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

Tudor Ferariu
University of Edinburgh, UK

Philip Wadler
University of Edinburgh, UK

Orestis Melkonian
Input Output, United Kingdom

Abstract

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

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

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

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

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

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

Supplementary Material Persistent DOI and Github link for the Agda formalisation:
Software (Zenodo archive): https://doi.org/10.5281/zenodo.15118668
Software (Source Code): https://github.com/tferariu/agda2plinth
archived at swh:1:dir:03dd5b9f802117f3b5ccef0ed56d4214fee41238

1 Introduction

Millions of dollars worth of value are managed by smart contracts. Alas, as made famous by numerous exploits, such contracts often contain flaws. Many such flaws might be eliminated by the application of formal methods, but the costs are high. Hence, most developers make do with property-based tests [49, 23] and auditing [42]. Formal methods might be less expensive to apply if we could identify a few common properties that most or all contracts should share. Such general properties are not widely studied.

Here we make a first step, by presenting three properties we believe generalize to a wide variety of smart contracts. We also introduce a novel technique for proving such properties, based on factoring smart contracts into a specification (written as inference rules) and a

(Author's Version)
Accepted at the 6th International Workshop on Formal Methods for Blockchains (FMBC 2025).

validator (written as a boolean-valued function) and demonstrating their equivalence. We focus on the UTxO-based blockchain Cardano.

We investigate three main properties:

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

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

Our specifications, validators, and the proof of their equivalence, are written using the Agda proof assistant [51]. A proof assistant provides a higher standard of rigour than property-based tests or auditing. We export the Agda code of the implementation to the Cardano target language, Haskell, using agda2hs [35]. Even though agda2hs is not formally verified itself,[1] we still get a lot more confidence that the code used in our proofs aligns with the on-chain code compared to writing the Haskell code by hand. Finally, we test the exported code with an emulator and property-based tests for an extra layer of confirmation.

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

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

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

`https://tferariu.github.io/agda2plinth`

2 Example: simulating accounts on UTxOs

Smart contract development for UTxO-based blockchains has some advantages, as exploits such as double spending, re-entrancy, and replay attacks are not possible (Guggenberger et. al., 2021, [41]). Nonetheless, undesirable behaviour might occur when dealing with the inputs and outputs of a transaction incorrectly.

[1] It is expected that Agda extraction will eventually be formally verified in the future, akin to what MetaCoq [60] has delivered for the more mature ecosystem of the Rocq proof assistant [61].

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

Account-based blockchains such as Tron, EOS, and Tezos [40] are widely used and popular. Chief among them is Ethereum [31], which ranks highest for Total Value Locked (TVL) [7] among contemporary blockchains. Since we are primarily investigating the UTxO-based Cardano blockchain, a simple smart contract simulating accounts serves as a good example for showcasing our properties and proof process.

Chakravarty et. al. (2020, [33]), uses the formalism of state machines [48] to model stateful computation for Cardano validators on the EUTxO ledger. Following in their footsteps, we also use state machines to specify smart contracts and then model their transactions as a state transition system in Section 3.

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

- **Open(pkh)**: Open an account associated with *pkh*
- **Deposit(pkh,v)**: Deposit value v into the account associated with *pkh*
- **Withdraw(pkh,v)**: Withdraw value v from the account associated with *pkh*
- **Transfer(pkh$_1$,pkh$_2$,v)**: Transfer value v from one account to another
- **Close(pkh)**: Close the account associated with *pkh*

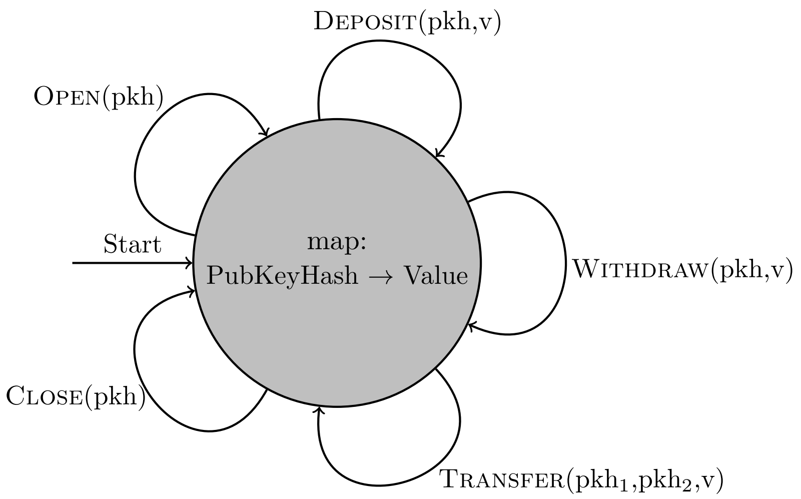


Figure 1 Account Simulation

3 Modelling smart contracts as a State Transition System

As previously mentioned, Cardano validators are Boolean-valued functions. They take as arguments: a Datum; an Input (sometimes called a ”redeemer”), a ScriptContext, and optionally some additional parameters. Since validator scripts are attached to UTxOs, the smart contract itself is an output of a previous transaction, where the Datum is supplied. When we attempt to spend our smart contract in a different transaction, we supply it with an Input, and the blockchain infrastructure supplies the ScriptContext of the transaction. The additional parameters are supplied only when the smart contract is put on the blockchain for the first time, and do not change during runtime.

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

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

Expressing and proving properties for such functions directly is cumbersome at best. Instead, we look for an alternative that is better suited to formal verification. Just like many other papers in the field [64, 32, 40, 31], it makes sense for us to model our contracts as state transition systems. This allows for a much easier way to translate specifications into Agda and prove our desired properties. We define a transition as:

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

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

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

$$\frac{}{P \vdash S \xrightarrow{\varnothing}\!* \; S} \qquad\qquad \frac{P \vdash S \xrightarrow{I} S' \quad P \vdash S' \xrightarrow{Is}\!* \; S''}{P \vdash S \xrightarrow{I::Is}\!* \; S''}$$

4 Smart contract properties

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

4.1 Validity

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

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

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

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

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

If state invariants are a superset of the accessible states, then transition invariants [53] are a superset of the transitive closure of our transition relation, restricted to accessible states. Validity corresponds to a state invariant, but what about transition invariants? Certain properties would indeed take the form of transition invariants when we are concerned with the relationship between states before and after a transition (e.g. after a Withdraw v, the contract value decreases by v). However, this type of property is once again highly contract-specific and does not generalize well.[2] Such properties can easily be expressed and proven in our framework, but are not foundational to our approach.

4.2 Liquidity

Liquidity first appeared in Tsankov et. al (2018) [62] as a property that holds when the contract always admits a trace that decreases its balance. Other papers such as Bartoletti and Zunino (2019, [27]) and Bartoletti et. al. (2024, [25]) also refer to similar properties as Liquidity. Unfortunately, it is trivially easy to permanently lock away your funds in a UTxO

[2] It is also standard practice to include the previous history of transitions in the state (purely as the "ghost" state [52, *auxiliary variables*] that is erased in the runtime), in which case any transition invariant can be expressed as a state invariant.

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

Liquidity. ∀ P and S, if S is **Valid**, then ∃ Is and S' such that $P \vdash S \xrightarrow{Is}* S'$ and $\mathsf{value}(S') = 0$.

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

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

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

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

4.3 Fidelity

When thinking about the DAO bug and its cause, developers focused on the exploit of re-entrancy [37, 55]. Re-entrancy itself is not possible on a UTxO blockchain, but it does have a root cause. The real issue was that the smart contract's internal state did not accurately represent its true balance (Herlihy, 2019, [43]) (Schrans et. al., 2018, [56]). We want to prove that the true value locked by our contract remains equal to the internal representation of value in the code throughout the runtime of the contract.

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

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

Fidelity. ∀ P, S, I, and S', if $\mathsf{value}(S) = \mathsf{internalV}(S)$ and $P \vdash S \xrightarrow{I} S'$, then $\mathsf{value}(S') = \mathsf{internalV}(S')$.

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

5 The validator and agda2hs

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

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

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

6 Equivalence between the model and validator

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

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

This connects a Boolean-valued function f, representing our validator, and a predicate R describing the set inhabited by our state transition system. In this relation, we say that if our validator returns true for an argument, then the same argument can be used to represent a transition from state to state, and vice-versa. This argument a is just a tuple containing the arguments of our validator function. Its type A describes that tuple, which for our validators and state transition system is (Params × Datum × Input × ScriptContext). It is simple to see how our validator can take the place of f and then be applied to a, but how do we turn a state transition system into predicate R? Our states S and S' are derived from the Datum and ScriptContext, both of which are components of the tuple. Once the states are extracted from the argument, the parameters and Input slot directly into our transition relation to give us the desired predicate. This is discussed in more detail within the context of our account simulation example in the next section.

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

7 The full pipeline

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

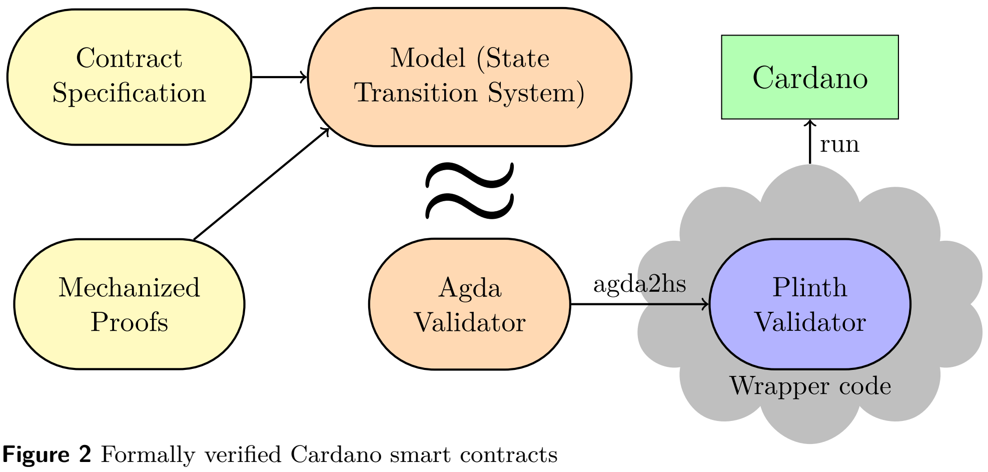


Figure 2 Formally verified Cardano smart contracts

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

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

$$\mathsf{signature}(S') \equiv \mathit{phk} \quad \mathsf{lookup}\ \mathit{phk}\ \mathsf{map}(S) \equiv \mathsf{Nothing}$$

$$\frac{\mathsf{map}(S') \equiv \mathsf{insert}\ \mathit{phk}\ 0\ \mathsf{map}(S) \quad \mathsf{value}(S') \equiv \mathsf{value}(S)}{S \xrightarrow{\textsc{Open}(\mathit{phk})} S'}$$

$$\frac{\begin{array}{c}\mathsf{signature}(S') \equiv \mathit{phk} \quad \mathsf{lookup}\ \mathit{phk}\ \mathsf{map}(S) \equiv \mathsf{Just}\ \mathit{val} \quad v \geq 0 \\ \mathsf{map}(S') \equiv \mathsf{insert}\ \mathit{phk}\ (\mathit{val} + v)\ \mathsf{map}(S) \quad \mathsf{value}(S') \equiv \mathsf{value}(S) + v\end{array}}{S \xrightarrow{\textsc{Deposit}(\mathit{phk},v)} S'}$$

$$\frac{\begin{array}{c}\mathsf{signature}(S') \equiv \mathit{phk} \quad \mathsf{lookup}\ \mathit{phk}\ \mathsf{map}(S) \equiv \mathsf{Just}\ \mathit{val} \quad v \geq 0 \quad \mathit{val} \geq v \\ \mathsf{map}(S') \equiv \mathsf{insert}\ \mathit{phk}\ (\mathit{val} - v)\ \mathsf{map}(S) \quad \mathsf{value}(S') \equiv \mathsf{value}(S) - v\end{array}}{S \xrightarrow{\textsc{Withdraw}(\mathit{phk},v)} S'}$$

$$\frac{\begin{array}{c}\mathsf{signature}(S') \equiv \mathit{phk}_1 \quad \mathsf{lookup}\ \mathit{phk}_1\ \mathsf{map}(S) \equiv \mathsf{Just}\ \mathit{valFrom} \\ \mathsf{lookup}\ \mathit{phk}_2\ \mathsf{map}(S) \equiv \mathsf{Just}\ \mathit{valTo} \quad v \geq 0 \quad \mathit{valFrom} \geq v \quad \mathit{phk}_1 \not\equiv \mathit{phk}_2 \\ \mathsf{map}(S') \equiv \mathsf{insert}\ \mathit{phk}_1\ (\mathit{valFrom} - v)\ (\mathsf{insert}\ \mathit{phk}_2\ (\mathit{valTo} + v)\ \mathsf{map}(S)) \\ \mathsf{value}(S') \equiv \mathsf{value}(S)\end{array}}{S \xrightarrow{\textsc{Transfer}(\mathit{phk}_1,\mathit{phk}_2,v)} S'}$$

$$\frac{\begin{array}{c}\mathsf{signature}(S') \equiv \mathit{phk} \quad \mathsf{lookup}\ \mathit{phk}\ \mathsf{map}(S) \equiv 0 \\ \mathsf{map}(S') \equiv \mathsf{delete}\ \mathit{phk}\ 0\ \mathsf{map}(S) \quad \mathsf{value}(S') \equiv \mathsf{value}(S)\end{array}}{S \xrightarrow{\textsc{Close}(\mathit{phk})} S'}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
Argument = Datum × Input × ScriptContext
```

```
toF : Argument → Bool
toF (d , i , ctx) = agdaValidator d i ctx

toR : Argument → Set
toR (d , i , ctx) = getS d ctx ~[ i ]~> getS' ctx
```

It is now possible to state the desired equivalence:

```
functionalEquiv : toF ≈ toR
```

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

```
transitionImpliesValidator :
  ∀ (l : Datum) (i : Input) (ctx : ScriptContext)
  → getS l ctx ~[ i ]~> getS' ctx
  → agdaValidator l i ctx ≡ true
```

```
validatorImpliesTransition :
  ∀ (l : Datum) (i : Input) (ctx : ScriptContext)
  → agdaValidator l i ctx ≡ true
  → getS l ctx ~[ i ]~> getS' ctx
```

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

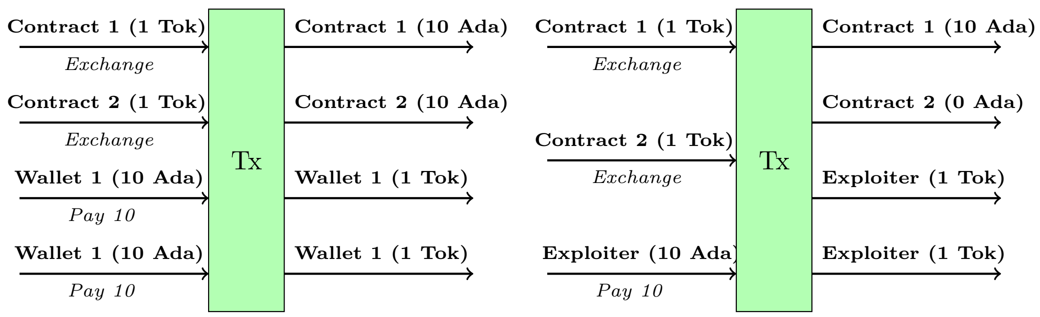
We have also applied this full process to contracts for a multi-signature wallet and order book decentralized exchange, which are described in Appendix A. We omit the proof details for brevity, but they were similar in complexity to the account simulation ones and are fully accessible in the GitHub repository.

8 Double Satisfaction on UTxO

Vinogradova and Melkonian (2025, [63]) define double satisfaction as a vulnerability that occurs for any constraint when two separate contracts impose the same constraint in a transaction (Tx). This is a serious issue which has also been covered in several audits [4, 17] of Cardano smart contracts. Building on their work, we focus on the two main cases where this phenomenon causes problems, namely when the constraints are fulfilled by the same input or output of a transaction.

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

Fidelity gives us exactly this property. If our model expects to receive some funds as part of a transition, this will be reflected in the internal representation. The account simulator only consumes outside inputs for the Deposit action. According to Fidelity, all transitions including Deposit explicitly guarantee that $\mathsf{internalV}(S) = \mathsf{value}(S)$, thus ensuring double satisfaction cannot happen. Consequently, our validator only approves transactions when

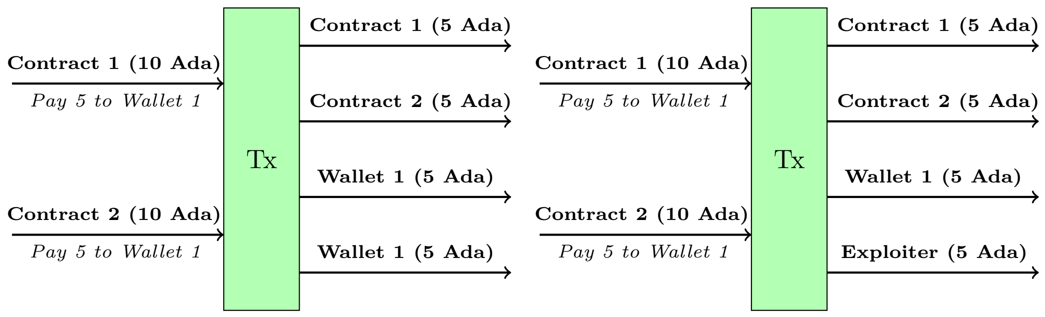


(a) Intended Behaviour **(b)** Erroneous Behaviour

Figure 3 Double satisfaction for inputs

given the necessary amount of currency, but does this without needing to inspect individual inputs. It does not strictly matter where the value comes from as long as the validator receives its required share.

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



(a) Intended Behaviour **(b)** Erroneous Behaviour

Figure 4 Double satisfaction for outputs

This is a much more nuanced issue, as it is unreasonable for a validator to check what all other scripts in the same transaction are doing. Marlowe (Seijas et. al, 2020, [47]) is a domain specific language for Cardano that attempts to solve this issue by limiting what scripts may run alongside each other. While this technically solves the issue, it is also very limiting and can potentially make submitting transactions cumbersome. Message-passing (Hoare, 1987, [45])(Vinogradova and Melkonian, 2025, [63]) can be used to deal with double satisfaction for both inputs and outputs, but this will have to be the subject of future work.

9 Related work

Anton Setzer's work **modelling Bitcoin in Agda** [59] offers a different approach, by formalizing transactions on the ledger not as state transitions, but by using transaction trees. Many other blockchains have also been spearheading the use of fully mechanized formalizations using proof assistants. **Scilla** [58] is the high-level smart contract language developed for the Zilliqa [57] blockchain. It is particularly significant because it also uses a state transition system to define contracts. Scilla is formalized in Coq, and attempts to address security and correctness issues via formal verification. A similar approach using Coq is also applied by both the Tezos [29] and Concordium [22] blockchains for smart contract certification and verification.

The **K Framework** [54] is a robust tool for language semantics that offers a structured approach to specifying the semantics of programming languages. It has been used to specify the formal semantics of the Ethereum virtual machine [44], and is of great interest for specifying smart contracts in general, as well as their properties. While our proposed properties generalize well to different contracts, the verification pipeline specifically targets the Cardano ecosystem. The K Framework could potentially be used to bridge this gap.

Tools such as **Certora** [28, 5] for Solidity [16] and the **Move Prover** [65, 38] for Move [30] and the Diem Blockchain [6] attempt to automate the process by leveraging SMT-based proof techniques. This generally involves compiling contracts and their associated properties into a logical formula, which can then be sent to an SMT solver. When compared to our proposed method, this approach has the potential to significantly reduce the manual work required, but offers fewer guarantees than full formal verification and poses the risk of significantly limiting the scope of provable properties.

Smart Code Verifier [15] is a formal verification tool currently under development for the Cardano blockchain that attempts to bridge the gap between the two approaches. Smart contract code (written in Aiken [1], Plinth, etc.) is annotated with specifications in a dedicated language. The annotations and source code are translated into Lean4 [50] proof obligations. These obligations are used to generate an SMTLib [24] formula which is then automatically discharged using Z3 [36]. Any proofs that the solver cannot complete automatically can instead be solved manually, and counterexamples are generated when specifications are violated.

10 Future Work

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

For the validator implementation on the side of Agda, we use several layers of abstraction instead of using the exact same types as the blockchain to simplify proof efforts. This decision was made to keep the proof of concept for this process as simple as possible. That being said, a **formalized Cardano Ledger** [46] already exists in Agda. Optimally, we would fully integrate our method with the ledger model to be as close as possible to the target language of Plinth. It is currently unclear how compatible the existing formalisation is with our current approach, so more investigation will be necessary.

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

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

11 Conclusion

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

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

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

References

1 Aiken. https://aiken-lang.org/. Accessed: 2025-03-25.

2 Aku's nightmare: $34m locked forever as flaw highlights danger of smart contracts. https://www.pymnts.com/blockchain/2022/akus-nightmare-34m-locked-forever-as-flaw-highlights-danger-of-smart-contracts/. Accessed: 2025-02-02.

3 Cardano node emulator. https://github.com/IntersectMBO/cardano-node-emulator. Accessed: 2025-02-02.

4 Cardano vulnerabilities #1 — double satisfaction. https://medium.com/@vacuumlabs_auditing/cardano-vulnerabilities-1-double-satisfaction-219f1bc9665e. Accessed: 2025-02-02.

5 Certora white paper. https://www.certora.com/blog/white-paper. Accessed: 2025-03-25.

6 The diem blockchain. https://developers.diem.com/docs/technical-papers/the-diem-blockchain-paper/. Accessed: 2025-03-25.

7 Largest blockchains in crypto ranked by TVL. https://coinmarketcap.com/chain-ranking/. Accessed: 2025-02-02.

8 Lessons from the wormhole exploit: Smart contract vulnerabilities introduce risk. https://www.chainalysis.com/blog/wormhole-hack-february-2022/. Accessed: 2025-02-02.

9 NFT project Aku dreams loses $34 million to smart contract flaw. https://bitcoinist.com/nft-project-aku-dreams-loses-34-million-to-smart-contract-flaw/. Accessed: 2025-02-02.

10 Over $1 million permanently locked in defi smart contract. https://coingeek.com/over-1-million-permanently-locked-in-defi-smart-contract/. Accessed: 2025-02-02.

11 Over $1m in user funds on compound fork perfect finance are frozen after code change error. https://www.theblock.co/linked/83792/user-funds-perfect-finance-frozen-code-error. Accessed: 2025-02-02.

12 Plinth and Plutus documentation. https://plutus.cardano.intersectmbo.org/docs/. Accessed: 2025-02-02.

13 Plutus Github. https://github.com/IntersectMBO/plutus. Accessed: 2025-02-02.

14 QuickCheck-dynamic. https://github.com/input-output-hk/quickcheck-dynamic. Accessed: 2025-02-02.

15 Smart code verifier. https://github.com/input-output-hk/smartcode-verifier. Accessed: 2025-03-25.

16 Solidity documentation. https://app.readthedocs.org/projects/solidity/downloads/pdf/develop/. Accessed: 2025-03-25.

17 Tweag technical review of marlowe. https://github.com/tweag/tweag-audit-reports/blob/main/Marlowe-2023-03.pdf. Accessed: 2025-02-02.

18 What is a DEX? https://www.coinbase.com/en-gb/learn/crypto-basics/what-is-a-dex. Accessed: 2025-02-02.

19 What is a DEX (decentralized exchange)? https://chain.link/education-hub/what-is-decentralized-exchange-dex. Accessed: 2025-02-02.

20 Wormhole cryptocurrency platform hacked for $325 million after error on GitHub. https://www.theverge.com/2022/2/3/22916111/wormhole-hack-github-error-325-million-theft-ethereum-solana. Accessed: 2025-02-02.

21 Bowen Alpern and Fred B Schneider. Defining liveness. *Information processing letters*, 21(4):181–185, 1985.

22 Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. Concert: a smart contract certification framework in coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 215–228, 2020.

23 Morena Barboni, Andrea Morichetta, and Andrea Polini. Smart contract testing: challenges and opportunities. In *Proceedings of the 5th International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 21–24, 2022.

24 Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, page 14, 2010.

25 Massimo Bartoletti, Angelo Ferrando, Enrico Lipparini, and Vadim Malvone. Solvent: liquidity verification of smart contracts. In *International Conference on Integrated Formal Methods*, pages 256–266. Springer, 2024.

26 Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. A theory of transaction parallelism in blockchains. *Logical Methods in Computer Science*, 17, 2021.

27 Massimo Bartoletti and Roberto Zunino. Verifying liquidity of bitcoin contracts. In *International Conference on Principles of Security and Trust*, pages 222–247. Springer, 2019.

28 Thomas Bernardi, Nurit Dor, Anastasia Fedotov, Shelly Grossman, Neil Immerman, Daniel Jackson, Alexander Nutz, Lior Oppenheim, Or Pistiner, Noam Rinetzky, et al. Wip: Finding bugs automatically in smart contracts with parameterized invariants. *Retrieved July*, 14:2020, 2020.

29 Bruno Bernardo, Raphaël Cauderlier, Guillaume Claret, Arvid Jakobsson, Basile Pesin, and Julien Tesson. Making tezos smart contracts more reliable with coq. In *International Symposium on Leveraging Applications of Formal Methods*, pages 60–72. Springer, 2020.

30 Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario Russi Rain, Stephane Sezer, et al. Move: A language with programmable resources. *Libra Assoc*, 1, 2019.

31 Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3(37):2–1, 2014.

32 Manuel MT Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann Müller, Michael Peyton Jones, Polina Vinogradova, and Philip Wadler. Native custom tokens in the extended UTXO model. In *International Symposium on Leveraging Applications of Formal Methods*, pages 89–111. Springer, 2020.

33 Manuel MT Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The extended UTXO model. In *Financial Cryptography and Data Security: FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers 24*, pages 525–539. Springer, 2020.

34 Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, 2000.

35 Jesper Cockx, Orestis Melkonian, Lucas Escot, James Chapman, and Ulf Norell. Reasonable Agda is correct Haskell: writing verified haskell using agda2hs. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*, pages 108–122, 2022.

36 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

37 Vikram Dhillon, David Metcalf, Max Hooper, Vikram Dhillon, David Metcalf, and Max Hooper. The DAO hacked. *blockchain enabled applications: Understand the blockchain Ecosystem and How to Make it work for you*, pages 67–78, 2017.

38 David Dill, Wolfgang Grieskamp, Junkil Park, Shaz Qadeer, Meng Xu, and Emma Zhong. Fast and reliable formal verification of smart contracts with the move prover. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 183–200. Springer, 2022.

39 Tudor Ferariu and Orestis Melkonian. agda2plinth: Formal verification of Cardano smart contracts in Agda. https://github.com/tferariu/agda2plinth, March 2025. doi:10.5281/zenodo.15118668.

40 LM Goodman. Tezos—a self-amending crypto-ledger white paper. *URL: https://www. tezos. com/static/papers/white paper. pdf*, 4:1432–1465, 2014.

41 Tobias Guggenberger, Vincent Schlatt, Jonathan Schmid, and Nils Urbach. A structured overview of attacks on blockchain systems. *PACIS*, page 100, 2021.

42 Daojing He, Zhi Deng, Yuxing Zhang, Sammy Chan, Yao Cheng, and Nadra Guizani. Smart contract vulnerability analysis and security audit. *IEEE Network*, 34(5):276–282, 2020. doi:10.1109/MNET.001.1900656.

43 Maurice Herlihy. Blockchains from a distributed computing perspective. *Communications of the ACM*, 62(2):78–85, 2019.

44 Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. KEVM: A complete formal semantics of the Ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217. IEEE, 2018.

45 Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

46 Andre Knispel, Orestis Melkonian, James Chapman, Alasdair Hill, Joosep Jääger, William DeMeo, and Ulf Norell. Formal specification of the Cardano blockchain ledger, mechanized in Agda. In *5th International Workshop on Formal Methods for Blockchains (FMBC 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.

47 Pablo Lamela Seijas, Alexander Nemish, David Smith, and Simon Thompson. Marlowe: implementing and analysing financial contracts on blockchain. In *Financial Cryptography and Data Security: FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers 24*, pages 496–511. Springer, 2020.

48 George H Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955.

49 Mikkel Milo, Eske Hoy Nielsen, Danil Annenkov, and Bas Spitters. Finding smart contract vulnerabilities with concert's property-based testing framework. *arXiv preprint arXiv:2208.00758*, 2022.

50 Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*, pages 625–635. Springer, 2021.

51 Ulf Norell. Dependently typed programming in Agda. In *International School on Advanced Functional Programming*, pages 230–266. Springer, 2008.

52 Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976. doi:10.1145/360051.360224.

53 Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.*, pages 32–41. IEEE, 2004.

54 Grigore Roșu and Traian Florin Șerbănută. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

55 Francisco Santos and Vasileios Kostakis. The DAO: a million dollar lesson in blockchain governance. *School of Business and Governance, Ragnar Nurkse Department of Innovation and Governance*, 2018.

56 Franklin Schrans, Susan Eisenbach, and Sophia Drossopoulou. Writing safe smart contracts in Flint. In *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming*, pages 218–219, 2018.

57 A Secure. The zilliqa project: A secure, scalable blockchain platform. 2018.

58 Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. Safer smart contract programming with Scilla. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019.

59 Anton Setzer. Modelling bitcoin in agda. *arXiv preprint arXiv:1804.06398*, 2018.

60 Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq project. *J. Autom. Reason.*, 64(5):947–999, 2020. URL: https://doi.org/10.1007/s10817-019-09540-0, doi:10.1007/S10817-019-09540-0.

61 The Coq Development Team. The Coq proof assistant, September 2024. doi:10.5281/zenodo.14542673.

62 Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 67–82, 2018.

63 Polina Vinogradova and Orestis Melkonian. Message-passing in the extended UTxO ledger. In Jurlind Budurushi, Oksana Kulyk, Sarah Allen, Theo Diamandis, Ariah Klages-Mundt, Andrea Bracciali, Geoffrey Goodell, and Shin'ichiro Matsuo, editors, *Financial Cryptography and Data Security. FC 2024 International Workshops*, pages 150–169, Cham, 2025. Springer Nature Switzerland.

64 Polina Vinogradova, Orestis Melkonian, Philip Wadler, Manuel Chakravarty, Jacco Krijnen, Michael Peyton Jones, James Chapman, and Tudor Ferariu. Structured contracts in the EUTxO ledger model. In *5th International Workshop on Formal Methods for Blockchains (FMBC 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.

65 Jingyi Emma Zhong, Kevin Cheang, Shaz Qadeer, Wolfgang Grieskamp, Sam Blackshear, Junkil Park, Yoni Zohar, Clark Barrett, and David L Dill. The move prover. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I 32*, pages 137–150. Springer, 2020.

A Additional example contracts

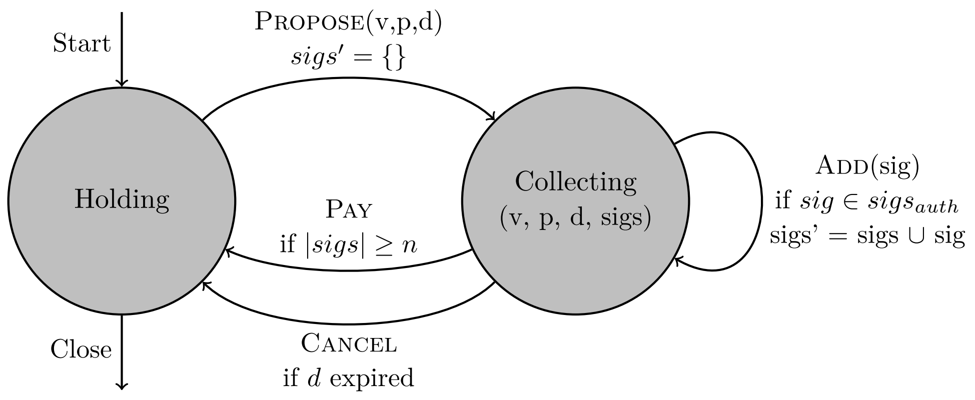


Figure 5 Multi-Signature Wallet

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

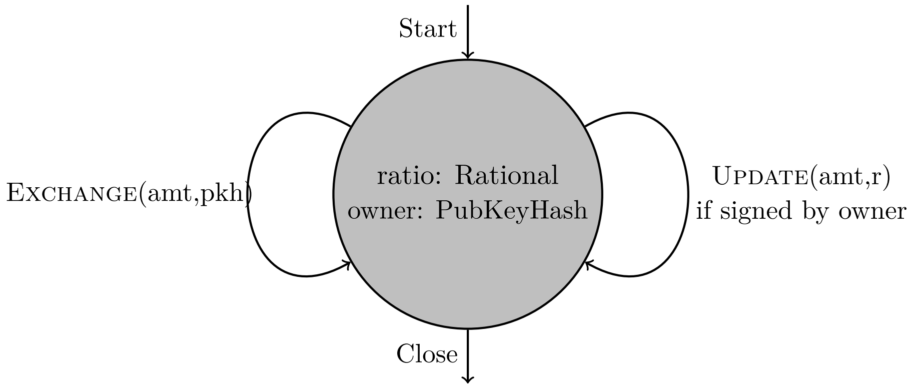


Figure 6 Decentralized Exchange

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

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

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

B State transition system for account simulation

```
-- Single-step transition
data _~[_]~>_ : State → Input → State → Set where

  TOpen : ∀ {pkh s s'}
    → pkh ≡ s .context .tsig
    → lookup pkh (s .datum) ≡ Nothing
    → s' .datum ≡ insert pkh 0 (s .datum)
    → s .context .value ≡ s .context .value
    ----------------------------------------
    → s ~[ (Open pkh) ]~> s'

  TClose : ∀ {pkh s s'}
    → pkh ≡ s' .context .tsig
    → lookup pkh (s .datum) ≡ Just 0
    → s' .datum ≡ delete pkh (s .datum)
    → s' .context .value ≡ s .context .value
    ----------------------------------------
    → s ~[ (Close pkh) ]~> s'

  TWithdraw : ∀ {pkh val s s' v}
    → pkh ≡ s' .context .tsig
    → lookup pkh (s .datum) ≡ Just v
    → val ≥ emptyValue
    → v ≥ val
    → s' .datum ≡ insert pkh (v - val) (s .datum)
    → s' .context .value ≡ s .context .value - val
    ----------------------------------------------
    → s ~[ (Withdraw pkh val) ]~> s'

  TDeposit : ∀ {pkh val s s' v}
    → pkh ≡ s' .context .tsig
    → lookup pkh (s .datum) ≡ Just v
    → val ≥ emptyValue
    → s' .datum ≡ insert pkh (v + val) (s .datum)
    → s' .context .value ≡ s .context .value + val
    ----------------------------------------------
    → s ~[ (Deposit pkh val) ]~> s'

  TTransfer : ∀ {from to val s s' vF vT}
    → from ≡ s' .context .tsig
    → lookup from (s .datum) ≡ Just vF
    → lookup to (s .datum) ≡ Just vT
    → vF ≥ val
    → val ≥ emptyValue
    → from ≢ to
    → s' .datum ≡ insert from (vF - val) (insert to (vT + val) (s .datum))
    → s' .context .value ≡ s .context .value
    -------------------------------------------------------------------
    → s ~[ (Transfer from to val) ]~> s'

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

  root : ∀ { s }
    ---------------
    → s ~[ [] ]~* s

  cons : ∀ { s s' s'' i is }
    → s  ~[ i       ]~> s'
    → s' ~[ is      ]~* s''
    -------------------------
    → s  ~[ (i ∷ is) ]~* s''
```

C Validator code for account simulation

C.1 Original Agda validator

```
agdaValidator : Datum → Input → ScriptContext → Bool
agdaValidator dat inp ctx = case inp of λ where
  (Open pkh)             → checkSigned pkh ctx && not (checkMembership (lookup pkh dat)) &&
                           newDatum ctx == insert pkh 0 dat && newValue ctx == oldValue ctx
  (Close pkh)            → checkSigned pkh ctx && checkEmpty (lookup pkh dat) &&
                           newDatum ctx == delete pkh dat && newValue ctx == oldValue ctx
  (Withdraw pkh val)     → checkSigned pkh ctx && checkWithdraw (lookup pkh dat) pkh val dat ctx &&
                           newValue ctx == oldValue ctx - val
  (Deposit pkh val)      → checkSigned pkh ctx && checkDeposit (lookup pkh dat) pkh val dat ctx &&
                           newValue ctx == oldValue ctx + val
  (Transfer from to val) → checkSigned from ctx &&
                           checkTransfer (lookup from dat) (lookup to dat) from to val dat ctx &&
                           newValue ctx == oldValue ctx
{-# COMPILE AGDA2HS agdaValidator #-}
```

C.2 Extracted Haskell validator

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