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, 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/LIPIcs...

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.

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 [18] – 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 4.1. It doubles as an executable reference implementation that we can utilize in production for conformance testing. Everything is openly public[1] and developed, much like this paper, as a literate Agda script.

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.[2] 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 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':

$$\Gamma \vdash s \xrightarrow[X]{b} s' \qquad\qquad \frac{\textit{Environments} \quad \textit{States} \quad \textit{(Signals)}}{\textit{Possible transitions}}$$

[1] https://github.com/IntersectMBO/formal-ledger-specifications

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

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, restricted to finite sets only for reasons of decidability. 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. 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*[3] by imposing uniqueness of keys (right):

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

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

with a given signature. 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.

```
SKey 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, 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 we define here.

There is also a special type of address without a payment credential, called a reward address. It cannot be used as an `Addr` but instead it is used to refer to reward accounts [29].

```
Credential = KeyHash ⊎ ScriptHash
```

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

record RwdAddr : Type where
  stake : Credential
```

```
Addr = BaseAddr ⊎ BootstrapAddr
```

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). A `Coin` is the smallest unit of currency, a `Slot` is the smallest number of time (corresponding to 1s in the main chain), and an `Epoch` is a fixed number of slots (corresponding to 5d 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. See [13].

```
Coin = Slot = Epoch = ℕ
```

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 `minfee` function. The two thresholds are collections of rational numbers, which are voting thresholds used in the governance system.

```
record PParams : Type where
  maxBlockSize maxTxSize : ℕ          drepThresholds   : DrepThresholds
  a b                    : ℕ          poolThresholds   : PoolThresholds
```

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 the epoch boundary, 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.

```
record NewEpochEnv : Type where
  field stakeDistrs : StakeDistrs

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

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

record ChainState : Type where
  field newEpochState : NewEpochState
```

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

3.3 Extending the ledger transaction-by-transaction

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

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

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 ≡ᵉ 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) ∪ˡ 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* `Γ ∈ UTxOEnv`*,* `s, s' ∈ UTxOState` *and* `tx ∈ Tx`*, if* `tx .body .txid ∉ map proj₁ (dom (s .UTxOState.utxo))` *and* `Γ ⊢ s ⇀⦇ tx ,UTXO⦈ s'` *then* `getCoin s ≡ getCoin s'`

5 Decentralized Governance

5.1 Entities and actions

The governance framework has three bodies of governance, 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 :
  ────────────────────────────────
  ⟦ gid ⊗ t ⊗ e ⟧ ⊢ s ⇀⦇ NewConstitution dh sh ,ENACT⦈ record s { constitution = (dh , sh) , gid }

Enact-Wdrl :
  let newWdrls = s .withdrawals ∪⁺ wdrl in ∑[ x ← newWdrls ] x ≤ t
  ────────────────────────────────
  ⟦ gid ⊗ t ⊗ e ⟧ ⊢ s ⇀⦇ TreasuryWdrl wdrl ,ENACT⦈ record s { withdrawals = newWdrls }
```

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
    ────────────────────────────────
    (Γ , k) ⊢ s ⇀⦇ sig ,GOV⦈ addVote s aid role cred v

GOV-Propose :
  • actionWellFormed a ≡ true • d ≡ govActionDeposit
    ────────────────────────────────
    (Γ , k) ⊢ s ⇀⦇ inj₂ prop ,GOV⦈ addAction s (govActionLifetime +ᵉ 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.

5.6 Ratification restrictions

```
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 ⟧ᵉ ⊢ 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 ⟧ᵉ ⊢ es ⇀⦇ action ,ENACT⦈ es')))
  ────────────────────────────────
  Γ ⊢ ⟦ es ⊗ removed ⊗ d ⟧ ⇀⦇ a ,RATIFY⦈ ⟦ es ⊗ removed ⊗ d ⟧
```

The main new ingredients for the rules of this 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, and
- `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. Some key ingredients in the transaction body are:

- A set of transaction inputs, each of which identifies an output from a previous transaction. A transaction input consists of a transaction id and an index to uniquely identify the output.
- An indexed collection of transaction outputs. The `TxOut` type is an address paired with a coin value.

- A transaction fee. This value will be added to the fee pot.
- The size and the hash 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.

```
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
```

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 proofs that each and every transition of the previous sections is indeed *computational*:

```
record Computational (_⊢_⇀⦇_,X⦈_ : C → S → Sig → S → Type) : Type where
  field 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[4] 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

---

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

compilation backend.[5] 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.

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 basic properties, *e.g.,* global value preservation as we saw in Section 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 [27].

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 [27, 28] 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).

Outside Cardano, there are very few mechanized results on UTxO-based blockchains (modeled after Bitcoin [18]), and all of them invariably are formulated on a idealized setting [24, 1], abstracting away the complexity that ensues when multiple features interact.

---

[5] https://agda.readthedocs.io/en/v2.6.4/tools/compilers.html#ghc-backend

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, 26] to full-blown formal verification [11, 7]. Notable blockchains that spearhead progress in this direction include Tezos [5, 6, 14], Ziliiqa and its Scilla smart-contract language [23, 22], and Concordium [3, 20, 2, 25, 19]. 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 [21] 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/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 S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/en/bitcoin-paper, October 2008.

19 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.

20 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.

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

22 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.

23 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.

24 Anton Setzer. Modelling Bitcoin in Agda. *CoRR*, abs/1804.06398, 2018. URL: http://arxiv.org/abs/1804.06398, arXiv:1804.06398.

25 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.

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

27 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.

28 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.

29 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 ∷ʳ (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

```
NEWEPOCH-New :
```

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

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