a value of the according types is constructed or modified. It is worth mentioning that the storage gas recalculation is part of the `block_prologue` execution path; thus, the data invariant of `GasCurve` plays an important role in the `block_prologue` verification.

Listing 8 Invariants of GasCurve and Point.

```
spec GasCurve {
  invariant min_gas <= max_gas;
  invariant (len(points) > 0 ==> points[0].x > 0);
  invariant forall i in 0..len(points) - 1:
    (points[i].x < points[i + 1].x && points[i].y <= points[i + 1].y);
  invariant max_gas <= MAX_U64 / BASIS_POINT_DENOMINATION;
}
```

```
spec Point {
  invariant x <= BASIS_POINT_DENOMINATION;
  invariant y <= BASIS_POINT_DENOMINATION;
}
```

3.2.3 Global invariants

Global invariants define the properties that the global state must consistently satisfy. Global invariants appear as members of the module specification. They are expressed as conditions over the global state that consist of Move resources published in the global memory space. Global invariants are important properties because they specify and ensure the correctness of the entire global state. Several global invariants have been inferred from some core modules of the Aptos Framework. For instance, the `stake` module defines Aptos' staking mechanism. Listing 9 shows the struct definitions related to validators. The resource `ValidatorSet` contains the configuration information for the active validators of the Aptos Network. In the struct `ValidatorConfig`, the field `validator_index` field denotes the index within the active validator set. Its value is updated following changes to the active validator set.

Listing 9 Struct definitions related to validators.

```
struct ValidatorSet has key {
  active_validators: vector<ValidatorInfo>,
  // other fields...
}
struct ValidatorInfo has copy, store, drop {
  addr: address,
  voting_power: u64,
  config: ValidatorConfig,
}
struct ValidatorConfig has key, copy, store, drop {
  validator_index: u64,
  // other fields...
}
```

In the function `update_stake_pool` (shown in Listing 10), the field `validator_index` is used to locate the corresponding validator's performance (i.e., the number of successful proposals) data entry in `ValidatorPerformance`. The function will abort if `validator_index` is equal to or greater than the length of `validators` in `ValidatorPerformance`.

Listing 10 Definition of the function update_stake_pool.

```
fun update_stake_pool(
  validator_perf: &ValidatorPerformance,
  pool_address: address,
  staking_config: &StakingConfig
) {
  let validator_config = borrow_global<ValidatorConfig>(pool_address);
  let cur_validator_perf = vector::borrow(
    &validator_perf.validators,
    validator_config.validator_index
  );
  // ...
}
```

To prove that `update_stake_pool` never aborts unexpectedly, it is necessary to establish the fact that `validator_index` never holds a value that is out-of-bounds for `validators` in `ValidatorPerformance`. Listing 11 shows the formal specification of the global invariant, which denotes the correct relation between the two resources `ValidatorSet` and `ValidatorPerformance` in the global memory. The invariant says that if the resource

`ValidatorSet` exists, all values of the `validator_index` fields in `ValidatorSet` must be smaller than the length of `validators` in `ValidatorPerformance`. Notice that this property cannot be expressed by a data invariant since those must not depend on global memory, but here, we indirectly index global memory by an address found in a `ValidatorSet`.

Listing 11 Global invariant on validators.

```
spec module {
  invariant exists<ValidatorSet>(@aptos_framework) ==>
    validator_set_is_valid();
  fun validator_set_is_valid(): bool {
    let set = global<ValidatorSet>(@aptos_framework);
    forall i in 0..len(set.active_validators):
      global<ValidatorConfig>(validators[i].addr).validator_index <
        len(global<ValidatorPerformance>(@aptos_framework).validators)
  }
}
```

4 Discussion

In this section, we discuss the benefits of formal verification for enhancing the security of the Aptos Framework, along with insights gained from this endeavor. Formal verification not only establishes proofs for key properties of the Aptos Framework but also aids in identifying bugs and issues. The process of writing formal specifications demands thorough code review, while the analysis of counterexamples unveils nuanced program behaviors, enabling the detection of certain bugs. Throughout the formal specification and verification process, numerous issues were identified, including the `aptos_governance::store_signer_cap` example shown in Listing 12.

Listing 12 Incorrect implementation of store_signer_cap.

```
public fun store_signer_cap(
  aptos_framework: &signer,
  signer_address: address,
  signer_cap: SignerCapability,
) acquires GovernanceResponsbility {
  system_addresses::assert_framework_reserved_address(
    address_of(aptos_framework)
  );
  if (!exists<GovernanceResponsbility>(@aptos_framework)) {
    move_to(aptos_framework, GovernanceResponsbility {
      signer_caps: simple_map::create<address,SignerCapability>()
    });
  };
  let signer_caps = &mut borrow_global_mut<GovernanceResponsbility>(
    @aptos_framework
  ).signer_caps;
  simple_map::add(signer_caps, signer_address, signer_cap);
}
```

Listing 12 shows the incorrect version of the function prior to our correction, caused by a subtle difference between the signer argument `aptos_framework` and the address constant `@aptos_framework`. This function misbehaves when `address_of(aptos_framework)` differs from `@aptos_framework`. While the address constant `@aptos_framework` is set to be `0x1`, the argument `aptos_framework` can represent any reserved address from `0x1` to `0xa` because the function checks if `aptos_framework` corresponds to one of these reserved addresses. Consequently, the resource `GovernanceResponsibility` might be established under `address_of(aptos_framework)`, which might not align with `@aptos_framework`. This

discrepancy raises the possibility that the resource `GovernanceResponsibility` may not be present under `@aptos_framework`, leading to the potential abort in `borrow_global_mut<GovernanceResponsibility>(@aptos_framework)`.

Listing 13 Specification of store_signer_cap.

```
spec store_signer_cap(
  aptos_framework: &signer,
  signer_address: address,
  signer_cap: SignerCapability,
) {
  aborts_if !system_addresses::is_aptos_framework_address(
    address_of(aptos_framework)
  );
  aborts_if !system_addresses::is_framework_reserved_address(
    signer_address
  );
  let signer_caps = global<GovernanceResponsbility>(
    @aptos_framework
  ).signer_caps;
  aborts_if exists<GovernanceResponsbility>(@aptos_framework) &&
    simple_map::spec_contains_key(signer_caps, signer_address);
  ensures exists<GovernanceResponsbility>(@aptos_framework);
}
```

Listing 13 formally specifies the intended behavior of the function `store_signer_cap`, which could not be verified against its incorrect version of the function. To resolve this, we amended the code to ensure that `aptos_framework` matches the address `@aptos_framework`, and that `signer_address` is indeed a reserved address.

Moreover, we have identified and defined various invariants within the modules, significantly enhancing our understanding of their behaviors and their security implications. This deeper insight has subsequently guided the refactoring and improvement of several modules. For instance, the `on_new_epoch` function in the stake module initially had a complex and lengthy while loop. In specifying the function, we divided this while loop into two separate loops. This division not only simplified the writing of loop invariants but also enhanced their readability and understandability. Furthermore, in the process of specifying the `calculate_reward_amount` function, we identified multiple issues, including overflow, rounding errors, and division by zero. To address these concerns, we refactored the code to use higher precision `u128` instead of `u64` in the intermediate steps to avoid overflow, performed division at the end to minimize rounding errors, and ensured the denominator was non-zero before division to prevent division-by-zero errors. Listing 14 presents the `calculate_reward_amount` post-refactoring.

Listing 14 the calculate_reward_amount function of the stake module.

```
fun calculate_rewards_amount(
  stake_amount: u64,
  num_successful_proposals: u64,
  num_total_proposals: u64,
  rewards_rate: u64,
  rewards_rate_denominator: u64
): u64 {
  let rewards_numerator = (stake_amount as u128) *
    (rewards_rate as u128) * (num_successful_proposals as u128);
  let rewards_denominator = (rewards_rate_denominator as u128) *
    (num_total_proposals as u128);
  if (rewards_denominator > 0) {
    ((rewards_numerator / rewards_denominator) as u64)
  } else {
    0
  }
}
```

The specifications for the Aptos Framework developed in this work are also an important component of the automatically generated reference documentation for the Aptos Framework, offering a comprehensive and precise description of the expected behavior of each function and module. This detailed documentation is vital to ongoing maintenance and quality assurance efforts. For example, the `requires` conditions specified for the `block_prologue` function, as illustrated in Listing 6, serve as assumptions that cannot be verified directly because the function is invoked by the VM, and VM verification falls outside the scope of MVP. Nonetheless, these conditions clearly document all the assumptions for the VM regarding invoking `block_prologue`, thus enabling a more streamlined and time-effective manual audit process.

Here are some key lessons learned: adopting a top-down approach proved beneficial, as it allowed us to grasp and establish high-level security requirements comprehensively, ensuring no critical security aspect was overlooked. These high-level requirements serve as an effective communication bridge between security experts and developers, proving invaluable for future code maintenance. However, verifying high-level requirements alone can be challenging without the support of local and global specifications, such as those provided through `aborts_if` specs and data/global invariants. Therefore, a bottom-up approach becomes essential for systematically developing these specifications. Moreover, integrating MVP into the Continuous Integration (CI) testing process significantly reduces the necessity for frequent audits with each contract modification. Consequently, formal verification has effectively decreased both the time and cost associated with auditing.

We encountered several challenges in writing local and global specifications, particularly due to the difficulties in writing loop invariants and timeouts with MVP. To overcome these issues, we employed several abstraction methods: (1) modeling non-linear functions as non-interpreted functions to facilitate verification of their callers; (2) applying loop unrolling techniques for complex or nested loops to enable bounded checking; and (3) for functions operating over large data domains like `u128` (e.g., `math128`), we performed verification within a smaller data domain such as `u8`. This strategy allowed for effective verification of non-linear functions like `max`, `min`, `mul_div`, `clamp`, `pow`, `floor_log2`, `sqrt`, and `ceil_div` within the domain of `u8`. However, there is more work to do specifically regarding loops in Move programs, which we hope to avoid as much as possible and instead be replaced by higher-order functions like `foreach`, `map`, `fold` and so on, which can then be specially treated by MVP.

The verification artifacts for this work can be accessed online. Appendix A offers a collection of annotated links to these artifacts. Also, Appendix B details the verification results (e.g., the number of verification conditions and the execution time) and outlines the steps for their reproduction.

5 Related Work

Many approaches have been applied to the verification of smart contracts; see e.g. the surveys [15, 19]. [19] refers to at least two dozen systems for smart contract verification. It distinguishes between *contract* and *program* level approaches. Our approach has aspects of both: we address program level properties via pre/post conditions, and contract ("blockchain state") level properties via global invariants. Among the existing approaches, the Move ecosystem is the first one where contract programming and specification language are fully integrated, and the language is designed from the first principles influenced by verification.

Methodologically, Move and MVP are thereby closer to systems like Dafny [13], or the older Spec# system [3], where instead of adding a specification approach posterior to an existing language, it is part of it from the beginning.

In contrast to other approaches that only focus on specific vulnerability patterns [5, 16, 17, 20], MVP offers a universal specification language. We support universal quantification over arbitrary memory content as well as global invariants. For comparison, the SMT Checker for Solidity [9, 11, 12] does not support quantifiers, because it interprets programming language constructs (requires and assert statements) as specifications and has no dedicated specification language. While in Solidity, one can simulate aspects of global invariants using modifiers by attaching pre/post conditions, this is not the same as our invariants, which are guaranteed to hold independent of whether a user may or (accidentally) may not attach a modifier. Moreover, our invariants are optimized to be evaluated only when necessary.

The Certora verifier [4] is a formal verification tool for Solidity smart contracts which has expressiveness comparable to that of MVP. However, because Move bytecode is fully typed and of a higher abstraction level than EVM bytecode, the verification task of Certora becomes substantially more challenging compared to MVP. This complexity could potentially render MVP more user-friendly and accessible for application.

6 Conclusion

To ensure the highest standards of quality and security within the Aptos Network, we have rigorously applied formal verification techniques to the Aptos Framework. We thoroughly documented high-level security requirements. Subsequently, we specified each aspect of the Aptos Framework, function-by-function, until most of the high-level security requirements and functional properties were eventually formally verified. During this process, we also found and fixed bugs and usability issues within MVP, thus benefiting all Aptos developers. This combined work gives the Aptos Framework a high level of quality assurance and, to the best of our knowledge, represents one of the first large-scale smart contract verification. Looking ahead, Aptos Labs plans to maintain and evolve the verification work as well as improve MVP itself. This effort aims to ensure the tool stays available for developers and auditors in the Aptos ecosystem, thereby enhancing software quality in the smart contract domain.

References

1. Aptos. The Aptos Blockchain. `https://aptos.dev/aptos-white-paper`, 2022.
2. Aptos Labs, MoveBit, and OtterSec. Securing the Aptos Framework through formal verification. `https://medium.com/aptoslabs/securing-the-aptos-framework-through-formal-verification-14124d1ed660`, 2024.
3. Mike Barnett, Robert DeLine, Manuel Fähndrich, Bart Jacobs, K. Rustan M. Leino, Wolfram Schulte, and Herman Venter. *The Spec# Programming System: Challenges and Directions*, pages 144–152. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. `doi:10.1007/978-3-540-69149-5_16`.
4. Certora. Certora Prover Documentation. `https://docs.certora.com/en/latest/index.html`, 2022.
5. ConsenSys. Mythril Classic: Security analysis tool for Ethereum smart contracts. URL: `https://github.com/skylightcyber/mythril-classic`.
6. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

7 E. W. Dijkstra. *On the Reliability of Programs*, pages 359–370. Association for Computing Machinery, New York, NY, USA, 1 edition, 2022. `doi:10.1145/3544585.3544608`.

8 David L. Dill, Wolfgang Grieskamp, Junkil Park, Shaz Qadeer, Meng Xu, and Jingyi Emma Zhong. Fast and reliable formal verification of smart contracts with the move prover (extended version). *CoRR*, abs/2110.08362, 2021. `arXiv:2110.08362`.

9 Ethereum Foundation. Solidity documentation, 2018. URL: `http://solidity.readthedocs.io`.

10 Wolfgang Grieskamp, Nicolas Kicillof, Keith Stobie, and Víctor A. Braberman. Model-based quality assurance of protocol documentation: tools and methodology. *Softw. Test. Verification Reliab.*, 21(1):55–71, 2011. `doi:10.1002/STVR.427`.

11 Ákos Hajdu and Dejan Jovanovic. solc-verify: A modular verifier for solidity smart contracts. *CoRR*, abs/1907.04262, 2019.

12 Ákos Hajdu and Dejan Jovanovic. SMT-Friendly Formalization of the Solidity Memory Model. In *ESOP*, volume 12075 of *Lecture Notes in Computer Science*, pages 224–250. Springer, 2020.

13 K. M. Leino. Accessible software verification with dafny. *IEEE Software*, 34(06):94–97, November 2017. `doi:10.1109/MS.2017.4121212`.

14 K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In Javier Esparza and Rupak Majumdar, editors, *TACAS*, pages 312–327, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

15 Jing Liu and Zhentian Liu. A survey on security verification of blockchain smart contracts. *IEEE Access*, 7:77894–77904, 2019.

16 Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *ACM Conference on Computer and Communications Security*, pages 254–269. ACM, 2016.

17 Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *ACSAC*, pages 653–663. ACM, 2018.

18 The CVC Team. CVC5. URL: `https://github.com/cvc5/cvc5`.

19 Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. A survey of smart contract formal specification and verification. *CoRR*, abs/2008.02712, 2020. `arXiv:2008.02712`.

20 Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. Securify: Practical security analysis of smart contracts. In *ACM Conference on Computer and Communications Security*, pages 67–82. ACM, 2018.

21 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 Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 137–150. Springer International Publishing, 2020.

A Verification Artifacts

This section overviews the artifacts of this verification work. The Aptos Framework consists of three Move packages such as

- `move-stdlib`: the common standard library of vanilla Move,
- `aptos-stdlib`: the Aptos-specific standard library,
- `aptos-framework`: the Aptos' standard modules for coin, staking, voting, and other operations

We've specified and verified all three Move packages. The Move modules and specs can be found via the following permanent links:

- `move-stdlib` modules and specs:
  - (modules) `https://github.com/aptos-labs/aptos-core/blob/fmbc-24/aptos-move/framework/move-stdlib/doc/overview.md#module-index`

  - (specs) https://github.com/aptos-labs/aptos-core/blob/fmbc-24/aptos-move/framework/move-stdlib/doc/overview.md#specification-index
- aptos-stdlib modules and specs:
  - (modules) https://github.com/aptos-labs/aptos-core/blob/fmbc-24/aptos-move/framework/aptos-stdlib/doc/overview.md#module-index
  - (specs) https://github.com/aptos-labs/aptos-core/blob/fmbc-24/aptos-move/framework/aptos-stdlib/doc/overview.md#specification-index
- aptos-framework modules and specs:
  - (modules) https://github.com/aptos-labs/aptos-core/blob/fmbc-24/aptos-move/framework/aptos-framework/doc/overview.md#module-index
  - (specs) https://github.com/aptos-labs/aptos-core/blob/fmbc-24/aptos-move/framework/aptos-framework/doc/overview.md#specification-index

Moreover, the high-level security requirement document is an important artifact of this verification work. The high-level security requirements are presented in the markdown table formats within the Aptos Framework's reference documentation, including hyperlinks to the relevant formal specifications, facilitating requirement traceability. All the high-level security requirements of aptos-framework can be accessed through this index: https://github.com/aptos-labs/aptos-core/blob/fmbc-24/aptos-move/framework/aptos-framework/doc/overview.md#high-level-security-requirement-index. Please note that if the browser does not automatically navigate to the item linked from the index pages above, refreshing your browser should resolve the issue and direct you to the correct item.

The Aptos Framework is open-source and accessible on GitHub. Each module in move-stdlib and its spec is contained in a single Move source file .move, while aptos-stdlib and aptos-framework define modules in .move and their specs in .spec.move separately. The source code of Move modules and specs can be found at the following link:

- move-stdlib: https://github.com/aptos-labs/aptos-core/tree/fmbc-24/aptos-move/framework/move-stdlib/sources
- aptos-stdlib: https://github.com/aptos-labs/aptos-core/tree/fmbc-24/aptos-move/framework/aptos-stdlib/sources
- aptos-framework: https://github.com/aptos-labs/aptos-core/tree/fmbc-24/aptos-move/framework/aptos-framework/sources

B Reproducing the Verification Result

This section explains how to reproduce the verification result. The steps are summarized as follows:

1. Install Aptos CLI. The Move Prover (MVP) is integrated in the Aptos CLI[4], a command line tool for developing, debugging, deploying and operating on the Aptos Network. To install the Aptos CLI, please refer to https://aptos.dev/tools/aptos-cli/install-cli/.
2. Clone the aptos-core repository. aptos-core contains the core components of the Aptos Network including the Aptos Framework and their specs. Please use the following command to download the snapshot of the repository made for this paper: `git clone --branch fmbc-24 git@github.com:aptos-labs/aptos-core.git`

[4] https://aptos.dev/tools/aptos-cli/

3. Install the dependencies of MVP. MVP requires the backend verification tools such as Z3 and Boogie. To install all the dependencies that MVP needs, please run the command `./script/dev_setup -ytp` in the `aptos-core` directory, and execute the environment command in `.profile` to properly set the environment variables such as `Z3_EXE` and `BOOGIE_EXE`.
4. Run MVP. To prove the `aptos-framework` package, please go to the `aptos-move/framework/aptos-framework` directory and run `aptos move prove` (`aptos-move/framework/move-stdlib` for `move-stdlib` and `aptos-move/framework/aptos-stdlib` for `aptos-stdlib`).

Listing 15 shows the verification result that has been performed on a Apple M1 Max machine with 64 GB of memory.

Listing 15 The verification result.

```
# The verification result for move-stdlib
[INFO] preparing module 0x1::BCS
...
[INFO] preparing module 0x1::string
[INFO] transforming bytecode
[INFO] generating verification conditions
[INFO] 138 verification conditions
[INFO] running solver
[INFO] 0.049s build, 0.023s trafo, 0.013s gen, 2.027s verify, total 2.113
    s
Success

# The verification result for aptos-stdlib
[INFO] preparing module 0x1::bls12381_algebra
...
[INFO] preparing module 0x1::smart_vector
[INFO] transforming bytecode
[INFO] generating verification conditions
[INFO] 338 verification conditions
[INFO] running solver
[INFO] 0.204s build, 0.124s trafo, 0.045s gen, 18.347s verify, total
    18.721s
Success

# The verification result for aptos-framework
[INFO] preparing module 0x1::system_addresses
...
[INFO] preparing module 0x1::staking_proxy
[INFO] transforming bytecode
[INFO] generating verification conditions
[INFO] 525 verification conditions
[INFO] running solver
[INFO] 0.735s build, 0.636s trafo, 0.193s gen, 84.024s verify, total
    85.589s
Success
```

Notice the number of verification conditions prompted for each of those commands corresponds to one function (if generic, one instantiation) and all its pre/post conditions, injected invariants, and properties inlined in the code (e.g. loop invariants).

Structured Contracts in the EUTxO Ledger Model

Polina Vinogradova
Input Output, Global

Orestis Melkonian
Input Output, Global

Philip Wadler
Input Output, Global
University of Edinburgh, UK

Manuel Chakravarty
Input Output, Global

Jacco Krijnen
Utrecht University, The Netherlands

Michael Peyton Jones
Input Output, Global

James Chapman
Input Output, Global

Tudor Ferariu
University of Edinburgh, UK

Abstract

Blockchain ledgers based on the *extended* UTxO model support fully expressive smart contracts to specify permissions for performing certain actions, such as spending transaction outputs or minting assets. There have been some attempts to standardize the implementation of stateful programs using this infrastructure, with varying degrees of success.

To remedy this, we introduce the framework of *structured contracts* to formalize what it means for a stateful program to be correctly implemented on the ledger. Using small-step semantics, our approach relates low-level ledger transitions to high-level transitions of the smart contract being specified, thus allowing users to prove that their abstract specification is adequately realized on the blockchain. We argue that the framework is versatile enough to cover a range of examples, in particular proving the equivalence of multiple concrete implementations of the same abstract specification.

Building upon prior meta-theoretical results, our results have been mechanized in the Agda proof assistant, paving the way to rigorous verification of smart contracts.

2012 ACM Subject Classification Theory of computation → Program specifications; Security and privacy → Formal methods and theory of security

Keywords and phrases blockchain, ledger, smart contract, formal verification, specification, transition systems, Agda, UTxO, EUTxO, small-step semantics

Digital Object Identifier 10.4230/OASIcs.FMBC.2024.10

1 Introduction

Many modern cryptocurrency blockchains are smart contract-enabled, meaning that they provide support for executing user-defined code as part of block or transaction processing. This code is used to specify agreements between untrusted parties that can be automatically enforced without a trusted intermediary. Examples of such contracts may include distributed exchanges (DEXs), escrow contracts, auctions, etc.

There is a lot of variation in the details of how smart contract support is implemented across different platforms. On account-based platforms such as Ethereum [6] and Tezos [17], smart contracts are inherently stateful and their states can be updated by transactions. Smart contracts in the extended UTxO (EUTxO) model, such as Cardano [20] and Ergo [13], on the other hand, take the form of boolean predicates on the transaction data and are inherently stateless. In this model, transactions specify all the changes being done to the ledger state, while contract predicates are used only to specify permissions for performing the UTxO set updates specified by the transaction, such as spending UTxOs or minting tokens.

© Polina Vinogradova, Orestis Melkonian, Philip Wadler, Manuel Chakravarty, Jacco Krijnen, Michael Peyton Jones, James Chapman, and Tudor Ferariu;
licensed under Creative Commons License CC-BY 4.0
5th International Workshop on Formal Methods for Blockchains (FMBC 2024).
Editors: Bruno Bernardo and Diego Marmsoler; Article No. 10; pp. 10:1–10:19
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Functional EUTxO programming is a less conventional paradigm than the stateful contracts of the account-based model [30, 14]. However, it has a notable advantage: the changes made by a transaction applied to the ledger are predictable, including the outputs of the contracts that it runs on-chain [20, 2]. This is because the data inspected by the executed contracts, the exact cost of on-chain contract execution, and UTxO set changes, are all fixed at the time of transaction construction. The unpredictability of transaction application and contract execution outcomes results in vulnerabilities in account-based models, such as the reentrancy or replay attacks [18, 17], which do not exist in the EUTxO model.

Like many prominent platforms [17, 6, 24, 31, 29], the Cardano implementation of the EUTxO ledger [20] is specified as a transition system. The reason for this design choice is that the evolution of the ledger takes place in atomic steps corresponding to the application of a single transaction. What sets the Cardano specification apart, however, is the formal rigor of its operational small-step semantics specification [12].

Many common contract applications require a model of stateful computation. There are multiple existing approaches to implementing and verifying specific designs of stateful programs running on the ledger [7, 15, 10], however, there are currently no principled standard practices for doing so. We propose the *structured contract framework* (SCF) as an extension of this approach to contract specification. It enables users to instantiate a small-step program specification that runs on the ledger via the use of smart contract scripts.

Generalizing the constraint-emitting-machine design pattern introduced in the seminal EUTxO paper [9] to establish a correspondence between high-level abstract state machines and low-level transactions, SCF formalizes the notion of stateful program running on the EUTxO ledger, and what it means for it to be implemented *correctly.* We do so by requiring instantiation of a stateful program to include a proof of a *simulation relation* between its specification and the ledger specification. Our generalization allows formalizing invariants (a.k.a. safety properties) of contracts for which it was not previously done in a uniform way. For example, we can express invariants of stateful contracts with state distributed across multiple UTxOs, as well as on the totality of tokens under a specific policy.

The class of contracts which can be instantiated in SCF is made up of all stateful contracts with implementations for which valid ledger evolution guarantees that the evolution of the on-chain contract state adheres to its specification. We argue that SCF constitutes a novel principled approach to stateful smart contract architecture that is amenable to formal analysis and suitable for a wide range of smart contract applications. The main contributions of this paper are:

(i) a formulation of the structured contract framework (SCF) on top of a simplified small-step semantics for EUTxO ledgers;
(ii) a case study expressing the minting policy of a single NFT as a structured contract;
(iii) a case study demonstrating the use of SCF to define two distinct ledger implementations of a single specification, including one that is distributed across multiple UTxO entries and interacting scripts.

We have mechanized our results in the Agda proof assistant [25], which are publicly available in HTML format:

`https://omelkonian.github.io/structured-contracts/`

In the future, we hope to integrate this framework into the existing Agda specification of Cardano's small-step ledger semantics.[1]

---

[1] `https://github.com/IntersectMBO/formal-ledger-specifications`

2 EUTxO ledger model

The EUTxO ledger model is a UTxO-based ledger model that supports the use of user-defined Turing-complete scripts to specify conditions for spending (consuming) UTxO entries as well as token minting and burning policies. The EUTxO model has been previously expressed in terms of a ledger state containing a list of transactions that have been validated, and a set of rules for validating incoming transactions [7]. We demonstrate here that it can be expressed as a labeled transition system with the UTxO set as its state, specified in small-step semantics, similar to the specification of the deployed Cardano ledger [20]. The transaction validation rules of the existing model are interpreted as constraints of the UTxO state transition rule in our model. Note that while in a realistic system transactions are applied to the ledger in blocks, here we abstract away block structure for simplicity.

Specifying transition relations. For some $\mathsf{Env}, \mathsf{State}$ and Input, the transition relation $\mathsf{TRANS} \subseteq \mathsf{Env} \times \mathsf{State} \times \mathsf{Input} \times \mathsf{State}$ contains 4-tuples $(e,\ s,\ i,\ s') \in \mathsf{TRANS}$ where $e \in \mathsf{Env}$ is the *environment*, $s \in \mathsf{State}$ is the *start state*, $i \in \mathsf{Input}$ is the *input*, and $s' \in \mathsf{State}$ is the *end state*.

A specification TRANS consists of one or more *transition rules*. The only 4-tuples that are members of TRANS are those that satisfy the preconditions of one of its transition rules. By convention, all variables that appear unbound in a given rule are universally quantified, unless they are bound by an explicit existential ($\exists$) or let-binding ($:=$). In the context of specifying transition rules, membership in TRANS is also denoted by the following notation:

$$e \vdash s \xrightarrow[\text{TRANS}]{i} s'$$

Input, environment, and labeled transition systems. The input and the environment together are used to calculate the possible end state(s) for a given start state, making up the *label* of the transition between the start and end states. If the transition relation is functional, there is exactly one end state for a given start state and label. We adopt the conventional distinction between environment and input due to its usefulness in the blockchain context [12]. In particular, input comes from users, e.g. transactions they submit. The environment, on the other hand, is outside the user's control, such as the blockchain time.

2.1 Ledger types

The ledger types and rules that we base this work on are, for the most part, similar to those presented in existing EUTxO ledger research [7]. We make some simplifications in order to remove details not relevant to this work. We give an overview of these for completeness, and clarify any omitted types in Appendix A. Notation we use that is outside conventional set-theoretic notation is listed in Figure 3, and explained in the text. Here, $\mathbb{B}, \mathbb{N}, \mathbb{Z}$ denote the type of Booleans, natural numbers, and integers, respectively. Some types described below are mutually recursive, so there is no natural order in which to describe them.

Value. We define $\mathsf{Value} := \mathsf{PolicyID} \mapsto (\mathsf{TokenName} \mapsto \mathsf{Quantity})$ as a nested map structure, closely following the original formulation as finitely supported functions [7]. A term of this type is a *bundle* of multiple kinds of assets. The type of an identifier of a class of fungible assets is given by $\mathsf{AssetID} := \mathsf{PolicyID} \times \mathsf{TokenName}$. $\mathsf{Quantity} := \mathbb{Z}$ is an integral value, which can be negative in the case of indicating quantities of tokens to be burned. For a given

$v \in \mathsf{Value}$, the nested map associates a quantity to each asset ID. The quantities of all assets with IDs not included in v are implicitly 0. Value forms a partial order, as well as a group with addition ($+$) and the empty map ($\emptyset$) as the zero [8]. The components of AssetID are:

(i) a script of type $\mathsf{PolicyID} := \mathsf{Script}$, which is executed any time a transaction is minting assets with this minting policy. If it validates, the transaction is allowed to mint the assets under this policy (specified in the mint field of the transaction), meaning that these assets are added to the the total amount of assets a transaction is transferring (see (v) and (ix) in Section 2.2) ;

(ii) a $\mathsf{TokenName} := [\mathsf{Char}]$, which is a character list specified by the user at the time of minting transaction construction, and is used to differentiate assets under the same minting policy. A minting policy may condition on token names.

UTxO set. The type of the UTxO set is a finite key-value map $\mathsf{UTxO} := \mathsf{OutputRef} \mapsto \mathsf{Output}$. The type of the key of this map is $\mathsf{OutputRef} := \mathsf{Tx} \times \mathsf{Ix}$, with $\mathsf{Ix} := \mathbb{N}$. A $(tx, ix) \in \mathsf{OutputRef}$ is called an output reference. It consists of a transaction tx that created the output to which it points, and index ix, which is the location of particular output in the list of outputs of that transaction. The pair uniquely identifies a transaction output. In practice, an injective function, which encodes a transaction as a natural number, is applied to a tx for inclusion in an output reference, so that OutputRef is the type $\mathbb{N} \times \mathsf{Ix}$. However, we omit this for simplicity and readability.

An output $(a, v, d) \in \mathsf{Output} := \mathsf{Script} \times \mathsf{Value} \times \mathsf{Datum}$ consists of (i) a script address a (field validator), which is run when the output is spent, (ii) an asset bundle v (field value), and (iii) a datum d (field datum), which is some additional data.

Data. Data is a type used for representing data encoded in a specific way. It is similar in structure to a CBOR encoding, c.f. the relevant Agda definitions[2] accompanying the seminal EUTxO papers [9, 7]. Data of this type is passed as arguments to scripts. The types $\mathsf{Datum} := \mathsf{Data}$ and $\mathsf{Redeemer} := \mathsf{Data}$ are both synonyms for the Data type. Conversion functions are required in order for a script to interpret Data-type inputs as the datatypes it is expecting. When the context is clear, the decoding function is called fromData, and the encoding one is toData.

Slot number. A slot number $slot \in \mathsf{Slot} := \mathbb{N}$ is a natural number used to represent the time at which a transaction is processed.

Transactions. The data structure Tx specifies a set of updates to the UTxO set. A transaction $tx \in \mathsf{Tx}$ contains (i) a set $tx.\mathsf{inputs} \in \mathsf{OutputRef} \times \mathsf{Output} \times \mathsf{Redeemer}$ of *inputs* each referencing entries in the UTxO set that the transaction is removing (spending), with their corresponding redeemers, (ii) a list of outputs $tx.\mathsf{outputs}$, which get entered into the UTxO set with the appropriately generated output references, (iii) a pair of slot numbers $tx.\mathsf{validityInterval}$ representing the validity interval of the transaction, (iv) a $tx.\mathsf{mint} \in \mathsf{Value}$ being minted by the transaction, (v) a redeemer for each of the minting policies being executed $tx.\mathsf{mintRdmrs} \in \mathsf{Script} \mapsto \mathsf{Redeemer}$, and (vi) the map $tx.\mathsf{sigs}$ of (public) keys that signed the transaction, paired with their signatures.

[2] `https://omelkonian.github.io/formal-utxo/UTxO.Types.html#DATA`

Scripts. A smart contract, or *script*, is a piece of stateless user-defined code with a boolean output, and has the (opaque) type Script. Scripts are associated with performing a specific action, such as spending an output, or minting assets. If a transaction attempts to perform an action associated with a script, that script is executed during transaction validation, and must return true (validate) given certain inputs. A script specifies the conditions a transaction must satisfy in order to be allowed to perform the associated action. We do not specify the language in which scripts are written, but we presume Turing-completeness. We write script pseudocode using set-theoretic notation.

The input to a script consists of (i) a summary of transaction data, (ii) a pointer to the specific action (within the transaction) for which the script is specifying the permission, (iii) and a piece of user-defined data we call a $\mathsf{Redeemer}$. A redeemer is defined at the time of transaction construction (by the transaction author) for each action requiring a script to be run. Evaluating a minting policy script s to validate minting tokens under policy p, run by transaction tx with redeemer r, is denoted by $[\![s]\!](r,(tx,\ p))$. To evaluate a script q, which validates spending input $i \in tx.\mathsf{inputs}$ with datum d and redeemer r, we write $[\![q]\!](d,r,(tx,\ i))$.

2.2 Ledger transition semantics

Permissible updates to the UTxO set are given by the transition system $\mathsf{LEDGER} \subseteq \mathsf{Slot} \times \mathsf{UTxO} \times \mathsf{Tx} \times \mathsf{UTxO}$. The output of the function $\mathsf{checkTx} : \mathsf{Slot} \times \mathsf{UTxO} \times \mathsf{Tx} \to \mathbb{B}$ determines whether a transaction is valid in a given state and environment. The output of $\mathsf{checkTx}(slot, utxo, tx)$ is given by the conjunction of the following checks, which are consistent with the previously specified EUTxO validation rules [7] :

(i) The transaction has at least one input:

$$tx.\mathsf{inputs} \neq \{\}$$

(ii) The current slot is within transaction validity interval:

$$slot \in tx.\mathsf{validityInterval}$$

(iii) All outputs have positive values:

$$\forall o \in tx.\mathsf{outputs},\ o.\mathsf{value} > \emptyset$$

(iv) All output references of transaction inputs exist in the UTxO:

$$\forall (oRef,\ o) \in \{(i.\mathsf{outputRef},\ i.\mathsf{output}) \mid i \in tx.\mathsf{inputs}\},\ (oRef \mapsto o) \in utxo$$

(v) Value is preserved:

$$tx.\mathsf{mint} + \sum_{i \in tx.\mathsf{inputs},\ (i.\mathsf{outputRef} \mapsto o) \in utxo} o.\mathsf{value} = \sum_{o \in tx.\mathsf{outputs}} o.\mathsf{value}$$

(vi) No output is double-spent:

$$\forall\ i,j \in tx.\mathsf{inputs},\ i.\mathsf{outputRef} = j.\mathsf{outputRef} \Rightarrow i = j$$

(vii) All inputs validate:

$$\forall\ (i,o,r) \in tx.\mathsf{inputs},\ [\![o.\mathsf{validator}]\!](o.\mathsf{datum},\ r,\ (tx,(i,o,r))) = \mathsf{true}$$

(viii) Minting redeemers are present:

$$\forall\ (s \mapsto _) \in tx.\mathsf{mint},\quad \exists\ r,\ \ (s, r) \in\ tx.\mathsf{mintRdmrs}$$

(ix) All minting scripts validate:

$$\forall\ (p, r) \in tx.\mathsf{mintRdmrs},\ \llbracket p \rrbracket(r, (tx, p)) = \mathsf{true}$$

(x) All signatures are correct:

$$\forall\ (pk \mapsto s) \in tx.\mathsf{sigs},\ \ \mathsf{checkSig}(tx, pk, s) = \mathsf{true}$$

Membership in the LEDGER set is defined using checkTx. The single rule defining LEDGER, called ApplyTx, states that $(slot,\ utxo,\ tx,\ utxo') \in \mathsf{LEDGER}$ whenever $\mathsf{checkTx}(slot,\ utxo,\ tx)$ holds and $utxo'$ is given by $\{\ i \mapsto o \in utxo\ \mid\ i \notin \mathsf{getORefs}(tx)\ \} \cup \mathsf{mkOuts}(tx)$.

$$\mathsf{ApplyTx}\ \frac{utxo' \ := \ \{\ i \mapsto o \in utxo\ \mid\ i \notin \mathsf{getORefs}(tx)\ \} \cup \mathsf{mkOuts}(tx) \qquad \mathsf{checkTx}(slot,\ utxo,\ tx)}{slot\ \vdash\ (\ utxo\ )\ \xrightarrow[\text{LEDGER}]{tx}\ (\ utxo'\ )}$$

The value $utxo'$ is calculated by removing the UTxO entries in $utxo$ corresponding to the output references of the transaction inputs, and adding the outputs of the transaction tx to the UTxO set with correctly generated output references. The function getORefs computes a UTxO set containing only the output references of transaction inputs, paired with the outputs contained in those inputs. The function mkOuts computes a UTxO set containing exactly the outputs of tx, each associated to the key (tx, ix), where the index ix is the place of the associated output in the list of tx outputs. For details, see Figure 4.

The EUTxO ledger definition [9], on which we base our semantics, models the ledger as a list of transactions, recorded in the order of processing. In essence, this is the reflexive-transitive closure of the step relation above, where there is a special initial transaction with no inputs a.k.a. the *genesis* transaction. We have deliberately left a full treatment of trace properties for future work: our model currently says nothing about an initial state and only specifies how to update an arbitrary UTxO set in accordance with the LEDGER rule. It is worth mentioning that validation check (i) would contradict the genesis transaction, although it is necessary for subsequent transaction in order to ensure *replay protection*: a transaction valid at some given point in time cannot again be valid in the future.

3 Simulations and the structured contract formalism

Intuitively, a stateful program is correctly implemented on the ledger whenever its state is observable in (i.e. computable from) the ledger state, and whenever the ledger state is updated, the observed program state is updated in accordance with the program's specification. In this section, we formalize the notion of smart contract scripts correctly implementing a specification.

The purpose of a smart contract script is to encode the conditions under which a transaction *can update a part of the ledger state* with which the script is associated, e.g. change the total quantity of tokens under a given policy. This interpretation of the use of stateless code on the ledger justifies a *stateful* program model for representing most programs running on the ledger. Stateful programs are implemented using one or more interacting

scripts controlling the updates of the contract state data within the ledger state. In this section, we specify what is required to construct a simulation relation between two transition systems specified via small-step semantics, and define a subset of such simulations that corresponds to the set of all (correctly) implemented stateful contracts on an EUTxO ledger.

3.1 Simulations

We instantiate the definition a *simulation* [23] with labeled state transition systems expressed as small-step semantics specifications.

Simulation definition. Let TRANS and STRUC be small-step labeled transition systems. A *simulation* of TRANS in STRUC, denoted by $(\mathsf{STRUC}, \sim, \simeq) \succeq \mathsf{TRANS}$, consists of of the following types together with the following relations :

$$\begin{aligned}
_ \vdash _ \xrightarrow[\text{TRANS}]{_} _ &\subseteq \mathsf{Env} \times \mathsf{State} \times \mathsf{Input} \times \mathsf{State} \\
_ \vdash _ \xrightarrow[\text{STRUC}]{_} _ &\subseteq \mathsf{Env}' \times \mathsf{State}' \times \mathsf{Input}' \times \mathsf{State}' \\
_ \sim _ &\subseteq \mathsf{State} \times \mathsf{State}' \\
_ \simeq _ &\subseteq (\mathsf{Env} \times \mathsf{Input}) \times (\mathsf{Env}' \times \mathsf{Input}')
\end{aligned}$$

such that the following holds :

$$\mathsf{Sim}\ \frac{(e,\ i) \simeq (e', j) \qquad u \sim s \qquad e \vdash (\ u\ ) \xrightarrow[\text{TRANS}]{i} (\ u'\ )}{\exists\ s',\ u' \sim s' \quad \wedge \quad e' \vdash (s) \xrightarrow[\text{STRUC}]{j} (s')} \tag{1}$$

Instantiating a simulation of TRANS in STRUC requires specifying TRANS, STRUC, π, π_{Tx}, and fulfilling the *proof obligation* Sim.

3.2 Structured contracts

The simulation definition we give is general, however, the rest of this work is geared towards reasoning about the programmable via user-defined scripts. For this reason, we define a particular class of simulations of LEDGER. Since scripts are not allowed to inspect block-level data (e.g. the current slot number), we fix the environment of the structured contract specification to be a singleton type $\{\star\}$. We also require that $\sim$ is a partial function, rather than a relation, which computes a specific contract state for a given UTxO state (or fails, returning $\star$). Additionally, we require that $\simeq$ is a function.

▶ **Definition** (Structured contract)**.** *Let* $(\mathsf{STRUC}, \sim, \simeq) \succeq \mathsf{LEDGER}$ *be a simulation. We say that it is a* structured contract *whenever* $\mathsf{Env} = \star$*, and there exist two functions* $\pi : \mathsf{UTxO} \rightarrow \mathsf{State} \cup \{\star\}$*,* $\pi_{\mathsf{Tx}} : \mathsf{Tx} \rightarrow \mathsf{Input}$ *such that :*

$$\begin{aligned}
utxo \ \sim \ s &:= (\pi(utxo) \ = \ s) \\
(slot,\ tx) \ \simeq \ (\star, i) &:= (\pi_{\mathsf{Tx}}(tx) = i)
\end{aligned}$$

Discussion. This definition states that a ledger step $(slot, utxo, tx, utxo')$ with $\pi(utxo) \neq \star$, there is a step given by $(\star, \pi(utxo), \pi_{\mathsf{Tx}}(tx), \pi(utxo')) \in \mathsf{STRUC}$ which corresponds to the ledger step. It is possible that a transaction updates the UTxO set but not the contract state, so that $\pi(utxo) = \pi(utxo')$.

We do not assume that a valid contract state can be computed from an arbitrary UTxO state. For this reason, the function π is partial. For example, it is possible that two copies of the same NFT exist in a given ledger state. When programmed correctly, an NFT minting policy would not allow this to happen. When reasoning about properties of such a policy, we ignore ledger start states where the NFT uniqueness condition has already been violated. Defining a class of structured contracts for which the relation in Equation 1 is a bisimulation [23] between STRUC and LEDGER is a more difficult problem, and we leave it for future work.

4 NFT minting policy as a structured contract

Our first structured contract example expresses a *specific minting policy*. Constructing structured contracts specifying the evolution of the quantity of tokens under a specific policy is a tool for formal analysis of minting policy code. In particular, for a correctly defined minting policy, we are able to express and prove the defining property of an NFT under this policy: at most one such token can exist on the ledger. Instantiating an NFT as a structured contract allows us to state and prove a property that is quite naturally expressed for account-based blockchains with stateful NFT contracts, such as the ERC-721 [16], but poses a challenge for EUTxO ledger program analysis. For the Agda mechanization of this example, see the accompanying code at:

`https://omelkonian.github.io/structured-contracts/NFT.html`.

We first pick an identifier for the policy we wish to express, myNFTPolicy. Before writing the policy code, we define a system NFT to specify how we want the total number of tokens on the ledger under this policy to behave. Here, the state type is State := Value, and Input := Tx.

$$\text{UpdateNFTTotal}\ \frac{i := \{\ (\mathsf{myNFTPolicy}\ \mapsto\ tkns)\ \in\ tx.\mathsf{mint}\} \qquad \emptyset \leq s \leq s + i \leq \{\ \mathsf{myNFTPolicy}\ \mapsto\ \{\emptyset \mapsto 1\}\ \}}{\vdash (\ s\ ) \xrightarrow[\text{NFT}]{tx} (\ s\ +\ i\ )}$$

This specification states that the only allowed transitions are (i) a constant one, and (ii) adding a single NFT $\{\ \mathsf{myNFTPolicy}\ \mapsto\ \{\emptyset \mapsto 1\}\ \} \in \mathsf{Value}$, to the state – if one does not yet exist. An NFT whose total ledger quantity obeys this specification can never be burned, must be the only token under its policy, and must have the empty string $\emptyset$ as a name. It also does not require any authentication to be minted. To define the policy and projection functions, we pick an output reference myNFTRef which we call an *anchor*. That is, myNFTRef must be spent by the NFT-minting transaction as a mechanism to ensure that no other transaction can mint another NFT under this policy. Next, we define the projection function,

$$\pi(utxo) := \begin{cases} s & \text{if}\ \ (s\ =\ \{\ \mathsf{myNFTPolicy}\ \mapsto\ \{\emptyset \mapsto 1\}\ \}\ \ \wedge\ \ \neg\ \mathsf{hasRef})\ \ \vee\ \ s = \emptyset \\ \star & \text{otherwise} \end{cases}$$

where

$$s := \{\ p\ \mapsto\ tkns\ \mid\ p \mapsto tkns \in \sum_{_ \mapsto out \in utxo} out.\mathsf{value},\ p = \mathsf{myNFTPolicy}\ \}$$

$$\mathsf{hasRef} := (\mathsf{myNFTRef} \mapsto _ \in utxo)$$

Here, $\pi(utxo)$ returns a non-$\star$ result when either no tokens under the $\mathsf{myNFTPolicy}$ policy exist, or only the token $\{ \mathsf{myNFTPolicy} \mapsto \{\emptyset \mapsto 1\} \}$ exists under this policy, and the anchor $\mathsf{myNFTRef}$ is not in the UTxO. We define the policy,

$$\mathsf{myNFTPolicy} := \mathsf{mkMyNFTPolicy}(\mathsf{myNFTRef})$$

$$[\![\mathsf{mkMyNFTPolicy}(myRef)]\!](_, (tx,\ pid)) := \exists\ (myRef, _, _) \ \in\ tx.\mathsf{inputs} \\ \wedge\ \{ \mathsf{myNFTPolicy} \mapsto \{\emptyset \mapsto 1\} \} \ \in\ tx.\mathsf{mint}$$

To prove Sim for the NFT contract (see Appendix B for a proof sketch, which is also mechanized in Agda), we need to make an additional assumption,

NFT re-minting protection. $\forall\ (slot,\ utxo,\ tx,\ utxo') \in \mathsf{LEDGER}$,

$$((\pi(utxo) = \emptyset\ \wedge\ \mathsf{myNFTRef} \in \mathsf{getORefs}(tx))\ \vee\ \pi(utxo) > \emptyset)\ \Rightarrow\ tx \neq \mathsf{myNFTRef.fst} \quad (2)$$

This assumption guarantees that if an NFT with policy $\mathsf{myNFTPolicy}$ exists on ledger state *utxo*, the transaction $\mathsf{myNFTRef.fst}$, whose output was spent as a condition for minting the NFT, cannot be valid again in *utxo*. It also requires that a transaction *spending* $\mathsf{myNFTRef}$ cannot be the same transaction that *added* $\mathsf{myNFTRef}$ to the UTxO set. Under reasonable constraints on an initial ledger state, replay protection (c.f. Section 2) is a global invariant of EUTxO ledger state traces (as proven in prior work on EUTxO under the name "uniqueness" [7]), from which we can directly derive re-minting protections.

We note here that, as exemplified by Assumption 2, the SCF approach to implementation allows us to express specific consequences of trace-based ledger transition system properties as local invariants. This enables the separation of reasoning about the global behavior of the ledger from reasoning about the correctness of contract implementation, for which we can make use of relevant local invariants of ledger behavior.

NFT property example. At most one NFT under the policy $\mathsf{myNFTPolicy}$ can ever exist in any *utxo* that is valid for NFT : for any *utxo* such that $\pi(utxo) \neq \star$,

$$\pi(utxo) \subseteq \{ \mathsf{myNFTPolicy} \mapsto \{\emptyset \mapsto 1\} \}$$

This is immediate from the definition of π, however, this result is meaningful. By definition of Sim, and the fact that NFT is a structured contract, it is not possible to transition from a UTxO state valid for NFT (i.e. $\pi(utxo) \neq \star$) to a state which is not valid for NFT. That is, with $(slot, utxo, tx, utxo') \in \mathsf{LEDGER}$, the updated state $\pi(utxo')$ must also always have at most one NFT under $\mathsf{myNFTPolicy}$. This also implies that at most one can ever be minted by a valid transaction applied to a *utxo* valid for NFT.

5 Multiple implementations of a single specification

In this section we present an example of a specification that has more than one correct implementation, one of which is distributed across multiple UTxO entries. The guarantee that the two implement the same specification enables contract authors to meaningfully compare them across relevant characteristics, such as space usage, or parallelizability.

5.1 Toggle specification

We define (and mechanize in the corresponding Agda code) a specification wherein the state consists of two booleans, and only one can be true at a time. We set the contract input to be $\{\mathsf{toggle}\} \cup \{\star\}$. The two booleans in the state are both flipped by the input toggle, and unchanged by $\star$. We define the transition system TOGGLE :

$$\text{Noop}\ \frac{}{(\ x,\ y\ ) \xrightarrow[\text{TOGGLE}]{\star} (\ x,\ y\ )} \qquad \text{Toggle}\ \frac{}{(\ x,\ y\ ) \xrightarrow[\text{TOGGLE}]{\mathsf{toggle}} (\ \boldsymbol{y},\ \boldsymbol{x}\ )}$$

5.2 Toggle implementations

The *naive implementation* uses the datum of a single UTxO entry to store a representation of the full state of the TOGGLE contract. The *distributed implementation* uses datums in two distinct UTxO entries to represent the first and the second value of the boolean pair that is the TOGGLE state.

Thread token scripts. We use the *thread tokens* mechanism [7] to construct a unique identifier of the UTxO (or pair of UTxOs) from which the contract state is computed. The mechanism for ensuring uniqueness of a thread token is the same as for the NFT contract example in Section 4. It relies on the replay protection property of the ledger, ensuring that if a thread token exists on the ledger, it must be unique. In both the naive and distributed implementations, the thread token minting policy guarantees that thread tokens are generated in quantity of at most 1 by a transaction that spends a specific output reference myRef.

For the naive implementation, one thread token is sufficient to identify the state-bearing UTxO. Upon minting, the policy requires the token to be placed into a UTxO locked by a specific script, which is passed as a parameter to the minting policy. This script (discussed below) ensures the correct evolution of the contract state. The datum in the UTxO containing the thread token is the initial state of the contract encoded as a pair of booleans (by the partial decoder function $\mathsf{fromData}_N : \mathsf{Data} \to \mathbb{B} \cup \{\star\}$). It can be any pair of correctly encoded booleans. See Figure 5 for the policy pseudocode.

For the distributed implementation, two distinct NFTs are needed to identify the UTxOs containing the TOGGLE state data. Both NFTs are under the same minting policy and must be minted by a single transaction, but have distinct token names, "a" and "b". Upon being minted, the policy requires that they are placed in separate UTxOs, locked by the same script (discussed below). The datum in each must be decodeable (by $\mathsf{fromData}_D : \mathsf{Data} \to \mathbb{B} \cup \{\star\}$) as a boolean. See Figure 6 for the policy pseudocode.

Validator scripts. We require different UTxO-locking scripts for our two distinct implementations. Both scripts serve the following function: when the UTxO locked by the script is spent, the script must ensure that thread tokens are propagated into UTxOs that are locked by the same validator as the spent UTxOs containing the thread tokens, and that the datums in those UTxOs are correct. This implements the Toggle rule. The Noop rule applies when the transaction does not spend the thread tokens.

For the naive version, the datum in the new UTxO containing the thread token must decode as a pair of booleans whose order is reversed as compared to the booleans encoded in the datum of the spent UTxO that previously contained the thread token. We define it by :

$$
\begin{aligned}
&[\![\mathsf{toggleVal}_N(\mathsf{myRef})]\!]((b, b'),\ r,\ (tx,\ i)) \ := \ ttt \ = \ i.\mathsf{output.value} \ \wedge \ r = \mathsf{toggle} \\
&\quad \wedge \ \exists\, o \ \in \ tx.\mathsf{outputs}, \ (b',\ b) \ = \ (o.\mathsf{datum}) \ \wedge \ (o.\mathsf{validator} \ = \ vi) \ \wedge \ (ttt = o.\mathsf{value}) \\
&\quad \textbf{where} \\
&\quad\quad vi := i.\mathsf{output.validator} \\
&\quad\quad ttt := \{\mathsf{toggleTT}_N(\mathsf{myRef},\ vi) \mapsto \{\mathsf{encode}(vi) \mapsto 1\}\}
\end{aligned}
$$

The function $\mathsf{encode} : \mathsf{Script} \to [\mathsf{Char}]$ encodes a script as a string for the purpose of specifying (via the token name) the output-locking script that must persistently lock the thread token.

The distributed implementation script ensures that both the thread token-containing UTxOs are spent simultaneously. Then, it checks that the booleans in the datums are switched places : the one that was in the UTxO with token "a" must now be in a new UTxO with token "b", and vice-versa. The validator script is given in Figure 1.

$$
\begin{aligned}
&[\![\mathsf{toggleVal}_D(\mathsf{myRef})]\!](b,\ \mathsf{toggle},\ (tx,\ i)) \ := \\
&\quad ((tta \ = \ i.\mathsf{output.value}) \Rightarrow \\
&\quad\ \exists\, o,\ o' \in \ tx.\mathsf{outputs},\ i' \in tx.\mathsf{inputs}, \\
&\quad\quad o.\mathsf{validator} \ = \ o'.\mathsf{validator} \ = \ vi \ \wedge \\
&\quad\quad \mathsf{tta} \ = \ o.\mathsf{value} \wedge ttb \ = \ o'.\mathsf{value} \ \wedge \ i'.\mathsf{output.value} \ = \ ttb \\
&\quad\quad o.\mathsf{datum} \ = \ i'.\mathsf{output.datum} \ \wedge o'.\mathsf{datum} \ = \ i.\mathsf{output.datum}\ ) \\
&\quad \wedge \\
&\quad ((ttb \ = \ i.\mathsf{output.value}) \Rightarrow \\
&\quad\ \exists\, o,\ o' \in \ tx.\mathsf{outputs},\ i' \in tx.\mathsf{inputs}, \\
&\quad\quad o.\mathsf{validator} \ = \ o'.\mathsf{validator} \ = \ vi \ \wedge \\
&\quad\quad \mathsf{tta} \ = \ o.\mathsf{value} \ \wedge \ ttb \ = \ o'.\mathsf{value} \ \wedge \ i'.\mathsf{output.value} \ = \ tta \\
&\quad\quad o.\mathsf{datum} \ = \ i.\mathsf{output.datum} \ \wedge \ o'.\mathsf{datum} \ = \ i'.\mathsf{output.datum}) \\
&\quad \wedge \\
&\quad ((tta \ = \ i.\mathsf{output.value}) \vee (ttb \ = \ i.\mathsf{output.value})) \\
&\quad\ \textbf{where} \\
&\quad\quad vi := i.\mathsf{output.validator} \\
&\quad\quad tta := \{\mathsf{toggleTT}_D(\mathsf{myRef}, vi) \mapsto \{(\mathsf{encode}(vi) \ +\!\!+ \ \text{"}a\text{"}) \mapsto 1\}\} \\
&\quad\quad ttb := \{\mathsf{toggleTT}_D(\mathsf{myRef}, vi) \mapsto \{(\mathsf{encode}(vi) \ +\!\!+ \ \text{"}b\text{"}) \mapsto 1\}\}
\end{aligned}
$$

Figure 1 TOGGLE validator script for the distributed implementation.

Ledger representation. The state projection function computations return a valid contract state (i.e. a pair of booleans) whenever the anchor reference myRef is not in the UTxO, and thread tokens have been minted according to their policy and placed alongside the appropriate datums and scripts. The input projection function returns toggle whenever a transaction contains the thread tokens in its input(s), and $\star$ otherwise. For details, see Figures 7 and 2.

$$\pi_d(utxo) \ := \begin{cases} (a,b) & \text{if } \mathsf{myRef} \notin \{\ i \mid i \mapsto o \in utxo\ \} \wedge\ \ \exists!\ (i \mapsto o,\ i' \mapsto o') \in utxo, \\ & \quad \mathsf{tta} = o.\mathsf{value} \wedge \mathsf{ttb} = o'.\mathsf{value} \\ & \quad \wedge\ o.\mathsf{validator} = \mathsf{toggleVal}_D(\mathsf{myRef}) = o'.\mathsf{validator} \\ & \quad \wedge\ o.\mathsf{datum} = a \ \wedge\ o'.\mathsf{datum} = b \\ \star & \text{otherwise} \end{cases}$$

$$\pi_{\mathsf{Tx},d}(tx) := \begin{cases} \mathsf{toggle} & \text{if } \exists\ i,\ i' \in tx.\mathsf{inputs},\ i.\mathsf{output.value} = \mathsf{tta} \ \wedge\ i'.\mathsf{output.value} = \mathsf{ttb} \\ \star & \text{otherwise} \end{cases}$$

Figure 2 TOGGLE distributed projections.

In Appendix C, we give a proof sketch for the simulation relations between TOGGLE and LEDGER to complete the instantiation of the two versions of the structured contract. The proofs are very similar to to those for the NFT contract, so we have not mechanized them. To avoid duplication of thread tokens, we again need to make the additional assumption that a transaction cannot be valid again if it has previously been applied. That is, for any $(slot,\ utxo,\ tx,\ utxo') \in \mathsf{LEDGER}$, with $\pi(utxo) \neq \star$, necessarily $tx \neq \mathsf{myRef.fst}$.

TOGGLE property example. The following property states that in any step of TOGGLE, either the state booleans are swapped, or stay the same. Its proof is immediate from the specification, regardless of the implementation.

$$(\star, (a,b), i, (c,d)) \in \mathsf{TOGGLE} \quad \Rightarrow \quad (c,d) = (b,a) \vee (c,d) = (a,b)$$

6 Related work

Scilla [14] is a intermediate-level language for expressing smart contracts as state machines on an account-based ledger model. It is formalized in Coq, and the contracts written in it are amenable to formal verification. In our work we pursue the same goal of correctly implementing stateful contracts and formally studying their behavior on the EUTxO ledger.

CoSplit [26] is a static analysis tool for implementing *sharding* in an account-based blockchain. Sharding is the act of separating contract state into smaller fragments that can be affected by commuting operations, usually for the purposes of increasing parallelism and scalability. Our work allows users to build contracts whose state is distributed across multiple UTxOs and tokens on the ledger. One of the benefits of an EUTxO ledger is that transaction application commutes [2]. Therefore, no additional work is required to ensure commutativity when updating only a part of a contract with distributed state.

The Bitcoin Modelling Language (**BitML**) [5] enables the definition of smart contracts in a particular restricted class of state machines on the Bitcoin ledger, where a *computational soundness* result securely establishes a relation between high-level BitML contracts and low-level Bitcoin transactions. The BitML state machines are less expressive than the class of specifications considered in our model, although subsequent work introduces recursion [3]. Another derivative of BitML, the **ILLUM** language [4], allows users to build Turing-complete contracts that achieve a secure simulation to UTxO. A key difference between ours approach and ILLUM appears to be the direction of the security proofs: our work presents users with a framework for choosing an implementation and proving the property that “transactions can only ever update the contract state according to the stateful contract specification”, whereas

the work on ILLUM enables, given a specified contract transition, automatically constructing *specific* transactions that are guaranteed to perform the corresponding contract update. The goal in all the above cases is again similar to ours – to guarantee certain properties of on-chain state machine implementations. Alas, we lack the accompanying meta-theoretical guarantees, but we are confident similar properties hold for our system and hope to make this formal in future work.

VeriSolid [22] synthesizes Solidity smart contracts from a state machine specification, with support for formal verification. The underlying ledger model for VeriSolid is an account-based model. Like BitML, VeriSolid is less flexible in the types of state machines that can be implemented, and how they can be implemented, but offers more automation than our work. Yet another language for expressing the formal semantics of (a subset of) Solidity exists [21], has been specified in Isabelle/HOL, and was developed with the goal of improving formal verification methods of smart contracts. This work differs from ours in the underlying ledger model, as well as the direction of the ledger-contract simulation relation.

The **K framework** [27] is a unifying formal semantics framework for all programming languages, which has been used as a tool to perform audits of smart contracts [28], as well as specifying Solidity operational semantics [19]. Auditing is a common approach to smart contract verification [1, 11], which will also be useful for structured contract specifications.

7 Conclusion

Previous work on stateful EUTxO contracts [7] constructs a consolidated (single-UTxO) on-chain implementation for a given constraint-emitting machine (CEM), including a proof that the contract state is always updated correctly by a given transaction. Constructing the implementation and associated proof in this formalism is done uniformly, and holds for any specification it is instantiated with. This fixed-implementation approach allows for specifying only a small subset of possible constraints on the updating transaction data. In this work, we introduce the *structured contract framework*, in which, for a particular specification, the contract author *decides* on an implementation that satisfies their specific requirements (e.g. on-chain memory constraints, distributed vs consolidated, etc.), and is not limited in the constraints a state update may place on the transaction data.

Our formalism defines the space of all stateful contracts implementable on the EUTxO ledger via user-defined scripts. Our work opens up the possibility of formal analysis of the behavior of a much larger class of contracts which may have previously been implemented ad-hoc, or via an ill-suited formalism. Instantiating our framework requires the definition of a small-step stateful contract specification, projection functions which compute the state and input values of a contract for a given UTxO state and transaction (resp.), and a proof of the integrity of the implementation. Similar to the proof completed for the CEMs formalism, the proof obligation in our framework ensures that on-chain state evolution of a contract adheres to its specification.

We note that the *existence* of a valid UTxO state update corresponding to a given contract update is not guaranteed, and such an update is difficult to construct in the general case. We leave this for future work. Another limitation of our approach is the lack of proof automation, which presents a challenge due to the fact that a contract state may be computed from a given UTxO state by an arbitrary user-defined function. A full formalization of structured contract behavior in terms of trace-based properties of contracts and their expression at the ledger level is needed, and is the subject of future work.

We present examples of contracts and safety properties satisfied by these examples: (i) a stateful model of an NFT policy, and, for another simple contract, a pair of (ii) a distributed and (iii) a consolidated implementation. Using our framework to construct more sophisticated and realistic contract examples, such as DEXs or account simulations, is the subject of future work. The approach we presented can also potentially be applied to existing contracts for which a small-step specification can be constructed, making them more amenable to formal verification.

References

1 Arnaud Bailly. Model-based testing with QuickCheck, 2022. URL: `https://engineering.iog.io/2022-09-28-introduce-q-d/`.

2 Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. A theory of transaction parallelism in blockchains. *Logical Methods in Computer Science*, Volume 17, Issue 4, November 2021. `doi:10.46298/lmcs-17(4:10)2021`.

3 Massimo Bartoletti, Stefano Lande, Maurizio Murgia, and Roberto Zunino. Verification of recursive bitcoin contracts. *CoRR*, abs/2011.14165, 2020. `arXiv:2011.14165`.

4 Massimo Bartoletti, Riccardo Marchesin, and Roberto Zunino. Secure compilation of rich smart contracts on poor UTXO blockchains, 2023. `arXiv:2305.09545`.

5 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*, pages 83–100. ACM, 2018.

6 Vitalik Buterin. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. `https://ethereum.org/en/whitepaper/`, 2014.

7 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`.

8 Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann Müller, Michael Peyton Jones, Polina Vinogradova, Philip Wadler, and Joachim Zahnentferner. $\text{UTXO}_{\mathsf{ma}}$: 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_9`.

9 Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The Extended UTxO model. In *Proceedings of Trusted Smart Contracts (WTSC)*, volume 12063 of *LNCS*. Springer, 2020.

10 Manuel M. T. Chakravarty, Sandro Coretti, Matthias Fitzi, Peter Gazi, Philipp Kant, Aggelos Kiayias, and Alexander Russell. Hydra: Fast isomorphic state channels. *IACR Cryptol. ePrint Arch.*, 2020:299, 2020.

11 Florent Chevrou. A journey through the auditing process of a smart contract, 2023. URL: `https://www.tweag.io/blog/2023-05-11-audit-smart-contract/`.

12 Jared Corduan, Matthias Güdemann, and Polina Vinogradova. A Formal Specification of the Cardano Ledger. `https://github.com/input-output-hk/cardano-ledger/releases/latest/download/shelley-ledger.pdf`, 2019.

13 Ergo Team. Ergo: A Resilient Platform For Contractual Money. `https://whitepaper.io/document/753/ergo-1-whitepaper`, 2019.

14 Ilya Sergey et al. Safer smart contract programming with Scilla. In *Proceedings of the ACM on Programming Languages*, volume 3 (OOPSLA), pages 1–30. ACM, 2019. doi:10.1145/3360611.

15 Pablo Lamela Seijas et al. Marlowe: Implementing and analysing financial contracts on blockchain. In *Financial Cryptography and Data Security*, pages 496–511, Cham, 2020. Springer International Publishing.

16 Ethereum Team. ERC-721 TOKEN STANDARD. https://ethereum.org/en/developers/docs/standards/tokens/erc-721, 2023.

17 LM Goodman. Tezos – a self-amending crypto-ledger (white paper), 2014.

18 Tobias Guggenberger, Vincent Schlatt, Jonathan Schmid, and Nils Urbach. A structured overview of attacks on blockchain systems. *PACIS*, page 100, 2021.

19 Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanan, Yang Liu, and Jun Sun. Semantic understanding of smart contracts: Executable operational semantics of Solidity. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1695–1712, 2020. doi:10.1109/SP40000.2020.00066.

20 Andre Knispel and Polina Vinogradova. A Formal Specification of the Cardano Ledger integrating Plutus Core. https://github.com/input-output-hk/cardano-ledger/releases/latest/download/alonzo-ledger.pdf, 2021.

21 Diego Marmsoler and Achim D. Brucker. A denotational semantics of Solidity in Isabelle/HOL. In *Software Engineering and Formal Methods: 19th International Conference, SEFM 2021, Virtual Event, December 6–10, 2021, Proceedings*, pages 403–422, Berlin, Heidelberg, 2021. Springer-Verlag. doi:10.1007/978-3-030-92124-8_23.

22 Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. VeriSolid: Correct-by-design smart contracts for Ethereum. In *International Conference on Financial Cryptography and Data Security*, pages 446–465. Springer, 2019.

23 Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999.

24 S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/en/bitcoin-paper, October 2008.

25 Ulf Norell. Dependently typed programming in Agda. In *International School on Advanced Functional Programming*, pages 230–266. Springer, 2008.

26 George Pîrlea, Amrit Kumar, and Ilya Sergey. Practical smart contract sharding with ownership and commutativity analysis. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, PLDI 2021, pages 1327–1341, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454112.

27 Grigore Roşu and Traian Şerbănuţă. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming*, 79:397–434, August 2010. doi:10.1016/j.jlap.2010.03.012.

28 Runtime Verification Team. Smart contract analysis and verification, 2023. URL: https://runtimeverification.com/smartcontract.

29 The ZILLIQA Team. The ZILLIQA Technical Whitepaper. https://docs.zilliqa.com/whitepaper.pdf, 2017.

30 The Solidity Authors. Solidity 0.8.25 documentation. https://docs.soliditylang.org/en/v0.8.25/, 2023.

31 Jan Xie. Nervos CKB: A Common Knowledge Base for Crypto-Economy. https://github.com/nervosnetwork/rfcs/blob/master/rfcs/0002-ckb/0002-ckb.md, 2018.

A EUTxO details

Figure 3 introduces non-standard syntax we use throughout.

$\mathbb{H} = \bigcup_{n=0}^{\infty} \{0,1\}^{8n}$	the type of bytestrings
$\star : \{\star\}$	the one-element set, and its one inhabitant
$a : A \cup \{\star\}$	maybe type over A
$\mathsf{fst} : (A \times B) \to A$	first projection
$(a, b) : \mathsf{Interval}[A]$	intervals over a totally-ordered set A
$Key \mapsto Value \subseteq \{ k \mapsto v \mid k \in Key, v \in Value \}$	finite map with unique keys
$[a1; ...; ak] : [C]$	finite list with terms of type C
$+\!\!+ : ([C], [C]) \to [C]$	list concatenation
$h :: t : [C]$	list with head h and tail t

Figure 3 Notation.

Figure 4 lists the primitives and derived types that comprise the foundations of the EUTxO model, along with some ancillary definitions. (Outputs normally refer to transaction IDs by hash, but we simplify here for clarity.)

B Sim relation proof sketch for NFT

Suppose $(slot, utxo, tx, utxo') \in \mathsf{LEDGER}$, and $\pi(utxo) \neq \star$. There are two disjuncts :

(i) If $i = \{ (\mathsf{myNFTPolicy} \mapsto tkns) \in tx.\mathsf{mint} \} = \emptyset$, by preservation of value (rule (v) in Section 2.2), the amount s of tokens under $\mathsf{myNFTPolicy}$ remains unchanged in $utxo'$. If $s = \emptyset$, we get $s' = \emptyset + \emptyset = \emptyset$. Then, by Assumption 2 we conclude that tx does not add myRef to $utxo$. So, $\pi(utxo')$ is defined, and $\pi(utxo) = \pi(utxo') = \emptyset$. If $s > \emptyset$, by Assumption 2, we conclude that tx cannot add an output with reference myRef in the $utxo'$. Since, by $\pi(utxo) \neq \star$, we know that myRef was not in $utxo$ either, we conclude that there is no output with myRef in $utxo'$. So, $\pi(utxo) = \pi(utxo')$.

(ii) If $i \neq \emptyset$, tokens under $\mathsf{myNFTPolicy}$ are being minted, and the policy must be checked by ledger rule (ix) in Section 2.2. Necessarily, by $\mathsf{myNFTPolicy}$, $i = \{pid \mapsto \{\emptyset \mapsto 1\}\}$. If $s = \emptyset$, $s' = s + i = i$ is the new total amount of tokens under policy $\mathsf{myNFTPolicy}$ in the UTxO. The unique output with reference myRef must be removed from the UTxO by tx, so that it is not contained in $utxo'$. By Assumption 2, it is also not added back by tx to $utxo'$. Then, $\pi(utxo') \neq \star$, and is equal to i. If $s \geq \emptyset$, an output with reference myRef is not in $utxo'$. So, $\mathsf{myNFTPolicy}$ fails, and the tx is not valid on the ledger.

C Sim relation proof sketch for TOGGLE

Suppose that $(slot, utxo, tx, utxo')$ and $\pi(utxo) = (a, b)$. We first observe that each of the thread tokens in either implementation is present in an input of the transaction if and only if it is present in the output. This is because $\pi(utxo) = (a, b)$ implies that the unique token(s) already exists in the UTxO set, and the minting policy cannot be satisfied, which holds because $\mathsf{myRef} \in \{ i.\mathsf{outputRef} \mid i \in tx.\mathsf{inputs} \}$ contradicts $\pi(utxo) = (a, b)$. So, thread tokens are not being minted or burned, and, by rule (v) in Section 2.2, we can make the required conclusion.

Ledger primitives

$$
\begin{array}{lcll}
[\![_]\!] & : & \mathsf{Script} \to \mathsf{Datum} \times \mathsf{Redeemer} \times \mathsf{ValidatorContext} \to \mathbb{B} & \textit{applies a validator script to its arguments} \\
[\![_]\!] & : & \mathsf{Script} \to \mathsf{Redeemer} \times \mathsf{PolicyContext} \to \mathbb{B} & \textit{applies a monetary policy to its arguments} \\
\mathsf{checkSig} & : & \mathsf{Tx} \to \mathsf{PubKey} \to \mathbb{H} \to \mathbb{B} & \textit{checks that a given key signed a transaction}
\end{array}
$$

Defined types

$$
\begin{array}{rcl}
\mathsf{Signature} & = & \mathsf{PubKey} \mapsto \mathbb{H} \\
\mathsf{OutputRef} & = & (\mathsf{id} : \mathsf{Tx}, \mathsf{index} : \mathsf{Ix}) \\
\\
\mathsf{Output} & = & (\mathsf{validator} : \mathsf{Script}, \\
& & \ \mathsf{value} : \mathsf{Value}, \\
& & \ \mathsf{datum} : \mathsf{Data}) \\
\mathsf{TxInput} & = & (\mathsf{outputRef} : \mathsf{OutputRef}, \\
& & \ \mathsf{output} : \mathsf{Output}, \\
& & \ \mathsf{redeemer} : \mathsf{Redeemer}) \\
\mathsf{Tx} & = & (\mathsf{inputs} : \mathbb{P}\ \mathsf{TxInput}, \\
& & \ \mathsf{outputs} : [\mathsf{Output}], \\
& & \ \mathsf{validityInterval} : \mathsf{Interval}[\mathsf{Slot}], \\
& & \ \mathsf{mint} : \mathsf{Value}, \\
& & \ \mathsf{mintRdmrs} : \mathsf{Script} \mapsto \mathsf{Redeemer}, \\
& & \ \mathsf{sigs} : \mathsf{Signature}) \\
\\
\mathsf{ValidatorContext} & = & (\mathsf{Tx}, (\mathsf{Tx},\ \mathsf{TxInput})) \\
\mathsf{PolicyContext} & = & (\mathsf{Tx}, \mathsf{PolicyID})
\end{array}
$$

Helper functions

$$
\begin{array}{l}
\mathsf{toMap} : \mathsf{Ix} \to [\mathsf{Output}] \to (\mathsf{Ix} \mapsto \mathsf{Output}) \\
\mathsf{toMap}(_,\ [\,]) \qquad\qquad = [\,] \\
\mathsf{toMap}(ix,\ u\ ::\ outs) = \{\ ix \mapsto u\ \} \cup \mathsf{toMap}(ix + 1,\ outs)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{mkOuts} : \mathsf{Tx} \to \mathsf{UTxO} \\
\mathsf{mkOuts}(tx) = \{\ (tx,\ ix) \mapsto o\ \mid\ (ix \mapsto o) \in\ \mathsf{toMap}(0,\ tx.\mathsf{outputs})\ \}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{getORefs} : \mathsf{Tx} \to \mathbb{P}\ (\mathsf{OutputRef}) \\
\mathsf{getORefs}(tx) = \{\ i.\mathsf{outputRef}\ \mid\ i\ \in\ tx.\mathsf{inputs}\ \}
\end{array}
$$

Figure 4 Primitives and basic types for the $\mathrm{EUTxO}_{\mathrm{ma}}$ model.

Now, there are two possibilities, $\pi(tx) = \star$ and $\pi(tx) = \mathsf{toggle}$, for each of which we must prove that $(\star,\ \pi(utxo),\ \pi(tx),\ \pi(utxo'))$ and $\pi(utxo)\ =\ \pi(utxo')$.

$$
\begin{aligned}
&\mathsf{toggleTT}_N \ : \mathsf{OutputRef} \to \mathsf{Script} \to \mathsf{Script}\\
&[\![\mathsf{toggleTT}_N(\mathsf{myRef}, s)]\!](_, (tx,\ pid)) \ := \ \mathsf{myRef} \ \in \ \{i.\mathsf{outputRef} \ | \ i \in tx.\mathsf{inputs}\ \}\\
&\qquad \wedge\ tx.\mathsf{mint} \ = \ \{pid \mapsto \{\mathsf{encode}(s) \mapsto 1\}\}\\
&\qquad \wedge\ \exists\ o \ \in \ tx.\mathsf{outputs},\\
&\qquad\qquad o.\mathsf{value} \ = \ \{pid \mapsto \{\mathsf{encode}(s) \mapsto 1\}\}\\
&\qquad\quad \wedge\ o.\mathsf{validator} \ = \ s\\
&\qquad\quad \wedge\ \mathsf{fromData}_N(o.\mathsf{datum}) \ \neq \ \star
\end{aligned}
$$

Figure 5 TOGGLE thread token minting policy for the naive implementation.

$$
\begin{aligned}
&\mathsf{toggleTT}_D \ : \mathsf{OutputRef} \to \mathsf{Script} \to \mathsf{Script}\\
&[\![\mathsf{toggleTT}_D(\mathsf{myRef},\ s)]\!](_,\ (tx,\ pid)) \ :=\\
&\qquad\quad \mathsf{myRef} \ \in \ \{i.\ \mathsf{outputRef} \ | \ i \in tx.\mathsf{inputs}\ \}\\
&\qquad \wedge \ \ tta \ + \ ttb \ = \ tx.\mathsf{mint}\\
&\qquad \wedge \ \ \exists\ o^a,\ o^b \ \in \ tx.\mathsf{outputs},\\
&\qquad\qquad o^a.\mathsf{value} \ = \ tta \ \wedge \ o^b.\mathsf{value} \ = \ ttb\\
&\qquad\quad \wedge\ o^a.\mathsf{validator} \ = \ o^b.\mathsf{validator} \ = \ s\\
&\qquad\quad \wedge\ \mathsf{fromData}_D(o^a.\mathsf{datum}) \ \neq \ \star \ \wedge \ \mathsf{fromData}_D\ (o^b.\mathsf{datum}) \ \neq \ \star\\
&\textbf{where}\\
&\qquad tta := \{pid \mapsto \{(\mathsf{encode}(s) \ +\!\!+ \ "a") \mapsto 1\}\\
&\qquad ttb := \{pid \mapsto \{(\mathsf{encode}(s) \ +\!\!+ \ "b") \mapsto 1\}
\end{aligned}
$$

Figure 6 TOGGLE thread token minting policy for the distributed implementation.

$$
\begin{aligned}
\mathsf{ttt} &:= \{\mathsf{toggleTT}_N(\mathsf{myRef}) \mapsto \{\mathsf{encode}(\mathsf{toggleVal}_N(\mathsf{myRef})) \mapsto 1\}\}\\
\mathsf{tta} &:= \{\mathsf{toggleTT}_D(\mathsf{myRef}) \mapsto \{(\mathsf{encode}(\mathsf{toggleVal}_D(\mathsf{myRef})) \ +\!\!+ \ "a") \mapsto 1\}\}\\
\mathsf{ttb} &:= \{\mathsf{toggleTT}_D(\mathsf{myRef}) \mapsto \{(\mathsf{encode}(\mathsf{toggleVal}(D\ \mathsf{myRef})) \ +\!\!+ \ "b") \mapsto 1\}\}
\end{aligned}
$$

$$
\pi_n(utxo) := \begin{cases} (a, b) & \text{if} \ \ \mathsf{myRef} \notin \{\ i \ | \ i \mapsto o \in utxo\ \}\\ & \ \wedge\ \exists!\ (i \mapsto o) \ \in \ utxo,\ \mathsf{ttt} \ = \ o.\mathsf{value}\\ & \ \wedge\ o.\mathsf{validator} \ = \ \mathsf{toggleVal}_N(\mathsf{myRef}) \ \wedge \ o.\mathsf{datum} \ = \ (a,\ b)\\ \star & \text{otherwise} \end{cases}
$$

$$
\pi_{\mathsf{Tx},n}(tx) := \begin{cases} \mathsf{toggle} & \text{if } \exists\ i \ \in \ tx.\mathsf{inputs},\ i.\mathsf{output.value} = \mathsf{ttt}\\ \star & \text{otherwise} \end{cases}
$$

Figure 7 TOGGLE thread tokens and naive projections.

Naive implementation.

(i) $\pi(tx) = \star$: We have that $\neg\,(\exists\ i\ \in\ tx.\mathsf{inputs}, i.\mathsf{output}.\mathsf{value} = \mathsf{ttt})$. Since an additional token ttt cannot be minted or burned, we also conclude $\neg\,(\exists\ o\ \in\ tx.\mathsf{outputs},\ o.\mathsf{value} = \mathsf{ttt})$. By $\pi(utxo) = (a, b)$, the $utxo$ state contains a unique output with token ttt, datum (a, b), and $\mathsf{toggleVal}_N(\mathsf{myRef})$ validator. By $\pi(tx) = \star$, that output was not spent, and still exists in the UTxO set $utxo'$. By Assumption 2, since $tx \neq \mathsf{myRef}.\mathsf{fst}$, the reference myRef is not added to the inputs of $utxo'$. So, that $\pi(utxo') = \pi(utxo) = (a, b)$. Then,

$$(\star, \pi(utxo), \pi(tx), \pi(utxo')) = (\star, (a, b), \star, (a, b)) \in \mathsf{TOGGLE}$$

(ii) $\pi(tx) = \mathsf{toggle}$: Implies that $\exists\ i\ \in\ tx.\mathsf{inputs},\ i.\mathsf{output}.\mathsf{value} = \mathsf{ttt}$. This means that that the (unique) UTxO containing ttt is spent, and no ttt tokens are minted or burned. Therefore, the transaction must create a single output in $utxo'$ with that token. The script $\mathsf{toggleVal}_N(\mathsf{myRef})$ must be run because ttt is spent and, by $\pi(utxo) = (a, b)$, was locked by $\mathsf{toggleVal}_N(\mathsf{myRef})$. Because $\mathsf{toggleVal}_N(\mathsf{myRef})$ must validate, the unique new output containing ttt must have a datum (b, a), the same validator. Again, myRef is not added to the inputs of $utxo'$ by assumption. We conclude that $\pi(utxo') = (b, a)$. Then,

$$(\star, \pi(utxo), \pi(tx), \pi(utxo')) = (\star, (a, b), \star, (b, a)) \in \mathsf{TOGGLE}$$

Distributed implementation. The proof for the distributed implementation is similar to the one for the naive implementation, except we must keep track of two inputs and two outputs containing two thread tokens. A transaction updating the state must necessarily spend both outputs containing each of the tokens, and that the new UTxOs containing them are such that the datum in UTxO with token tta now has the boolean that was in the datum of ttb, and vice-versa. Both must still be locked by $\mathsf{toggleVal}_N(\mathsf{myRef})$.