

# Structured Contracts in the EUTxO Ledger Model

**Polina Vinogradova** ✉ 

Input Output, Global


**Philip Wadler** ✉ 

Input Output, Global

University of Edinburgh, UK

**Jacco Krijnen** ✉ 


Utrecht University, Netherlands

**James Chapman** ✉ 

Input Output, Global

**Orestis Melkonian** ✉ 

Input Output, Global

**Manuel Chakravarty** ✉ 

Input Output, Global

**Michael Peyton Jones** ✉ 

Input Output, Global

**Tudor Ferariu** ✉ 

University of Edinburgh, UK

---

## Abstract

Blockchain ledgers based on the *extended* UTxO model support fully expressive smart contracts to specify permissions for performing certain actions, such as spending transaction outputs or minting assets. There have been some attempts to standardize the implementation of stateful programs using this infrastructure, with varying degrees of success.

To remedy this, we introduce the framework of *structured contracts* to formalize what it means for a stateful program to be correctly implemented on the ledger. Using small-step semantics, our approach relates low-level ledger transitions to high-level transitions of the smart contract being specified, thus allowing users to prove that their abstract specification is adequately realized on the blockchain. We argue that the framework is versatile enough to cover a range of examples, in particular proving the equivalence of multiple concrete implementations of the same abstract specification.

Building upon prior meta-theoretical results, our results have been mechanized in the Agda proof assistant, paving the way to rigorous verification of smart contracts.

**2012 ACM Subject Classification** Theory of computation → Program specifications; Security and privacy → Formal methods and theory of security

**Keywords and phrases** blockchain, ledger, smart contract, formal verification, specification, transition systems, Agda, UTxO, EUTxO, small-step semantics

Digital Object Identifier [10.4230/OASICS.FMBC.2024.4](https://doi.org/10.4230/OASICS.FMBC.2024.4)

## 1 Introduction

Many modern cryptocurrency blockchains are smart contract-enabled, meaning that they provide support for executing user-defined code as part of block or transaction processing. This code is used to specify agreements between untrusted parties that can be automatically enforced without a trusted intermediary. Examples of such contracts may include distributed exchanges (DEXs), escrow contracts, auctions, etc.

There is a lot of variation in the details of how smart contract support is implemented across different platforms. On account-based platforms such as Ethereum [6] and Tezos [17], smart contracts are inherently stateful and their states can be updated by transactions. Smart contracts in the extended UTxO (EUTxO) model, such as Cardano [20] and Ergo [13], on the other hand, take the form of boolean predicates on the transaction data and are inherently stateless. In this model, transactions specify all the changes being done to the ledger state, while contract predicates are used only to specify permissions for performing the UTxO set updates specified by the transaction, such as spending UTxOs or minting tokens.

Functional EUTxO programming is a less conventional paradigm than the stateful contracts of the account-based model [30, 14]. However, it has a notable advantage: the



© Polina Vinogradova;

licensed under Creative Commons License CC-BY 4.0

5th International Workshop on Formal Methods for Blockchains (FMBC 2024).

Editors: Bruno Bernardo and Diego Marmosler; Article No. 4; pp. 4:1–4:19



OpenAccess Series in Informatics

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

## 4:2 Structured Contracts in the EUTxO Ledger Model

changes made by a transaction applied to the ledger are predictable, including the outputs of the contracts that it runs on-chain [20, 2]. This is because the data inspected by the executed contracts, the exact cost of on-chain contract execution, and UTxO set changes, are all fixed at the time of transaction construction. The unpredictability of transaction application and contract execution outcomes results in vulnerabilities in account-based models, such as the reentrancy or replay attacks [18, 17], which do not exist in the EUTxO model.

Like many prominent platforms [17, 6, 24, 31, 29], the Cardano implementation of the EUTxO ledger [20] is specified as a transition system. The reason for this design choice is that the evolution of the ledger takes place in atomic steps corresponding to the application of a single transaction. What sets the Cardano specification apart, however, is the formal rigor of its operational small-step semantics specification [12].

Many common contract applications require a model of stateful computation. There are multiple existing approaches to implementing and verifying specific designs of stateful programs running on the ledger [7, 15, 10], however, there are currently no principled standard practices for doing so. We propose the *structured contract framework* (SCF) as an extension of this approach to contract specification. It enables users to instantiate a small-step program specification that runs on the ledger via the use of smart contract scripts.

Generalizing the constraint-emitting-machine design pattern introduced in the seminal EUTxO paper [9] to establish a correspondence between high-level abstract state machines and low-level transactions, SCF formalizes the notion of stateful program running on the EUTxO ledger, and what it means for it to be implemented *correctly*. We do so by requiring instantiation of a stateful program to include a proof of a *simulation relation* between its specification and the ledger specification. Our generalization allows formalizing invariants (a.k.a. safety properties) of contracts for which it was not previously done in a uniform way. For example, we can express invariants of stateful contracts with state distributed across multiple UTxOs, as well as on the totality of tokens under a specific policy.

The class of contracts which can be instantiated in SCF is made up of all stateful contracts with implementations for which valid ledger evolution guarantees that the evolution of the on-chain contract state adheres to its specification. We argue that SCF constitutes a novel principled approach to stateful smart contract architecture that is amenable to formal analysis and suitable for a wide range of smart contract applications. The main contributions of this paper are:

- (i) a formulation of the structured contract framework (SCF) on top of a simplified small-step semantics for EUTxO ledgers;
- (ii) a case study expressing the minting policy of a single NFT as a structured contract;
- (iii) a case study demonstrating the use of SCF to define two distinct ledger implementations of a single specification, including one that is distributed across multiple UTxO entries and interacting scripts.

We have mechanized our results in the Agda proof assistant [25], which are publicly available in HTML format:

<https://omelkonian.github.io/structured-contracts/>

In the future, we hope to integrate this framework into the existing Agda specification of Cardano’s small-step ledger semantics.<sup>1</sup>

---

<sup>1</sup> <https://github.com/IntersectMBO/formal-ledger-specifications>

## 2 EUTxO ledger model

The EUTxO ledger model is a UTxO-based ledger model that supports the use of user-defined Turing-complete scripts to specify conditions for spending (consuming) UTxO entries as well as token minting and burning policies. The EUTxO model has been previously expressed in terms of a ledger state containing a list of transactions that have been validated, and a set of rules for validating incoming transactions [7]. We demonstrate here that it can be expressed as a labeled transition system with the UTxO set as its state, specified in small-step semantics, similar to the specification of the deployed Cardano ledger [20]. The transaction validation rules of the existing model are interpreted as constraints of the UTxO state transition rule in our model. Note that while in a realistic system transactions are applied to the ledger in blocks, here we abstract away block structure for simplicity.

**Specifying transition relations.** For some  $\text{Env}, \text{State}$  and  $\text{Input}$ , the transition relation  $\text{TRANS} \subseteq \text{Env} \times \text{State} \times \text{Input} \times \text{State}$  contains 4-tuples  $(e, s, i, s') \in \text{TRANS}$  where  $e \in \text{Env}$  is the *environment*,  $s \in \text{State}$  is the *start state*,  $i \in \text{Input}$  is the *input*, and  $s' \in \text{State}$  is the *end state*.

A specification  $\text{TRANS}$  consists of one or more *transition rules*. The only 4-tuples that are members of  $\text{TRANS}$  are those that satisfy the preconditions of one of its transition rules. By convention, all variables that appear unbound in a given rule are universally quantified, unless they are bound by an explicit existential ( $\exists$ ) or let-binding ( $:=$ ). In the context of specifying transition rules, membership in  $\text{TRANS}$  is also denoted by the following notation:

$$e \vdash s \xrightarrow[\text{TRANS}]{i} s'$$

**Input, environment, and labeled transition systems.** The input and the environment together are used to calculate the possible end state(s) for a given start state, making up the *label* of the transition between the start and end states. If the transition relation is functional, there is exactly one end state for a given start state and label. We adopt the conventional distinction between environment and input due to its usefulness in the blockchain context [12]. In particular, input comes from users, e.g. transactions they submit. The environment, on the other hand, is outside the user's control, such as the blockchain time.

### 2.1 Ledger types

The ledger types and rules that we base this work on are, for the most part, similar to those presented in existing EUTxO ledger research [7]. We make some simplifications in order to remove details not relevant to this work. We give an overview of these for completeness, and clarify any omitted types in Appendix A. Notation we use that is outside conventional set-theoretic notation is listed in Figure 3, and explained in the text. Here,  $\mathbb{B}, \mathbb{N}, \mathbb{Z}$  denote the type of Booleans, natural numbers, and integers, respectively. Some types described below are mutually recursive, so there is no natural order in which to describe them.

**Value.** We define  $\text{Value} := \text{PolicyID} \mapsto (\text{TokenName} \mapsto \text{Quantity})$  as a nested map structure, closely following the original formulation as finitely supported functions [7]. A term of this type is a *bundle* of multiple kinds of assets. The type of an identifier of a class of fungible assets is given by  $\text{AssetID} := \text{PolicyID} \times \text{TokenName}$ .  $\text{Quantity} := \mathbb{Z}$  is an integral value, which can be negative in the case of indicating quantities of tokens to be burned. For a given  $v \in \text{Value}$ , the nested map associates a quantity to each asset ID. The quantities of all assets with IDs not included in  $v$  are implicitly 0.  $\text{Value}$  forms a partial order, as well as a group with addition (+) and the empty map ( $\emptyset$ ) as the zero [8]. The components of  $\text{AssetID}$  are:

## 4:4 Structured Contracts in the EUTxO Ledger Model

- (i) a script of type  $\text{PolicyID} := \text{Script}$ , which is executed any time a transaction is minting assets with this minting policy. If it validates, the transaction is allowed to mint the assets under this policy (specified in the `mint` field of the transaction), meaning that these assets are added to the the total amount of assets a transaction is transferring (see (v) and (ix) in Section 2.2);
- (ii) a  $\text{TokenName} := [\text{Char}]$ , which is a character list specified by the user at the time of minting transaction construction, and is used to differentiate assets under the same minting policy. A minting policy may condition on token names.

**UTxO set.** The type of the UTxO set is a finite key-value map  $\text{UTxO} := \text{OutputRef} \mapsto \text{Output}$ . The type of the key of this map is  $\text{OutputRef} := \text{Tx} \times \text{Ix}$ , with  $\text{Ix} := \mathbb{N}$ . A  $(tx, ix) \in \text{OutputRef}$  is called an output reference. It consists of a transaction  $tx$  that created the output to which it points, and index  $ix$ , which is the location of particular output in the list of outputs of that transaction. The pair uniquely identifies a transaction output. In practice, an injective function, which encodes a transaction as a natural number, is applied to a  $tx$  for inclusion in an output reference, so that  $\text{OutputRef}$  is the type  $\mathbb{N} \times \text{Ix}$ . However, we omit this for simplicity and readability.

An output  $(a, v, d) \in \text{Output} := \text{Script} \times \text{Value} \times \text{Datum}$  consists of (i) a script address  $a$  (field validator), which is run when the output is spent, (ii) an asset bundle  $v$  (field value), and (iii) a datum  $d$  (field datum), which is some additional data.

**Data.** Data is a type used for representing data encoded in a specific way. It is similar in structure to a CBOR encoding, c.f. the relevant Agda definitions<sup>2</sup> accompanying the seminal EUTxO papers [9, 7]. Data of this type is passed as arguments to scripts. The types  $\text{Datum} := \text{Data}$  and  $\text{Redeemer} := \text{Data}$  are both synonyms for the  $\text{Data}$  type. Conversion functions are required in order for a script to interpret  $\text{Data}$ -type inputs as the datatypes it is expecting. When the context is clear, the decoding function is called `fromData`, and the encoding one is `toData`.

**Slot number.** A slot number  $slot \in \text{Slot} := \mathbb{N}$  is a natural number used to represent the time at which a transaction is processed.

**Transactions.** The data structure  $\text{Tx}$  specifies a set of updates to the UTxO set. A transaction  $tx \in \text{Tx}$  contains (i) a set  $tx.inputs \in \text{OutputRef} \times \text{Output} \times \text{Redeemer}$  of *inputs* each referencing entries in the UTxO set that the transaction is removing (spending), with their corresponding redeemers, (ii) a list of outputs  $tx.outputs$ , which get entered into the UTxO set with the appropriately generated output references, (iii) a pair of slot numbers  $tx.validityInterval$  representing the validity interval of the transaction, (iv) a  $tx.mint \in \text{Value}$  being minted by the transaction, (v) a redeemer for each of the minting policies being executed  $tx.mintRdmrs \in \text{Script} \mapsto \text{Redeemer}$ , and (vi) the map  $tx.sigs$  of (public) keys that signed the transaction, paired with their signatures.

**Scripts.** A smart contract, or *script*, is a piece of stateless user-defined code with a boolean output, and has the (opaque) type  $\text{Script}$ . Scripts are associated with performing a specific action, such as spending an output, or minting assets. If a transaction attempts to perform an action associated with a script, that script is executed during transaction validation, and must return `true` (validate) given certain inputs. A script specifies the conditions a transaction must satisfy in order to be allowed to perform the associated action. We do not specify the language in which scripts are written, but we presume Turing-completeness. We write script pseudocode using set-theoretic notation.

---

<sup>2</sup> <https://omelkonian.github.io/formal-utxo/UTxO.Types.html#DATA>

The input to a script consists of (i) a summary of transaction data, (ii) a pointer to the specific action (within the transaction) for which the script is specifying the permission, (iii) and a piece of user-defined data we call a Redeemer. A redeemer is defined at the time of transaction construction (by the transaction author) for each action requiring a script to be run. Evaluating a minting policy script  $s$  to validate minting tokens under policy  $p$ , run by transaction  $tx$  with redeemer  $r$ , is denoted by  $\llbracket s \rrbracket(r, (tx, p))$ . To evaluate a script  $q$ , which validates spending input  $i \in tx.inputs$  with datum  $d$  and redeemer  $r$ , we write  $\llbracket q \rrbracket(d, r, (tx, i))$ .

## 2.2 Ledger transition semantics

Permissible updates to the UTxO set are given by the transition system  $\text{LEDGER} \subseteq \text{Slot} \times \text{UTxO} \times \text{Tx} \times \text{UTxO}$ . The output of the function  $\text{checkTx} : \text{Slot} \times \text{UTxO} \times \text{Tx} \rightarrow \mathbb{B}$  determines whether a transaction is valid in a given state and environment. The output of  $\text{checkTx}(\text{slot}, \text{utxo}, tx)$  is given by the conjunction of the following checks, which are consistent with the previously specified EUTxO validation rules [7]:

(i) **The transaction has at least one input:**

$$tx.inputs \neq \{\}$$

(ii) **The current slot is within transaction validity interval:**

$$\text{slot} \in tx.validityInterval$$

(iii) **All outputs have positive values:**

$$\forall o \in tx.outputs, o.value > \emptyset$$

(iv) **All output references of transaction inputs exist in the UTxO:**

$$\forall (oRef, o) \in \{(i.outputRef, i.output) \mid i \in tx.inputs\}, (oRef \mapsto o) \in utxo$$

(v) **Value is preserved:**

$$tx.mint + \sum_{i \in tx.inputs, (i.outputRef \mapsto o) \in utxo} o.value = \sum_{o \in tx.outputs} o.value$$

(vi) **No output is double-spent:**

$$\forall i, j \in tx.inputs, i.outputRef = j.outputRef \Rightarrow i = j$$

(vii) **All inputs validate:**

$$\forall (i, o, r) \in tx.inputs, \llbracket o.validator \rrbracket(o.datum, r, (tx, (i, o, r))) = \text{true}$$

(viii) **Minting redeemers are present:**

$$\forall (s \mapsto \_) \in tx.mint, \exists r, (s, r) \in tx.mintRdmrs$$

(ix) **All minting scripts validate:**

$$\forall (p, r) \in tx.mintRdmrs, \llbracket p \rrbracket(r, (tx, p)) = \text{true}$$

## 4:6 Structured Contracts in the EUTxO Ledger Model

(x) All signatures are correct:

$$\forall (pk \mapsto s) \in tx.sigs, \text{checkSig}(tx, pk, s) = \text{true}$$

Membership in the LEDGER set is defined using  $\text{checkTx}$ . The single rule defining LEDGER, called  $\text{ApplyTx}$ , states that  $(slot, utxo, tx, utxo') \in \text{LEDGER}$  whenever  $\text{checkTx}(slot, utxo, tx)$  holds and  $utxo'$  is given by  $\{ i \mapsto o \in utxo \mid i \notin \text{getORefs}(tx) \} \cup \text{mkOuts}(tx)$ .

$$utxo' := \{ i \mapsto o \in utxo \mid i \notin \text{getORefs}(tx) \} \cup \text{mkOuts}(tx)$$

$$\text{checkTx}(slot, utxo, tx)$$

$$\text{ApplyTx} \frac{}{slot \vdash ( utxo ) \xrightarrow[\text{LEDGER}]{tx} ( utxo' )}$$

The value  $utxo'$  is calculated by removing the UTxO entries in  $utxo$  corresponding to the output references of the transaction inputs, and adding the outputs of the transaction  $tx$  to the UTxO set with correctly generated output references. The function  $\text{getORefs}$  computes a UTxO set containing only the output references of transaction inputs, paired with the outputs contained in those inputs. The function  $\text{mkOuts}$  computes a UTxO set containing exactly the outputs of  $tx$ , each associated to the key  $(tx, ix)$ , where the index  $ix$  is the place of the associated output in the list of  $tx$  outputs. For details, see Figure 4.

The EUTxO ledger definition [9], on which we base our semantics, models the ledger as a list of transactions, recorded in the order of processing. In essence, this is the reflexive-transitive closure of the step relation above, where there is a special initial transaction with no inputs a.k.a. the *genesis* transaction. We have deliberately left a full treatment of trace properties for future work: our model currently says nothing about an initial state and only specifies how to update an arbitrary UTxO set in accordance with the LEDGER rule. It is worth mentioning that validation check (i) would contradict the genesis transaction, although it is necessary for subsequent transaction in order to ensure *replay protection*: a transaction valid at some given point in time cannot again be valid in the future.

### 3 Simulations and the structured contract formalism

Intuitively, a stateful program is correctly implemented on the ledger whenever its state is observable in (i.e. computable from) the ledger state, and whenever the ledger state is updated, the observed program state is updated in accordance with the program's specification. In this section, we formalize the notion of smart contract scripts correctly implementing a specification.

The purpose of a smart contract script is to encode the conditions under which a transaction *can update a part of the ledger state* with which the script is associated, e.g. change the total quantity of tokens under a given policy. This interpretation of the use of stateless code on the ledger justifies a *stateful* program model for representing most programs running on the ledger. Stateful programs are implemented using one or more interacting scripts controlling the updates of the contract state data within the ledger state. In this section, we specify what is required to construct a simulation relation between two transition systems specified via small-step semantics, and define a subset of such simulations that corresponds to the set of all (correctly) implemented stateful contracts on an EUTxO ledger.

### 3.1 Simulations

We instantiate the definition a *simulation* [23] with labeled state transition systems expressed as small-step semantics specifications.

**Simulation definition.** Let TRANS and STRUC be small-step labeled transition systems. A *simulation* of TRANS in STRUC, denoted by  $(\text{STRUC}, \sim, \simeq) \succeq \text{TRANS}$ , consists of the following types together with the following relations :

$$\begin{aligned} \_ \vdash \_ &\xrightarrow{\text{TRANS}} \_ \subseteq \text{Env} \times \text{State} \times \text{Input} \times \text{State} \\ \_ \vdash \_ &\xrightarrow{\text{STRUC}} \_ \subseteq \text{Env}' \times \text{State}' \times \text{Input}' \times \text{State}' \\ \_ &\sim \_ \subseteq \text{State} \times \text{State}' \\ \_ &\simeq \_ \subseteq (\text{Env} \times \text{Input}) \times (\text{Env}' \times \text{Input}') \end{aligned}$$

such that the following holds :

$$\text{Sim} \frac{(e, i) \simeq (e', j) \quad u \sim s \quad e \vdash (u) \xrightarrow{\text{TRANS}} (u')}{\exists s', u' \sim s' \wedge e' \vdash (s) \xrightarrow{\text{STRUC}} (s')} \quad (1)$$

Instantiating a simulation of TRANS in STRUC requires specifying TRANS, STRUC,  $\pi, \pi_{\text{Tx}}$ , and fulfilling the *proof obligation* Sim.

### 3.2 Structured contracts

The simulation definition we give is general, however, the rest of this work is geared towards reasoning about the programmable via user-defined scripts. For this reason, we define a particular class of simulations of LEDGER. Since scripts are not allowed to inspect block-level data (e.g. the current slot number), we fix the environment of the structured contract specification to be a singleton type  $\{\star\}$ . We also require that  $\sim$  is a partial function, rather than a relation, which computes a specific contract state for a given UTxO state (or fails, returning  $\star$ ). Additionally, we require that  $\simeq$  is a function.

**Definition (Structured contract).** Let  $(\text{STRUC}, \sim, \simeq) \succeq \text{LEDGER}$  be a simulation. We say that it is a *structured contract* whenever  $\text{Env} = \star$ , and there exist two functions  $\pi : \text{UTxO} \rightarrow \text{State} \cup \{\star\}$ ,  $\pi_{\text{Tx}} : \text{Tx} \rightarrow \text{Input}$  such that :

$$\begin{aligned} \text{utxo} \sim s &:= (\pi(\text{utxo}) = s) \\ (\text{slot}, \text{tx}) \simeq (\star, i) &:= (\pi_{\text{Tx}}(\text{tx}) = i) \end{aligned}$$

**Discussion.** This definition states that a ledger step  $(\text{slot}, \text{utxo}, \text{tx}, \text{utxo}')$  with  $\pi(\text{utxo}) \neq \star$ , there is a step given by  $(\star, \pi(\text{utxo}), \pi_{\text{Tx}}(\text{tx}), \pi(\text{utxo}')) \in \text{STRUC}$  which corresponds to the ledger step. It is possible that a transaction updates the UTxO set but not the contract state, so that  $\pi(\text{utxo}) = \pi(\text{utxo}')$ .

We do not assume that a valid contract state can be computed from an arbitrary UTxO state. For this reason, the function  $\pi$  is partial. For example, it is possible that two copies of the same NFT exist in a given ledger state. When programmed correctly, an NFT minting policy would not allow this to happen. When reasoning about properties of such a policy, we ignore ledger start states where the NFT uniqueness condition has already been violated. Defining a class of structured contracts for which the relation in Equation 1 is a bisimulation [23] between STRUC and LEDGER is a more difficult problem, and we leave it for future work.

#### 4 NFT minting policy as a structured contract

Our first structured contract example expresses a *specific minting policy*. Constructing structured contracts specifying the evolution of the quantity of tokens under a specific policy is a tool for formal analysis of minting policy code. In particular, for a correctly defined minting policy, we are able to express and prove the defining property of an NFT under this policy: at most one such token can exist on the ledger. Instantiating an NFT as a structured contract allows us to state and prove a property that is quite naturally expressed for account-based blockchains with stateful NFT contracts, such as the ERC-721 [16], but poses a challenge for EUTxO ledger program analysis. For the Agda mechanization of this example, see the accompanying code at:

<https://omelkonian.github.io/structured-contracts/NFT.html>.

We first pick an identifier for the policy we wish to express, `myNFTPolicy`. Before writing the policy code, we define a system NFT to specify how we want the total number of tokens on the ledger under this policy to behave. Here, the state type is `State := Value`, and `Input := Tx`.

$$\begin{array}{c}
 i := \{ (\text{myNFTPolicy} \mapsto \text{tkns}) \in \text{tx.mint} \} \\
 \emptyset \leq s \leq s + i \leq \{ \text{myNFTPolicy} \mapsto \{ \emptyset \mapsto 1 \} \} \\
 \text{UpdateNFTTotal} \frac{}{\vdash (s) \xrightarrow[\text{NFT}]{\text{tx}} (s + i)}
 \end{array}$$

This specification states that the only allowed transitions are (i) a constant one, and (ii) adding a single NFT  $\{ \text{myNFTPolicy} \mapsto \{ \emptyset \mapsto 1 \} \} \in \text{Value}$ , to the state — if one does not yet exist. An NFT whose total ledger quantity obeys this specification can never be burned, must be the only token under its policy, and must have the empty string  $\emptyset$  as a name. It also does not require any authentication to be minted. To define the policy and projection functions, we pick an output reference `myNFTRef` which we call an *anchor*. That is, `myNFTRef` must be spent by the NFT-minting transaction as a mechanism to ensure that no other transaction can mint another NFT under this policy. Next, we define the projection function,

$$\pi(\text{utxo}) := \begin{cases} s & \text{if } (s = \{ \text{myNFTPolicy} \mapsto \{ \emptyset \mapsto 1 \} \} \wedge \neg \text{hasRef}) \vee s = \emptyset \\ \star & \text{otherwise} \end{cases}$$

where

$$\begin{aligned}
 s &:= \{ p \mapsto \text{tkns} \mid p \mapsto \text{tkns} \in \sum_{_ \mapsto \text{out} \in \text{utxo}} \text{out.value}, p = \text{myNFTPolicy} \} \\
 \text{hasRef} &:= (\text{myNFTRef} \mapsto \_ \in \text{utxo})
 \end{aligned}$$

Here,  $\pi(\text{utxo})$  returns a non- $\star$  result when either no tokens under the `myNFTPolicy` policy exist, or only the token  $\{ \text{myNFTPolicy} \mapsto \{ \emptyset \mapsto 1 \} \}$  exists under this policy, and the anchor `myNFTRef` is not in the UTxO. We define the policy,

$$\text{myNFTPolicy} := \text{mkMyNFTPolicy}(\text{myNFTRef})$$

$$\begin{aligned}
 \llbracket \text{mkMyNFTPolicy}(\text{myRef}) \rrbracket(\_, (\text{tx}, \text{pid})) &:= \exists (\text{myRef}, \_, \_) \in \text{tx.inputs} \\
 &\wedge \{ \text{myNFTPolicy} \mapsto \{ \emptyset \mapsto 1 \} \} \in \text{tx.mint}
 \end{aligned}$$



To prove Sim for the NFT contract (see Appendix B for a proof sketch, which is also mechanized in Agda), we need to make an additional assumption,

**NFT re-minting protection.**  $\forall (slot, utxo, tx, utxo') \in \text{LEDGER},$

$$((\pi(utxo) = \emptyset \wedge \text{myNFTRef} \in \text{getORefs}(tx)) \vee \pi(utxo) > \emptyset) \Rightarrow tx \neq \text{myNFTRef.fst} \quad (2)$$

This assumption guarantees that if an NFT with policy myNFTPolicy exists on ledger state  $utxo$ , the transaction myNFTRef.fst, whose output was spent as a condition for minting the NFT, cannot be valid again in  $utxo$ . It also requires that a transaction *spending* myNFTRef cannot be the same transaction that *added* myNFTRef to the UTxO set. Under reasonable constraints on an initial ledger state, replay protection (c.f. Section 2) is a global invariant of EUTxO ledger state traces (as proven in prior work on EUTxO under the name ‘uniqueness’ [7]), from which we can directly derive re-minting protections.

We note here that, as exemplified by Assumption 2, the SCF approach to implementation allows us to express specific consequences of trace-based ledger transition system properties as local invariants. This enables the separation of reasoning about the global behavior of the ledger from reasoning about the correctness of contract implementation, for which we can make use of relevant local invariants of ledger behavior.

**NFT property example.** At most one NFT under the policy myNFTPolicy can ever exist in any  $utxo$  that is valid for NFT : for any  $utxo$  such that  $\pi(utxo) \neq \star$ ,

$$\pi(utxo) \subseteq \{ \text{myNFTPolicy} \mapsto \{ \emptyset \mapsto 1 \} \}$$

This is immediate from the definition of  $\pi$ , however, this result is meaningful. By definition of Sim, and the fact that NFT is a structured contract, it is not possible to transition from a UTxO state valid for NFT (i.e.  $\pi(utxo) \neq \star$ ) to a state which is not valid for NFT. That is, with  $(slot, utxo, tx, utxo') \in \text{LEDGER}$ , the updated state  $\pi(utxo')$  must also always have at most one NFT under myNFTPolicy. This also implies that at most one can ever be minted by a valid transaction applied to a  $utxo$  valid for NFT.

## 5 Multiple implementations of a single specification

In this section we present an example of a specification that has more than one correct implementation, one of which is distributed across multiple UTxO entries. The guarantee that the two implement the same specification enables contract authors to meaningfully compare them across relevant characteristics, such as space usage, or parallelizability.

### 5.1 Toggle specification

We define (and mechanize in the corresponding Agda code) a specification wherein the state consists of two booleans, and only one can be true at a time. We set the contract input to be  $\{\text{toggle}\} \cup \{\star\}$ . The two booleans in the state are both flipped by the input toggle, and unchanged by  $\star$ . We define the transition system TOGGLE :

$$\begin{array}{c} \text{Noop} \text{-----} \\ \left( x, y \right) \xrightarrow[\text{TOGGLE}]{\star} \left( x, y \right) \end{array} \quad \begin{array}{c} \text{Toggle} \text{-----} \\ \left( x, y \right) \xrightarrow[\text{TOGGLE}]{\text{toggle}} \left( y, x \right) \end{array}$$

## 5.2 Toggle implementations

The *naive implementation* uses the datum of a single UTxO entry to store a representation of the full state of the TOGGLE contract. The *distributed implementation* uses datums in two distinct UTxO entries to represent the first and the second value of the boolean pair that is the TOGGLE state.

**Thread token scripts.** We use the *thread tokens* mechanism [7] to construct a unique identifier of the UTxO (or pair of UTxOs) from which the contract state is computed. The mechanism for ensuring uniqueness of a thread token is the same as for the NFT contract example in Section 4. It relies on the replay protection property of the ledger, ensuring that if a thread token exists on the ledger, it must be unique. In both the naive and distributed implementations, the thread token minting policy guarantees that thread tokens are generated in quantity of at most 1 by a transaction that spends a specific output reference `myRef`.

For the naive implementation, one thread token is sufficient to identify the state-bearing UTxO. Upon minting, the policy requires the token to be placed into a UTxO locked by a specific script, which is passed as a parameter to the minting policy. This script (discussed below) ensures the correct evolution of the contract state. The datum in the UTxO containing the thread token is the initial state of the contract encoded as a pair of booleans (by the partial decoder function  $\text{fromData}_N : \text{Data} \rightarrow \mathbb{B} \cup \{\star\}$ ). It can be any pair of correctly encoded booleans. See Figure 5 for the policy pseudocode.

For the distributed implementation, two distinct NFTs are needed to identify the UTxOs containing the TOGGLE state data. Both NFTs are under the same minting policy and must be minted by a single transaction, but have distinct token names, “*a*” and “*b*”. Upon being minted, the policy requires that they are placed in separate UTxOs, locked by the same script (discussed below). The datum in each must be decodeable (by  $\text{fromData}_D : \text{Data} \rightarrow \mathbb{B} \cup \{\star\}$ ) as a boolean. See Figure 6 for the policy pseudocode.

**Validator scripts.** We require different UTxO-locking scripts for our two distinct implementations. Both scripts serve the following function: when the UTxO locked by the script is spent, the script must ensure that thread tokens are propagated into UTxOs that are locked by the same validator as the spent UTxOs containing the thread tokens, and that the datums in those UTxOs are correct. This implements the Toggle rule. The Noop rule applies when the transaction does not spend the thread tokens.

For the naive version, the datum in the new UTxO containing the thread token must decode as a pair of booleans whose order is reversed as compared to the booleans encoded in the datum of the spent UTxO that previously contained the thread token. We define it by :

$$\begin{aligned} & \llbracket \text{toggleVal}_N(\text{myRef}) \rrbracket((b, b'), r, (tx, i)) := ttt = i.\text{output.value} \wedge r = \text{toggle} \\ & \wedge \exists o \in tx.\text{outputs}, (b', b) = (o.\text{datum}) \wedge (o.\text{validator} = vi) \wedge (ttt = o.\text{value}) \\ & \textbf{where} \\ & \quad vi := i.\text{output.validator} \\ & \quad ttt := \{\text{toggleTT}_N(\text{myRef}, vi) \mapsto \{\text{encode}(vi) \mapsto 1\}\} \end{aligned}$$

The function  $\text{encode} : \text{Script} \rightarrow [\text{Char}]$  encodes a script as a string for the purpose of specifying (via the token name) the output-locking script that must persistently lock the thread token.

The distributed implementation script ensures that both the thread token-containing UTxOs are spent simultaneously. Then, it checks that the booleans in the datums are

$$\begin{aligned}
\llbracket \text{toggleVal}_D(\text{myRef}) \rrbracket(b, \text{toggle}, (tx, i)) := & \\
& ((tta = i.\text{output.value}) \Rightarrow \\
& \exists o, o' \in tx.\text{outputs}, i' \in tx.\text{inputs}, \\
& o.\text{validator} = o'.\text{validator} = vi \wedge \\
& tta = o.\text{value} \wedge ttb = o'.\text{value} \wedge i'.\text{output.value} = ttb \\
& o.\text{datum} = i'.\text{output.datum} \wedge o'.\text{datum} = i.\text{output.datum}) \\
& \wedge \\
& ((ttb = i.\text{output.value}) \Rightarrow \\
& \exists o, o' \in tx.\text{outputs}, i' \in tx.\text{inputs}, \\
& o.\text{validator} = o'.\text{validator} = vi \wedge \\
& tta = o.\text{value} \wedge ttb = o'.\text{value} \wedge i'.\text{output.value} = tta \\
& o.\text{datum} = i.\text{output.datum} \wedge o'.\text{datum} = i'.\text{output.datum}) \\
& \wedge \\
& ((tta = i.\text{output.value}) \vee (ttb = i.\text{output.value})) \\
\text{where} & \\
vi := & i.\text{output.validator} \\
tta := & \{\text{toggleTT}_D(\text{myRef}, vi) \mapsto \{\text{encode}(vi) \# "a" \mapsto 1\}\} \\
ttb := & \{\text{toggleTT}_D(\text{myRef}, vi) \mapsto \{\text{encode}(vi) \# "b" \mapsto 1\}\}
\end{aligned}$$

■ **Figure 1** TOGGLE validator script for the distributed implementation

switched places : the one that was in the UTxO with token "a" must now be in a new UTxO with token "b", and vice-versa. The validator script is given in Figure 1.

**Ledger representation.** The state projection function computations return a valid contract state (i.e. a pair of booleans) whenever the anchor reference `myRef` is not in the UTxO, and thread tokens have been minted according to their policy and placed alongside the appropriate datums and scripts. The input projection function returns `toggle` whenever a transaction contains the thread tokens in its input(s), and `*` otherwise. For details, see Figures 7 and 2.

In Appendix C, we give a proof sketch for the simulation relations between TOGGLE and LEDGER to complete the instantiation of the two versions of the structured contract. The proofs are very similar to those for the NFT contract, so we have not mechanized them. To avoid duplication of thread tokens, we again need to make the additional assumption that a transaction cannot be valid again if it has previously been applied. That is, for any  $(slot, utxo, tx, utxo') \in \text{LEDGER}$ , with  $\pi(utxo) \neq *$ , necessarily  $tx \neq \text{myRef.fst}$ .

**TOGGLE property example.** The following property states that in any step of TOGGLE, either the state booleans are swapped, or stay the same. Its proof is immediate from the specification, regardless of the implementation.

$$(*, (a, b), i, (c, d)) \in \text{TOGGLE} \Rightarrow (c, d) = (b, a) \vee (c, d) = (a, b)$$

$$\pi_d(utxo) := \begin{cases} (a, b) & \text{if } \text{myRef} \notin \{i \mid i \mapsto o \in utxo\} \wedge \exists! (i \mapsto o, i' \mapsto o') \in utxo, \\ & \text{tta} = o.\text{value} \wedge \text{ttb} = o'.\text{value} \\ & \wedge o.\text{validator} = \text{toggleVal}_D(\text{myRef}) = o'.\text{validator} \\ & \wedge o.\text{datum} = a \wedge o'.\text{datum} = b \\ * & \text{otherwise} \end{cases}$$

$$\pi_{Tx,d}(tx) := \begin{cases} \text{toggle} & \text{if } \exists i, i' \in tx.\text{inputs}, i.\text{output.value} = \text{tta} \wedge i'.\text{output.value} = \text{ttb} \\ * & \text{otherwise} \end{cases}$$

■ Figure 2 TOGGLE distributed projections

## 6 Related work

**Scilla** [14] is a intermediate-level language for expressing smart contracts as state machines on an account-based ledger model. It is formalized in Coq, and the contracts written in it are amenable to formal verification. In our work we pursue the same goal of correctly implementing stateful contracts and formally studying their behavior on the EUTxO ledger.

**CoSplit** [26] is a static analysis tool for implementing *sharding* in an account-based blockchain. Sharding is the act of separating contract state into smaller fragments that can be affected by commuting operations, usually for the purposes of increasing parallelism and scalability. Our work allows users to build contracts whose state is distributed across multiple UTxOs and tokens on the ledger. One of the benefits of an EUTxO ledger is that transaction application commutes [2]. Therefore, no additional work is required to ensure commutativity when updating only a part of a contract with distributed state.

The Bitcoin Modelling Language (**BitML**) [5] enables the definition of smart contracts in a particular restricted class of state machines on the Bitcoin ledger, where a *computational soundness* result securely establishes a relation between high-level BitML contracts and low-level Bitcoin transactions. The BitML state machines are less expressive than the class of specifications considered in our model, although subsequent work introduces recursion [3]. Another derivative of BitML, the **ILLUM** language [4], allows users to build Turing-complete contracts that achieve a secure simulation to UTxO. A key difference between ours approach and ILLUM appears to be the direction of the security proofs: our work presents users with a framework for choosing an implementation and proving the property that “transactions can only ever update the contract state according to the stateful contract specification”, whereas the work on ILLUM enables, given a specified contract transition, automatically constructing *specific* transactions that are guaranteed to perform the corresponding contract update. The goal in all the above cases is again similar to ours — to guarantee certain properties of on-chain state machine implementations. Alas, we lack the accompanying meta-theoretical guarantees, but we are confident similar properties hold for our system and hope to make this formal in future work.

**VeriSolid** [22] synthesizes Solidity smart contracts from a state machine specification, with support for formal verification. The underlying ledger model for VeriSolid is an account-based model. Like BitML, VeriSolid is less flexible in the types of state machines that can be implemented, and how they can be implemented, but offers more automation than our work. Yet another language for expressing the formal semantics of (a subset of)

Solidity exists [21], has been specified in Isabelle/HOL, and was developed with the goal of improving formal verification methods of smart contracts. This work differs from ours in the underlying ledger model, as well as the direction of the ledger-contract simulation relation.

The **K framework** [27] is a unifying formal semantics framework for all programming languages, which has been used as a tool to perform audits of smart contracts [28], as well as specifying Solidity operational semantics [19]. Auditing is a common approach to smart contract verification [1, 11], which will also be useful for structured contract specifications.

## 7 Conclusion

Previous work on stateful EUTxO contracts [7] constructs a consolidated (single-UTxO) on-chain implementation for a given constraint-emitting machine (CEM), including a proof that the contract state is always updated correctly by a given transaction. Constructing the implementation and associated proof in this formalism is done uniformly, and holds for any specification it is instantiated with. This fixed-implementation approach allows for specifying only a small subset of possible constraints on the updating transaction data. In this work, we introduce the *structured contract framework*, in which, for a particular specification, the contract author *decides* on an implementation that satisfies their specific requirements (e.g. on-chain memory constraints, distributed vs consolidated, etc.), and is not limited in the constraints a state update may place on the transaction data.

Our formalism defines the space of all stateful contracts implementable on the EUTxO ledger via user-defined scripts. Our work opens up the possibility of formal analysis of the behavior of a much larger class of contracts which may have previously been implemented ad-hoc, or via an ill-suited formalism. Instantiating our framework requires the definition of a small-step stateful contract specification, projection functions which compute the state and input values of a contract for a given UTxO state and transaction (resp.), and a proof of the integrity of the implementation. Similar to the proof completed for the CEMs formalism, the proof obligation in our framework ensures that on-chain state evolution of a contract adheres to its specification.

We note that the *existence* of a valid UTxO state update corresponding to a given contract update is not guaranteed, and such an update is difficult to construct in the general case. We leave this for future work. Another limitation of our approach is the lack of proof automation, which presents a challenge due to the fact that a contract state may be computed from a given UTxO state by an arbitrary user-defined function. A full formalization of structured contract behavior in terms of trace-based properties of contracts and their expression at the ledger level is needed, and is the subject of future work.

We present examples of contracts and safety properties satisfied by these examples: (i) a stateful model of an NFT policy, and, for another simple contract, a pair of (ii) a distributed and (iii) a consolidated implementation. Using our framework to construct more sophisticated and realistic contract examples, such as DEXs or account simulations, is the subject of future work. The approach we presented can also potentially be applied to existing contracts for which a small-step specification can be constructed, making them more amenable to formal verification.

---

**References**

---

- 1 Arnaud Bailly. Model-based testing with QuickCheck, 2022. URL: <https://engineering.iog.io/2022-09-28-introduce-q-d/>.
- 2 Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. A theory of transaction parallelism in blockchains. *Logical Methods in Computer Science*, Volume 17, Issue 4, nov 2021. URL: <https://doi.org/10.46298/2Flmcs-17%284%3A10%292021>, doi:10.46298/lmcs-17(4:10)2021.
- 3 Massimo Bartoletti, Stefano Lande, Maurizio Murgia, and Roberto Zunino. Verification of recursive bitcoin contracts. *CoRR*, abs/2011.14165, 2020. URL: <https://arxiv.org/abs/2011.14165>, arXiv:2011.14165.
- 4 Massimo Bartoletti, Riccardo Marchesin, and Roberto Zunino. Secure compilation of rich smart contracts on poor UTXO blockchains, 2023. [arXiv:2305.09545](https://arxiv.org/abs/2305.09545).
- 5 Massimo Bartoletti and Roberto Zunino. BitML: a calculus for Bitcoin smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 83–100. ACM, 2018.
- 6 Vitalik Buterin. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. <https://ethereum.org/en/whitepaper/>, 2014.
- 7 Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann Müller, Michael Peyton Jones, Polina Vinogradova, and Philip Wadler. Native custom tokens in the Extended UTXO model. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III*, volume 12478 of *Lecture Notes in Computer Science*, pages 89–111. Springer, 2020. URL: "[https://doi.org/10.1007/978-3-030-61467-6\\_7](https://doi.org/10.1007/978-3-030-61467-6_7)", doi:10.1007/978-3-030-61467-6\_7.
- 8 Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann Müller, Michael Peyton Jones, Polina Vinogradova, Philip Wadler, and Joachim Zahnentferner. UTXO<sub>ma</sub>: UTXO with multi-asset support. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III*, volume 12478 of *Lecture Notes in Computer Science*, pages 112–130. Springer, 2020. URL: "[https://doi.org/10.1007/978-3-030-61467-6\\_8](https://doi.org/10.1007/978-3-030-61467-6_8)", doi:10.1007/978-3-030-61467-6\_9.
- 9 Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The Extended UTxO model. In *Proceedings of Trusted Smart Contracts (WTSC)*, volume 12063 of *LNCS*. Springer, 2020.
- 10 Manuel M. T. Chakravarty, Sandro Coretti, Matthias Fitzi, Peter Gazi, Philipp Kant, Aggelos Kiayias, and Alexander Russell. Hydra: Fast isomorphic state channels. *IACR Cryptol. ePrint Arch.*, 2020:299, 2020.
- 11 Florent Chevrou. A journey through the auditing process of a smart contract, 2023. URL: <https://www.tweag.io/blog/2023-05-11-audit-smart-contract/>.
- 12 Jared Corduan, Matthias Gudemann, and Polina Vinogradova. A Formal Specification of the Cardano Ledger. <https://github.com/input-output-hk/cardano-ledger/releases/latest/download/shelley-ledger.pdf>, 2019.
- 13 Ergo Team. Ergo: A Resilient Platform For Contractual Money. <https://whitepaper.io/document/753/ergo-1-whitepaper>, 2019.
- 14 Ilya Sergey et al. Safer smart contract programming with Scilla. 3(OOPSLA):185, 2019.
- 15 Pablo Lamela Seijas et al. Marlowe: Implementing and analysing financial contracts on blockchain. In *Financial Cryptography and Data Security*, pages 496–511, Cham, 2020. Springer International Publishing.
- 16 Ethereum Team. ERC-721 TOKEN STANDARD. <https://ethereum.org/en/developers/docs/standards/tokens/erc-721>, 2023.
- 17 LM Goodman. Tezos—a self-amending crypto-ledger (white paper). 2014.

- 18 Tobias Guggenberger, Vincent Schlatt, Jonathan Schmid, and Nils Urbach. A structured overview of attacks on blockchain systems. *PACIS*, page 100, 2021.
- 19 Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanan, Yang Liu, and Jun Sun. Semantic understanding of smart contracts: Executable operational semantics of Solidity. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1695–1712, 2020. doi:10.1109/SP40000.2020.00066.
- 20 Andre Knispel and Polina Vinogradova. A Formal Specification of the Cardano Ledger integrating Plutus Core. <https://github.com/input-output-hk/cardano-ledger/releases/latest/download/alonzo-ledger.pdf>, 2021.
- 21 Diego Marmosler and Achim D. Brucker. A denotational semantics of Solidity in Isabelle/HOL. In *Software Engineering and Formal Methods: 19th International Conference, SEFM 2021, Virtual Event, December 6–10, 2021, Proceedings*, page 403–422, Berlin, Heidelberg, 2021. Springer-Verlag. doi:10.1007/978-3-030-92124-8\_23.
- 22 Anastasia Mavridou, Aron Laszka, Emmanouela Stachtari, and Abhishek Dubey. VeriSolid: Correct-by-design smart contracts for Ethereum. In *International Conference on Financial Cryptography and Data Security*, pages 446–465. Springer, 2019.
- 23 Robin Milner. *Communicating and mobile systems: the  $\pi$ -calculus*. Cambridge University Press, 1999.
- 24 S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. <https://bitcoin.org/en/bitcoin-paper>, October 2008.
- 25 Ulf Norell. Dependently typed programming in Agda. In *International School on Advanced Functional Programming*, pages 230–266. Springer, 2008.
- 26 George Pirlea, Amrit Kumar, and Ilya Sergey. Practical smart contract sharding with ownership and commutativity analysis. *PLDI 2021*, page 1327–1341, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454112.
- 27 Grigore Roşu and Traian Şerbănuţă. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming*, 79:397–434, 08 2010. doi:10.1016/j.jlap.2010.03.012.
- 28 Runtime Verification Team. Smart contract analysis and verification, 2023. URL: <https://runtimeverification.com/smartcontract>.
- 29 The ZILLIQA Team. The ZILLIQA Technical Whitepaper. <https://docs.zilliqa.com/whitepaper.pdf>, 2017.
- 30 The Solidity Authors. Solidity 0.8.25 documentation. <https://docs.soliditylang.org/en/v0.8.25/>, 2023.
- 31 Jan Xie. Nervos CKB: A Common Knowledge Base for Crypto-Economy. <https://github.com/nervosnetwork/rfcs/blob/master/rfcs/0002-ckb/0002-ckb.md>, 2018.

**A** EUTxO details

Figure 3 introduces non-standard syntax we use throughout.

$\mathbb{H} = \bigcup_{n=0}^{\infty} \{0,1\}^{8n}$	the type of bytestrings
$\star : \{\star\}$	the one-element set, and its one inhabitant
$a : A \cup \{\star\}$	maybe type over $A$
$\text{fst} : (A \times B) \rightarrow A$	first projection
$(a, b) : \text{Interval}[A]$	intervals over a totally-ordered set $A$
$\text{Key} \mapsto \text{Value} \subseteq \{k \mapsto v \mid k \in \text{Key}, v \in \text{Value}\}$	finite map with unique keys
$[a1; \dots; ak] : [C]$	finite list with terms of type $C$
$\# : ([C], [C]) \rightarrow [C]$	list concatenation
$h :: t : [C]$	list with head $h$ and tail $t$

**Figure 3** Notation

Figure 4 lists the primitives and derived types that comprise the foundations of the EUTxO model, along with some ancillary definitions. (Outputs normally refer to transaction IDs by hash, but we simplify here for clarity.)

**B** Sim relation proof sketch for NFT

Suppose  $(\text{slot}, \text{utxo}, \text{tx}, \text{utxo}') \in \text{LEDGER}$ , and  $\pi(\text{utxo}) \neq \star$ . There are two disjuncts :

- (i) If  $i = \{(\text{myNFTPolicy} \mapsto \text{tkns}) \in \text{tx.mint}\} = \emptyset$ , by preservation of value (rule (v) in Section 2.2), the amount  $s$  of tokens under `myNFTPolicy` remains unchanged in  $\text{utxo}'$ . If  $s = \emptyset$ , we get  $s' = \emptyset + \emptyset = \emptyset$ . Then, by Assumption 2 we conclude that  $\text{tx}$  does not add `myRef` to  $\text{utxo}$ . So,  $\pi(\text{utxo}')$  is defined, and  $\pi(\text{utxo}) = \pi(\text{utxo}') = \emptyset$ . If  $s > \emptyset$ , by Assumption 2, we conclude that  $\text{tx}$  cannot add an output with reference `myRef` in the  $\text{utxo}'$ . Since, by  $\pi(\text{utxo}) \neq \star$ , we know that `myRef` was not in  $\text{utxo}$  either, we conclude that there is no output with `myRef` in  $\text{utxo}'$ . So,  $\pi(\text{utxo}) = \pi(\text{utxo}')$ .
- (ii) If  $i \neq \emptyset$ , tokens under `myNFTPolicy` are being minted, and the policy must be checked by ledger rule (ix) in Section 2.2. Necessarily, by `myNFTPolicy`,  $i = \{\text{pid} \mapsto \{\emptyset \mapsto 1\}\}$ . If  $s = \emptyset$ ,  $s' = s + i = i$  is the new total amount of tokens under policy `myNFTPolicy` in the UTxO. The unique output with reference `myRef` must be removed from the UTxO by  $\text{tx}$ , so that it is not contained in  $\text{utxo}'$ . By Assumption 2, it is also not added back by  $\text{tx}$  to  $\text{utxo}'$ . Then,  $\pi(\text{utxo}') \neq \star$ , and is equal to  $i$ . If  $s \geq \emptyset$ , an output with reference `myRef` is not in  $\text{utxo}'$ . So, `myNFTPolicy` fails, and the  $\text{tx}$  is not valid on the ledger.

**C** Sim relation proof sketch for TOGGLE

Suppose that  $(\text{slot}, \text{utxo}, \text{tx}, \text{utxo}')$  and  $\pi(\text{utxo}) = (a, b)$ . We first observe that each of the thread tokens in either implementation is present in an input of the transaction if and only if it is present in the output. This is because  $\pi(\text{utxo}) = (a, b)$  implies that the unique token(s)



## LEDGER PRIMITIVES

$[\_]$  :  $\text{Script} \rightarrow \text{Datum} \times \text{Redeemer} \times \text{ValidatorContext} \rightarrow \mathbb{B}$  *applies a validator script to its arguments*  
 $[\_]$  :  $\text{Script} \rightarrow \text{Redeemer} \times \text{PolicyContext} \rightarrow \mathbb{B}$  *applies a monetary policy to its arguments*  
 $\text{checkSig}$  :  $\text{Tx} \rightarrow \text{PubKey} \rightarrow \mathbb{H} \rightarrow \mathbb{B}$  *checks that a given key signed a transaction*

## DEFINED TYPES

$\text{Signature}$  =  $\text{PubKey} \mapsto \mathbb{H}$   
 $\text{OutputRef}$  =  $(\text{id} : \text{Tx}, \text{index} : \text{Ix})$   
 $\text{Output}$  =  $(\text{validator} : \text{Script},$   
            $\text{value} : \text{Value},$   
            $\text{datum} : \text{Data})$   
 $\text{TxInput}$  =  $(\text{outputRef} : \text{OutputRef},$   
                $\text{output} : \text{Output},$   
                $\text{redeemer} : \text{Redeemer})$   
 $\text{Tx}$  =  $(\text{inputs} : \mathbb{P} \text{TxInput},$   
            $\text{outputs} : [\text{Output}],$   
            $\text{validityInterval} : \text{Interval}[\text{Slot}],$   
            $\text{mint} : \text{Value},$   
            $\text{mintRdmrs} : \text{Script} \mapsto \text{Redeemer},$   
            $\text{sigs} : \text{Signature})$   
 $\text{ValidatorContext}$  =  $(\text{Tx}, (\text{Tx}, \text{TxInput}))$   
 $\text{PolicyContext}$  =  $(\text{Tx}, \text{PolicyID})$

## HELPER FUNCTIONS

$\text{toMap} : \text{Ix} \rightarrow [\text{Output}] \rightarrow (\text{Ix} \mapsto \text{Output})$   
 $\text{toMap}(\_, []) = []$   
 $\text{toMap}(ix, u :: \text{outs}) = \{ ix \mapsto u \} \cup \text{toMap}(ix + 1, \text{outs})$   
 $\text{mkOuts} : \text{Tx} \rightarrow \text{UTxO}$   
 $\text{mkOuts}(tx) = \{ (tx, ix) \mapsto o \mid (ix \mapsto o) \in \text{toMap}(0, tx.\text{outputs}) \}$   
 $\text{getORefs} : \text{Tx} \rightarrow \mathbb{P} (\text{OutputRef})$   
 $\text{getORefs}(tx) = \{ i.\text{outputRef} \mid i \in tx.\text{inputs} \}$

■ **Figure 4** Primitives and basic types for the  $\text{EUTxO}_{\text{ma}}$  model

already exists in the  $\text{UTxO}$  set, and the minting policy cannot be satisfied, which holds because  $\text{myRef} \in \{ i.\text{outputRef} \mid i \in tx.\text{inputs} \}$  contradicts  $\pi(utxo) = (a, b)$ . So, thread tokens are not being minted or burned, and, by rule (v) in Section 2.2, we can make the required conclusion.

Now, there are two possibilities,  $\pi(tx) = \star$  and  $\pi(tx) = \text{toggle}$ , for each of which we must prove that  $(\star, \pi(utxo), \pi(tx), \pi(utxo'))$  and  $\pi(utxo) = \pi(utxo')$ .

**Naive implementation.**

## 4:18 Structured Contracts in the EUTxO Ledger Model

$$\begin{aligned}
& \text{toggleTT}_N : \text{OutputRef} \rightarrow \text{Script} \rightarrow \text{Script} \\
& \llbracket \text{toggleTT}_N(\text{myRef}, s) \rrbracket(\_, (tx, pid)) := \text{myRef} \in \{i.\text{outputRef} \mid i \in tx.\text{inputs}\} \\
& \quad \wedge tx.\text{mint} = \{pid \mapsto \{\text{encode}(s) \mapsto 1\}\} \\
& \quad \wedge \exists o \in tx.\text{outputs}, \\
& \quad \quad o.\text{value} = \{pid \mapsto \{\text{encode}(s) \mapsto 1\}\} \\
& \quad \quad \wedge o.\text{validator} = s \\
& \quad \quad \wedge \text{fromData}_N(o.\text{datum}) \neq \star
\end{aligned}$$

■ **Figure 5** TOGGLE thread token minting policy for the naive implementation

$$\begin{aligned}
& \text{toggleTT}_D : \text{OutputRef} \rightarrow \text{Script} \rightarrow \text{Script} \\
& \llbracket \text{toggleTT}_D(\text{myRef}, s) \rrbracket(\_, (tx, pid)) := \\
& \quad \text{myRef} \in \{i.\text{outputRef} \mid i \in tx.\text{inputs}\} \\
& \quad \wedge tta + ttb = tx.\text{mint} \\
& \quad \wedge \exists o^a, o^b \in tx.\text{outputs}, \\
& \quad \quad o^a.\text{value} = tta \wedge o^b.\text{value} = ttb \\
& \quad \quad \wedge o^a.\text{validator} = o^b.\text{validator} = s \\
& \quad \quad \wedge \text{fromData}_D(o^a.\text{datum}) \neq \star \wedge \text{fromData}_D(o^b.\text{datum}) \neq \star
\end{aligned}$$

where

$$\begin{aligned}
tta & := \{pid \mapsto \{\text{encode}(s) \text{ ++ } "a" \mapsto 1\}\} \\
ttb & := \{pid \mapsto \{\text{encode}(s) \text{ ++ } "b" \mapsto 1\}\}
\end{aligned}$$

■ **Figure 6** TOGGLE thread token minting policy for the distributed implementation

- (i)  $\pi(tx) = \star$ : We have that  $\neg(\exists i \in tx.\text{inputs}, i.\text{output.value} = \text{ttt})$ . Since an additional token ttt cannot be minted or burned, we also conclude  $\neg(\exists o \in tx.\text{outputs}, o.\text{value} = \text{ttt})$ . By  $\pi(utxo) = (a, b)$ , the  $utxo$  state contains a unique output with token ttt, datum  $(a, b)$ , and  $\text{toggleVal}_N(\text{myRef})$  validator. By  $\pi(tx) = \star$ , that output was not spent, and still exists in the UTXO set  $utxo'$ . By Assumption 2, since  $tx \neq \text{myRef.fst}$ , the reference  $\text{myRef}$  is not added to the inputs of  $utxo'$ . So, that  $\pi(utxo') = \pi(utxo) = (a, b)$ . Then,

$$(\star, \pi(utxo), \pi(tx), \pi(utxo')) = (\star, (a, b), \star, (a, b)) \in \text{TOGGLE}$$

- (i)  $\pi(tx) = \text{toggle}$ : Implies that  $\exists i \in tx.\text{inputs}, i.\text{output.value} = \text{ttt}$ . This means that that the (unique) UTXO containing ttt is spent, and no ttt tokens are minted or burned. Therefore, the transaction must create a single output in  $utxo'$  with that token. The script  $\text{toggleVal}_N(\text{myRef})$  must be run because ttt is spent and, by  $\pi(utxo) = (a, b)$ , was locked by  $\text{toggleVal}_N(\text{myRef})$ . Because  $\text{toggleVal}_N(\text{myRef})$  must validate, the unique new output containing ttt must have a datum  $(b, a)$ , the same validator. Again,  $\text{myRef}$  is not added to the inputs of  $utxo'$  by assumption. We conclude that  $\pi(utxo') = (b, a)$ . Then,

$$(\star, \pi(utxo), \pi(tx), \pi(utxo')) = (\star, (a, b), \star, (b, a)) \in \text{TOGGLE}$$

$$\begin{aligned}
\text{ttt} &:= \{\text{toggleTT}_N(\text{myRef}) \mapsto \{\text{encode}(\text{toggleVal}_N(\text{myRef})) \mapsto 1\}\} \\
\text{tta} &:= \{\text{toggleTT}_D(\text{myRef}) \mapsto \{(\text{encode}(\text{toggleVal}_D(\text{myRef})) \mapsto "a") \mapsto 1\}\} \\
\text{ttb} &:= \{\text{toggleTT}_D(\text{myRef}) \mapsto \{(\text{encode}(\text{toggleVal}_D(\text{myRef})) \mapsto "b") \mapsto 1\}\} \\
\pi_n(\text{utxo}) &:= \begin{cases} (a, b) & \text{if } \text{myRef} \notin \{i \mid i \mapsto o \in \text{utxo}\} \\ & \wedge \exists! (i \mapsto o) \in \text{utxo}, \text{ttt} = o.\text{value} \\ & \wedge o.\text{validator} = \text{toggleVal}_N(\text{myRef}) \wedge o.\text{datum} = (a, b) \\ \star & \text{otherwise} \end{cases} \\
\pi_{\text{Tx},n}(\text{tx}) &:= \begin{cases} \text{toggle} & \text{if } \exists i \in \text{tx.inputs}, i.\text{output.value} = \text{ttt} \\ \star & \text{otherwise} \end{cases}
\end{aligned}$$

■ **Figure 7** TOGGLE thread tokens and naive projections

**Distributed implementation.** The proof for the distributed implementation is similar to the one for the naive implementation, except we must keep track of two inputs and two outputs containing two thread tokens. A transaction updating the state must necessarily spend both outputs containing each of the tokens, and that the new UTxOs containing them are such that the datum in UTxO with token *tta* now has the boolean that was in the datum of *ttb*, and vice-versa. Both must still be locked by  $\text{toggleVal}_N(\text{myRef})$ .