Structured Contracts in the EUTxO Ledger Model *

Polina Vinogradova
IOG, Singapore

Manuel Chakravarty
IOG, Singapore

Philip Wadler
IOG, Singapore
University of Edinburgh, UK

James Chapman
IOG, Singapore

Tudor Ferariu
University of Edinburgh, UK

Michael Peyton Jones
IOG, Singapore

Jacco Krijnen
Utrecht University, Netherlands

Orestis Melkonian
IOG, Singapore
University of Edinburgh, UK

Abstract

The extended UTxO ledger is a kind of a UTxO-based cryptocurrency ledger that supports the use of smart contract scripts to specify permissions for performing certain actions, such as spending a UTxO or minting assets. There have been some attempts to standardize the implementation of stateful programs using this infrastructure, with varying degrees of success. In this work, we present a model of stateful computation on the ledger, which we call the structured contract framework.

Using a small-step semantics approach to program specification, the structured contracts framework establishes a simulation relation between the ledger state transition system and the transition system of the program being specified. This gives users the tools for demonstrating the correctness of a transition system's ledger implementation. We argue that the framework is versatile and amenable to formal verification by presenting several distinctive examples. In particular, we demonstrate how the framework allows for building multiple implementations of the same specification (some of which may be distributed across the UTxO set), as well as expressing the policy of an NFT as a stateful contract with a specific invariant.

2012 ACM Subject Classification Security and privacy → Formal methods and theory of security

Keywords and phrases blockchain, ledger, smart contract, formal verification, specification, transition system, UTxO, semantics

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Many modern cryptocurrency blockchains are smart contract-enabled, meaning, generally, that they provide some 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. In particular, on account-based platforms such as Ethereum [4] and Tezos [13], smart contracts are inherently stateful, and their states can be updated by transactions. Smart contracts in the extended UTxO (EUTxO) model, such as Cardano [16] and [7], on the other hand, take the form of boolean predicates on the data of the transaction executing them, and are inherently stateless. In this model, transactions specify all the

* This work was supported by Input Output (iohk.io) through their funding of the Edinburgh Blockchain Technology Lab.

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.

Just within the UTxO model, there are multiple existing approaches to implementing and formalizing specific designs of stateful programs running on the ledger [9] [11] [5]. There are also languages, such as BitML [3], a DSL for specifying contracts that regulate transfers of Bitcoin among participants. There is also much existing work on smart contract formal verification. However, currently, there are no principled standard practices for specifying and implementing stateful programs on the EUTxO ledger. In this work, we propose to re-use an existing ledger specification standard to specify stateful contracts.

Like many prominent platforms [13] [4] [21] [29] [28], the Cardano implementation of the EUTxO ledger [16] 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 rigour of its operational small-step semantics specification [24]. We propose the *structured contract framework* (SCF) as an extension of this approach to 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 [9], the 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 expressing invariants (or safety properties) of contracts for which it was not previously done. For example, we can express invariants of stateful contracts that are implemented across multiple different UTxOs. We can also express invariants on the totality of tokens under a specific policy by interpreting it as the state of a structured contract. We argue that the SCF constitutes a new, 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 definition of a simplified EUTxO ledger using small-steps semantics ;
(ii) a definition of the structured contract formalism (SCF) ;
(iii) a case study expressing the minting policy of a single NFT as a structured contract ;
(iv) a case study demonstrating the use the SCF to define two distinct ledger implementations of a single specification, including one that is distributed across multiple UTxO entries and interacting scripts ;

2 EUTxO ledger model

The EUTxO ledger model is 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 [9]. We demonstrate here that it can be expressed as a labelled transition system with the UTxO set as its state, specified in small-step semantics [24], similar to the specification of a deployed EUTxO ledger [16]. The transaction validation rules become constraints of the UTxO state transition rule. Note that while in a realistic system, transactions are applied to the ledger in blocks, we abstract away block structure here for simplicity.

2.1 Transition Relation Specification

For some Env, State and Input, the transition relation TRANS is a subset of a 4-tuple :

$$\_ \vdash \_ \xrightarrow[\text{TRANS}]{} \_ \subseteq (\mathsf{Env} \times \mathsf{State} \times \mathsf{Input} \times \mathsf{State})$$

Membership $(env,\ s,\ inp,\ s') \in \mathsf{TRANS}$ is also denoted by

$$env \vdash s \xrightarrow[\text{TRANS}]{inp} s'$$

where each component serves the following purpose in the state transition :

(i) $env \in \mathsf{Env}$ is the *environment* ;
(ii) $s \in \mathsf{State}$ is the *starting state* to which an input is applied ;
(iii) $i \in \mathsf{Input}$ is the *input* ;
(iv) $s' \in \mathsf{State}$ is the *end state* computed from the start state as the result of the application of the input in the given environment.

A specification TRANS is made up of one or more *transition rules*. The only 4-tuples that are members of TRANS are those that satisfy the preconditions in one of its transition rules.

Input, environment, and labelled 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 system is deterministic, 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 [14]. In particular, input comes from users, e.g. a transactions they submit. The environment, on the other hand, is outside the user's control, such as the blockchain time. Timekeeping is done at the consensus and block level, which is outside the scope of this work.

2.2 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 [9]. 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 Figures 4 5. Notation we use that is outside conventional set-theoretic notation is listed in Figure 3, and explained in the text. Here, $\mathbb{B}, \mathbb{N}, \mathbb{Z}$ denote the type of Booleans, natural numbers, and integers, respectively. Some types described below are mutually recursive, so there is no natural order in which to describe them.

Value. We define $\mathsf{Value} := \mathsf{FinSup}[\mathsf{PolicyID}, \mathsf{FinSup}[\mathsf{TokenName}, \mathsf{Quantity}]]$. A term of this type is a bundle of multiple kinds of assets. The type of an identifier of a class of fungible assets is given by $\mathsf{AssetID} := \mathsf{PolicyID} \times \mathsf{TokenName}$. $\mathsf{Quantity} := \mathbb{Z}$ is an integral value. For a given $v \in \mathsf{Value}$, the nested map associates a quantity of each asset with a given asset ID to its asset ID. The quantities of all assets with IDs not included in v are 0. A group is formed by Value, with the group operation denoted by $+$, and the empty map as the zero of the group. Value also forms a partial order [10]. The components of AssetID are :

(i) a script of type $\mathsf{PolicyID} := \mathsf{Script}$, which executed any time a transaction is minting assets with this minting policy ;
(ii) a $\mathsf{TokenName} := [\mathsf{Char}]$, selected by the user, and used to differentiate assets that have the same minting policy.

To define a Value with a single asset ID and quantity one, we use the following function,

$$\mathsf{oneT}\ \mathsf{policy}\ \mathsf{tokenName} := \{\ \mathsf{policy} \mapsto \{\mathsf{tokenName} \mapsto 1\}\ \}$$

UTxO set. The type of the UTxO set is a finite map $\mathsf{UTxO} := \mathsf{OutputRef} \mapsto \mathsf{Output}$. The type of the key of this key-value map is $\mathsf{OutputRef} = \mathsf{Tx} \times \mathsf{Ix}$, with $\mathsf{Ix} = \mathbb{N}$. A $(tx, ix) \in \mathsf{OutputRef}$ is called an output reference. It consists of a transaction tx that created the output to which it points, and index of ix. The index is the location of particular output in the list of outputs of that transaction. The pair uniquely identifies a transaction output.

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

Data. Data is a type used for representing data encoded in a specific way, is similar in structure to a CBOR encoding, for details, see [18]. It was introduced in the definition of the EUTxO ledger [9], and also used in its Cardano implementation [25]. Data of this type is passed as arguments to scripts. The types $\mathsf{Datum} := \mathsf{Data}$ and $\mathsf{Redeemer} := \mathsf{Data}$ are both synonyms for the Data type. Conversion functions are required in order for a script to interpret Data-type inputs as the datatypes it is expecting. When the context is clear, the decoding function is called fromData, and the encoding one is toData.

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

Transactions. The data structure Tx specifies a set of updates to the UTxO set. A transaction $tx \in \mathsf{Tx}$ contains (i) a set of *inputs* each referencing entries in the UTxO set that the transaction is removing (spending), with their corresponding redeemers, (ii) a set of outputs, which get entered into the UTxO set with the appropriately generated output references, (iii) a pair of slot numbers representing the validity interval of the transaction, (iv) a Value being minted by the transaction, (v) a redeemer for each of the minting policies being executed, and (vi) the set 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 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 use set-theoretic notation for script pseudocode.

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.

We denote evaluation of scripts that control minting tokens and spending outputs by $[\![\_]\!]$, followed by the script arguments. For example, evaluating a minting policy script s to validate minting tokens under policy p, run by transaction tx with redeemer r, is denoted by $[\![s]\!]\ r\ (tx,\ p)$ Evaluating a script q to validate spending input $i \in \mathsf{inputs}\ tx$ with datum d and redeemer r, we write $[\![q]\!]\ d\ r\ (tx,\ i)$

2.3 Ledger Transition Semantics

Permissible updates to the UTxO set are given by the LEDGER transition system,

$$\_ \vdash \_ \xrightarrow[\text{LEDGER}]{\_} \_ \subseteq (\mathsf{Slot} \times \mathsf{UTxO} \times \mathsf{Tx} \times \mathsf{UTxO})$$

The output of the function $\mathsf{checkTx} : \mathsf{Slot} \times \mathsf{UTxO} \times \mathsf{Tx} \to \mathbb{B}$ determines whether a transaction is valid in a given state and environment. The output of $\mathsf{checkTx}\ (slot, utxo, tx)$ is given by the conjunction of the following checks, which are consistent with the previously specified EUTxO validation rules [9] :

(i) The transaction has at least one input : $\mathsf{inputs}\ tx \neq \{\}$
(ii) The current slot is within transaction validity interval : $slot \in \mathsf{validityInterval}\ tx$
(iii) All outputs have positive values : $\forall o \in \mathsf{outputs}\ tx,\ \mathsf{value}\ o > 0$
(iv) All output references of transaction inputs exist in the UTxO :
$\forall (oRef,\ o) \in \{(\mathsf{outputRef}\ i,\ \mathsf{output}\ i) \mid i \in \mathsf{inputs}\ tx\},\ oRef \mapsto o \in utxo$
(v) Value is preserved : $\mathsf{mint}\ tx + \sum_{i \in \mathsf{inputs}\ tx,\ (\mathsf{outputRef}\ i) \mapsto o \in utxo} \mathsf{value}\ o = \sum_{o \in \mathsf{outputs}\ tx} \mathsf{value}\ o$
(vi) No output is double-spent : $\forall\ i_1, i \in \mathsf{inputs}\ tx,\ \mathsf{outputRef}\ i = \mathsf{outputRef}\ i_1 \Rightarrow i = i_1$
(vii) All inputs validate : $\forall\ (i, o, r) \in \mathsf{inputs}\ tx,\ [\![\mathsf{validator}\ o]\!](\mathsf{datum}\ o,\ r,\ (tx, (i, o, r)))$
(viii) Minting redeemers are present : $\forall\ pid \mapsto \_ \in (\mathsf{mint}\ tx),\ \exists (pid, \_) \in \mathsf{mintScsRdmrs}\ tx$
(ix) All minting scripts validate : $\forall\ (p, r) \in \mathsf{mintScsRdmrs}\ tx,\ [\![p]\!](r, (tx, p))$
(x) All signatures are present : $\forall\ (pk \mapsto s) \in \mathsf{sigs}\ tx,\ \mathsf{checkSig}(tx, pk, s)$

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

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

$$\text{ApplyTx}\ \frac{\mathsf{checkTx}\ (slot,\ utxo,\ tx)}{slot \vdash (\ utxo\ ) \xrightarrow[\text{LEDGER}]{tx} (\ \boldsymbol{utxo'}\ )} \tag{1}$$

The value $utxo'$ is calculated (deterministically) by removing the UTxO entries in $utxo$ corresponding to the output references of the transaction inputs, and adding the outputs of the transaction tx to the UTxO set with correctly generated output references. The function getORefs computes a UTxO set containing only the output references of transaction inputs, paired with the outputs contained in those inputs. The function mkOuts computes a UTxO set containing exactly the outputs of tx, each associated to the key (tx, ix). The index ix is the place of the associated output in the list of tx outputs. For details, see 5.

Unlike the *empty* ledger state, which is an initial ledger state in existing EUTxO definitions [9], the ledger model we define by LEDGER does not have an initial ledger state. For this reason, we make additional assumptions about transactions, which could be verified as properties or valid ledger traces starting from some valid initial state. This is the subject of future work.

3 Simulations and the structured contract formalism

The programs we are interested in specifying for the purpose of this work are those that *run on the ledger*. Intuitively, a stateful program is 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. We now formalize the notion of correctly implementing a stateful program on the ledger using smart contract scripts.

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, or remove some UTxO entries. 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 corresponding data in the UTxO state. The state of a program on the ledger is *observed* by applying a projection function to the ledger state which aggregates the relevant data.

A structured contract includes a specification, and a projection function that computes the contract state from a given ledger state. It also requires a proof of the integrity of the contract's implementation, establishing a simulation relation between it and the ledger. That is, that the scripts controlling the ledger data returned by the projection function ensure the evolution of that data is according to the contract specification. For example, the projection function may return the value and datum of a UTxO containing a special NFT. This pair of makes up the state of a given stateful contract. The script locking that UTxO must guarantee that upon being spent, the NFT is always placed into a new UTxO with a particular datum and value, which are computed according to the contract specification. This approach to ensuring adherence to specification is called a *thread token* mechanism [9], and we will elaborate on it in later sections.

3.1 Simulations

We instantiate the definition a *simulation* [20] with labelled state transition systems expressed as small-step semantics specifications.

Simulation definition. Let TRANS and STRUC be small-step labelled transition systems. A *simulation* of TRANS in STRUC, denoted by

$$(\mathsf{STRUC}, \sim_{\mathsf{TRANS,STRUC}}, \sim_{\rightarrow,\mathsf{TRANS,STRUC}}) \succeq \mathsf{TRANS}$$

Consists of of the following types together with the following relations :

$$\_ \vdash \_ \xrightarrow[\mathrm{TRANS}]{=} \_ \subseteq (\mathsf{Env}_{\mathsf{TRANS}} \times \mathsf{State}_{\mathsf{TRANS}} \times \mathsf{Input}_{\mathsf{TRANS}} \times \mathsf{State}_{\mathsf{TRANS}})$$

$$\_ \vdash \_ \xrightarrow[\mathrm{STRUC}]{=} \_ \subseteq (\mathsf{Env}_{\mathsf{STRUC}} \times \mathsf{State}_{\mathsf{STRUC}} \times \mathsf{Input}_{\mathsf{STRUC}} \times \mathsf{State}_{\mathsf{STRUC}})$$

$$\_ \sim_{\mathsf{TRANS,STRUC}} \_ : \mathsf{State}_{\mathsf{TRANS}} \times \mathsf{State}_{\mathsf{STRUC}} \to \mathsf{Bool}$$

$$\_ \sim_{\rightarrow,\mathsf{TRANS,STRUC}} \_ : (\mathsf{Env}_{\mathsf{TRANS}} \times \mathsf{Input}_{\mathsf{TRANS}}) \times (\mathsf{Env}_{\mathsf{STRUC}} \times \mathsf{Input}_{\mathsf{STRUC}}) \to \mathsf{Bool}$$

such that the following holds :

$$\sim> \frac{(e,\ i) \sim_{\rightarrow,\mathsf{TRANS,STRUC}} (e', j) \qquad u \sim_{\mathsf{TRANS,STRUC}} s \qquad e \vdash (\ s\ ) \xrightarrow[\mathrm{TRANS}]{i} (\ s'\ )}{u' \sim_{\mathsf{TRANS,STRUC}} s' \qquad e' \vdash (u) \xrightarrow[\mathrm{STRUC}]{j} (\boldsymbol{u'})} \tag{2}$$

We write $\sim$ instead of $\sim_{\mathsf{TRANS,STRUC}}$ whenever the subscript is unambiguous. This relation states that a if a valid state $s \in \mathsf{State}_{\mathsf{STRUC}}$ is associated with a valid UTxO state u, then any ledger transition starting in $s \in \mathsf{State}_{\mathsf{TRANS}}$ is necessarily associated with a valid transition starting in state s. Note that $\sim$ is a *proof obligation* that must be fulfilled as part of the definition, which *does not define a rule*. One can construct pairs of transition systems with $\sim$, $\sim_{\rightarrow}$ relations for which this proof obligation cannot be fulfilled. However, *if* it is possible, we have a simulation of TRANS in STRUC.

3.2 Structured contracts.

The simulation definition we give is general, however, the rest of this work is geared towards reasoning about the programmable parts of the ledger, i.e. those where the permissions are controlled by user-defined scripts. For this reason, we define a particular class of simulations of LEDGER. First of all, since scripts are not allowed to inspect block-level data (i.e. the current slot number), we fix the environment of the structured contract specification to be a singleton type $\{\star\}$. Secondly, $\sim$ must be a partial function, rather than a relation, which computes a unique contract state for a given UTxO state (or fails, returning $\{\star\}$). The relation between arrows must also be expressible as a function which computes a specific contract input value for any given transaction.

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

$$utxo \sim s := (\pi_{\mathsf{STRUC}}\ utxo = s)$$

$$(slot,\ tx) \sim_{\rightarrow,\mathsf{LEDGER,STRUC}} (\star, i) := (\pi_{\mathsf{Tx,STRUC}}\ tx = i)$$

Discussion. We sometimes denote $\pi_{\mathsf{Tx,STRUC}}$ and π_{STRUC} by π and π_{Tx}, respectively, when the context is clear. We also denote the structured contracts by $(\mathsf{STRUC}, \pi_{\mathsf{STRUC}}, \pi_{\mathsf{Tx,STRUC}}) \succeq \mathsf{LEDGER}$. We say that a *utxo* is *valid for* STRUC whenever $\pi\ utxo \neq \star$.

It is possible for transactions to update the ledger state, but not the STRUC state. There is no a special case for this in the simulation relation, and the STRUC specification rules must allow trivial steps if such ledger transactions are possible. The STRUC specification may not be deterministic, however, since the LEDGER is deterministic, given a ledger step, there is a unique step in STRUC corresponding to the ledger one. We do not assume that a valid contract state can be computed from an arbitrary UTxO state. For this reason, the function π_{STRUC} is partial. For example, it is possible that two NFTs exist in a given ledger state. When programmed correctly, an NFT minting policy would not allow this to happen. To reason about properties of such a policy, we must exclude ledger states where the NFT uniqueness condition has a been violated in the start state.

Requiring $\sim>$ to be a bisimulation [20] between STRUC and LEDGER is too restrictive, and excludes a lot of interesting contracts. Defining a class of structured contracts for which this is possible is a difficult problem, and we leave it for future work.

4 NFT minting policy as a structured contract

Our first structured contract example is expressing 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, we are able to express

and prove the defining property of a specific NFT : at most one such token can exist on the ledger. Instantiating an NFT as a structured contract and expressing this property allows us to mimic something that is quite naturally expressed for account-based blockchains with stateful NFT contracts, such as the ERC-721 [12], but poses a challenge for EUTxO ledger program analysis.

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.

$$i := \{\ \mathsf{myNFTPolicy} \mapsto \mathit{tkns} \in \mathsf{mint}\ \mathit{tx}\ \}$$

$$\text{UpdateNFTTotal}\ \frac{\{\} \subseteq s \subseteq s + i \subseteq \mathsf{oneT}\ \mathsf{myNFTPolicy}\ [\,]}{\vdash (\ s\ ) \xrightarrow[\mathrm{NFT}]{\mathit{tx}} (\ s + i\ )} \tag{3}$$

The specification states that the only allowed transitions are (i) a constant one, and (ii) adding a single NFT, given by oneT myNFTPolicy [ ], to the state if one does not yet exist. An NFT whose total ledger quantity obeys this specification can never be burned, and must be the only token under its policy. 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,

$$\pi\ \mathit{utxo}\ := \begin{cases} s & \text{if } (s = \mathsf{oneT}\ \mathsf{myNFTPolicy}\ [\,] \ \wedge\ \neg\, \mathsf{hasRef}) \ \vee\ s = 0 \\ \star & \text{otherwise} \end{cases}$$

where

$$s := \{\ p \mapsto \mathit{tkns} \mid p \mapsto \mathit{tkns} \in \sum_{\_ \mapsto \mathit{out} \in \mathit{utxo}} \mathsf{value}\ \mathit{out},\ p = \mathsf{myNFTPolicy}\ \}$$

$$\mathsf{hasRef} := (\mathsf{myNFTRef} \mapsto \_ \in \mathit{utxo})$$

Here, $\pi\ \mathit{utxo}$ returns a non-$\star$ result when either no tokens under the myNFTPolicy policy exist, or only the token oneT myNFTPolicy [ ] exists under this policy, and the anchor myNFTRef is not in the UTxO. We define the policy,

$$\mathsf{myNFTPolicy} := \mathsf{mkMyNFTPolicy}\ \mathsf{myNFTRef}$$

$$[\![\mathsf{mkMyNFTPolicy}\ \mathit{myRef}]\!]\ \_\ (\mathit{tx},\ \mathit{pid}) := \exists\ (\mathit{myRef}, \_, \_) \in \mathsf{inputs}\ \mathit{tx} \ \wedge\ \mathsf{oneT}\ \mathit{pid}\ [\,] \in \mathsf{mint}\ \mathit{tx}$$

To prove $\sim>$ for the NFT contract (see Appendix A.0.0.1 for a proof sketch), we need to make an additional assumption stating that a transaction which adds myNFTRef to the UTxO cannot be valid again later, once the NFT has been minted :

NFT re-minting protection. For any $(\mathit{slot},\ \mathit{utxo},\ \mathit{tx},\ \mathit{utxo}') \in \mathsf{LEDGER}$, with $\pi\ \mathit{utxo} = \mathsf{oneT}\ \mathsf{myNFTPolicy}\ [\,]$, necessarily $\mathit{tx} \neq \mathsf{fst}\ \mathsf{myNFTRef}$.

Under reasonable assumptions about the initial state of the ledger, this property should be consequence of replay protection, which is a trace-based safety property of the EUTxO LEDGER. A full treatment of traces and properties is the subject of future work. So, to ensure correct program behaviour, we introduce the above assumption.

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* $\neq \star$,

$$\mathsf{pi}\ utxo \subseteq \mathsf{oneT}\ \mathsf{myNFTPolicy}\ [\,]$$

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 state valid for NFT to a state which is not valid for NFT. That is, with $(slot, utxo, tx, utxo') \in \mathsf{LEDGER}$, the updated state $\pi\ utxo'$ must also always have at most one NFT under 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 any relevant characteristics, such as space usage, or parallelizability.

5.1 Toggle specification

We define 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 be $\{\mathsf{toggle}\} \cup \star$. The two booleans in the state are both flipped by the input toggle, and unchanged by $\star$. We define the transition system TOGGLE :

$$\text{DoNothing}\ \frac{}{\vdash (\ x, y\ ) \xrightarrow[\text{TOGGLE}]{\star} (\ x, y\ )} \tag{4}$$

$$\text{Toggle}\ \frac{}{\vdash (\ x, y\ ) \xrightarrow[\text{TOGGLE}]{toggle} (\ y, x\ )} \tag{5}$$

5.2 Toggle implementations

We present two implementations of the TOGGLE specification. The *naive implementation* is one that 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 pair that is the TOGGLE state.

Thread token scripts. We use the *thread tokens* mechanism [9] to ensure the unique identification of the UTxO (or pair of UTxOs) from which the contract state is computed. In both implementations, the thread token minting policy guarantees that they are generated in quantity 1 by a transaction that spends a specific output reference myRef, similar to the NFT policy in Section 4.

For the naive implementation, one thread token NFT 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 contract, which is passed as a parameter to the minting policy. This contract (discussed below) ensures the correct evolution of the contract state. The datum in the UTxO containing the thread token is the initial state of the contract encoded as a pair of booleans

(by the partial decoder function $\mathsf{fromData}_N : \mathsf{Data} \to \mathbb{B} \cup \{\star\}$). It can be any pair of correctly encoded booleans. See Figure 6 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 UTxO, locked by the same contract (discussed below). The datum in each must be decodeable (by $\mathsf{fromData}_N : \mathsf{Data} \to (\mathbb{B} \times \mathbb{B}) \cup \{\star\}$) as a boolean. See Figure 7 for the policy pseudocode.

Validator scripts. We require different UTxO-locking scripts for our two distinct implementations. Both scripts serve the following function : when toggle redeemers are specified, the script must ensure that the 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. For the naive version, the datum in the new UTxO containing the thread token must decode as a pair of booleans whose order is reversed as compared to the booleans encoded in the datum of the spent UTxO that previously contained the thread token. We define it by :

$$
\begin{aligned}
[\![\mathsf{toggleVal}_N\ \mathsf{myRef}]\!]\ (b, b')\ r\ (tx,\ i)\ :=\ & ttt\ =\ \mathsf{value}\ (\mathsf{output}\ i)\ \wedge\ r = \mathsf{toggle} \\
& \wedge\ \exists\, o\ \in\ \mathsf{outputs}\ tx,\ (b',\ b)\ =\ (\mathsf{datum}\ o) \\
& \wedge\ (\mathsf{validator}\ o\ =\ vi)\ \wedge\ (ttt = \mathsf{value}\ o) \\
& \textbf{where} \\
& vi := \mathsf{validator}\ (\mathsf{output}\ i) \\
& ttt := \mathsf{oneT}\ (\mathsf{toggleTT}_N\ \mathsf{myRef}\ vi)\ (\mathsf{encode}\ vi)
\end{aligned}
$$

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

The distributed implementation script ensures that both the thread token-containing UTxOs are spent simultaneously. Then, it checks that the booleans in the datums are switched places : the one that was in the UTxO with token "a" must now be in a new UTxO with token "b", and vice-versa. The validator script is given in Figure 1.

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 UTxO scripts. The input projection function returns toggle whenever a transaction contains the thread tokens in its input(s), and $\star$ otherwise. For details, see Figures 8 2.

In Appendix A.0.0.2, 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. To avoid duplication of thread tokens, we again need to make the additional assumption that for any $(slot,\ utxo,\ tx,\ utxo') \in \mathsf{LEDGER}$, with $\pi\ utxo\ = (a, b)$, necessarily $tx \neq \mathsf{fst}\ \mathsf{myRef}$.

TOGGLE **property example.**

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

This 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.

$$
\begin{aligned}
&[\![\mathsf{toggleVal}_D\ \mathsf{myRef}]\!]\ b\ \mathsf{toggle}\ (tx,\ i)\ :=\ (tta\ =\ \mathsf{value}\ (\mathsf{output}\ i)) \Rightarrow \\
&\qquad \exists\ o,\ o' \in\ \mathsf{outputs}\ tx,\ i' \in \mathsf{inputs}\ tx, \\
&\qquad \mathsf{validator}\ o\ =\ \mathsf{validator}\ o'\ =\ vi \wedge \\
&\qquad \mathsf{tta}\ =\ \mathsf{value}\ o \wedge ttb\ =\ \mathsf{value}\ o' \wedge \mathsf{value}\ (\mathsf{output}\ i')\ =\ ttb \\
&\qquad \mathsf{datum}\ o\ =\ \mathsf{datum}\ (\mathsf{output}\ i') \wedge \mathsf{datum}\ o'\ =\ \mathsf{datum}\ (\mathsf{output}\ i) \\
&\quad \wedge \\
&\quad (ttb\ =\ \mathsf{value}\ (\mathsf{output}\ i)) \Rightarrow \\
&\qquad \exists\ o,\ o' \in\ \mathsf{outputs}\ tx,\ i' \in \mathsf{inputs}\ tx, \\
&\qquad \mathsf{validator}\ o\ =\ \mathsf{validator}\ o'\ =\ vi \wedge \\
&\qquad \mathsf{tta}\ =\ \mathsf{value}\ o \wedge ttb\ =\ \mathsf{value}\ o' \wedge \mathsf{value}\ (\mathsf{output}\ i')\ =\ tta \\
&\qquad \mathsf{datum}\ o\ =\ \mathsf{datum}\ (\mathsf{output}\ i) \wedge \mathsf{datum}\ o'\ =\ \mathsf{datum}\ (\mathsf{output}\ i') \\
&\quad \wedge \\
&\quad (tta\ =\ \mathsf{value}\ (\mathsf{output}\ i)) \vee (ttb\ =\ \mathsf{value}\ (\mathsf{output}\ i)) \\
&\quad \textbf{where} \\
&\qquad vi := \mathsf{validator}\ (\mathsf{output}\ i) \\
&\qquad tta := \mathsf{oneT}\ (\mathsf{toggleTT}_D\ \mathsf{myRef}\ vi)\ (\mathsf{encode}\ vi\ +\!+\ "a") \\
&\qquad ttb := \mathsf{oneT}\ (\mathsf{toggleTT}_D\ \mathsf{myRef}\ vi)\ (\mathsf{encode}\ vi\ +\!+\ "b")
\end{aligned}
$$

Figure 1 TOGGLE validator script for the distributed implementation

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

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

Figure 2 TOGGLE distributed projections

6 Related work

Scilla [8] 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 building stateful contracts and formally studying their behaviour. However, the contribution of this work is a framework for stateful contract implementation on the EUTxO ledger.

CoSplit, presented in [22], 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]. So, no additional work is required to ensure commutativity when updating parts of a contract with distributed state in multiple steps.

The Bitcoin Modelling Language (BitML) [3] enables the definition of smart contracts in a particular restricted class of state machines on the Bitcoin ledger. The BitML state machines are less expressive than the class of specifications considered in our model. The goal here is again is similar to ours - to guarantee that the behaviour of certain state machines is in accordance with the changes made by valid transactions, i.e. soundness. The approach we present in this work is more general (applying to arbitrary Turing complete contracts), but does not, at this time, support automation for constructing implementations and verifying their properties.

VeriSolid [17] synthesises Solidity smart contracts from a state machine specification, and verifies temporal properties of the state machine using CTL. The underlying ledger model for VeriSolid is an account-based model, rather than the EUTxO model we work with. 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 structured contracts.

The K framework [23] is a unifying formal semantics framework for all programming languages, which has been used as a tool to perform audits of smart contracts [27], as well as specifying Solidity operational semantics [15]. Auditing is a common approach to smart contract verification [1] [6]. We present a framework which, when instantiated, provides certain guarantees about the implemented contracts.

7 Conclusion

The key contribution of this work is a new, versatile, and principled approach to modelling stateful contracts on the EUTxO ledger. We generalize the application of *simulation* for demonstrating integrity of consolidated (single-UTxO) state [9] to simulation of arbitrarily implemented state machines with varying ledger representations.

We do this by introducing a formalism for modelling stateful contracts, which we call the structured contract framework. It is instantiated by first specifying a labelled transition system expressed in terms of small-step semantics. Then, functions must be defined that compute the contract state and input for a given ledger state and transaction, respectively. The functions include information about the implementing scripts used to control evolution of the relevant ledger data. Finally, a proof obligation of the integrity of the implementation must be fulfilled for the given specification and projection functions.

This approach opens up the possibility of formal verification of the behaviour a much larger class of contracts, which would previously have been implemented ad-hoc. We presented examples of such contracts and safety properties satisfied by these examples. These examples include a distributed implementation of a contract, and a stateful model of an NFT policy. A full analysis of structured contract behaviour in terms of trace-based properties and their expression at the ledger level is the subject of future work.

An Agda mechanization of the SCF has been completed [19], and the mechanization of the examples we presented in under way. The completion of these examples, as well as of more realistic ones, is a natural next step. We also intend to build a mechanization of this work integrated with the Agda mechanization of small-steps semantics specification of the deployed Cardano ledger [26].

A Appendix

$$\mathbb{H} = \bigcup_{n=0}^{\infty} \{0,1\}^{8n} \qquad \text{the type of bytestrings}$$

$$\star : \{\star\} \qquad \text{the one-element set, and its one inhabitant}$$

$$a : A \cup \{\star\} \qquad \text{maybe type over } A$$

$$\mathsf{fst} : (A \times B) \to A \qquad \text{first projection}$$

$$(a, b) : \mathsf{Interval}[A] \qquad \text{intervals over a totally-ordered set } A$$

$$Key \mapsto Value \subsetneq \{\, k \mapsto v \mid k \in Key,\ v \in Value \,\} \qquad \text{finite map with unique keys}$$

$$[a1; ...; ak] : [C] \qquad \text{finite list with terms of type } C$$

$$++ : ([C], [C]) \to [C] \qquad \text{list concatenation}$$

$$h :: t : [C] \qquad \text{list with head } h \text{ and tail } t$$

$$V : \mathsf{FinSup}[K, M] \qquad \text{the type of finitely supported functions from a type } K \text{ to a monoid } M$$

$$\mathsf{map} : (A \to B) \to [A] \to [B] \qquad \text{apply first argument to every element in given list}$$

$$\mathsf{map} : (A \to B) \to \mathbb{P}\ A \to \mathbb{P}\ B \qquad \text{apply first argument to every element in given set}$$

Figure 3 Notation

A.0.0.1 ∼> proof sketch for NFT.

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

(i) If $i = \{\ \mathsf{myNFTPolicy} \mapsto tkns \in \mathsf{mint}\ tx\ \} = 0$, by preservation of value (rule (v) in Section 2.3), the amount s of tokens under myNFTPolicy remains unchanged in $utxo'$. If $s = 0$, we get $s' = 0 + 0 = 0$. Then, $\pi\ utxo'$ is defined, and $\pi\ utxo = \pi\ utxo'$. If $s \neq 0$, by assumption in Section 4, we conclude that tx cannot add an output with reference myRef in the $utxo'$. Since, by $\pi\ utxo \neq \star$, we know that myRef was not in $utxo$ either, we conclude that there is no output with myRef in $utxo'$. So, $\pi\ utxo = \pi\ utxo'$.

(ii) If $i \neq 0$, tokens under myNFTPolicy are being minted, and the policy must be checked by ledger rule (ix) in Section 2.3. Necessarily, by myNFTPolicy, $i = \mathsf{oneT}\ pid\ [\,]$. If $s = 0$, $s' = s + i = i$ 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 tx, so that it is not contained in $utxo'$. By assumption in Section 4, it is also not added back by tx to $utxo'$. Then, $\pi\ utxo' \neq \star$, and is equal to i. If $s \geq 0$, an output with reference myRef is not in $utxo'$. So, myNFTPolicy fails, and the tx is not valid on the ledger.

A.0.0.2 ∼> relation proof sketch

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

LEDGER PRIMITIVES

$[\![\_]\!] : \mathsf{Script} \to \mathsf{Datum} \times \mathsf{Redeemer} \times \mathsf{ValidatorContext} \to \mathbb{B}$	applies a script to its arguments
$[\![\_]\!] : \mathsf{Script} \to \mathsf{Redeemer} \times \mathsf{PolicyContext} \to \mathbb{B}$	applies a script to its arguments
$\mathsf{checkSig} : \mathsf{Tx} \to \mathsf{pubkey} \to \mathbb{H} \to \mathbb{B}$	checks that the given PK signed the transaction (excl. signatures)

DEFINED TYPES

$$
\begin{aligned}
\mathsf{Signature} &= \mathsf{pubkey} \mapsto \mathbb{H} \\
\mathsf{OutputRef} &= (\mathsf{id} : \mathsf{Tx}, \mathsf{index} : \mathsf{Ix}) \\
\\
\mathsf{Output} &= (\mathsf{validator} : \mathsf{Script}, \\
&\quad \mathsf{value} : \mathsf{Value}, \\
&\quad \mathsf{datum} : \mathsf{Data}) \\
\\
\mathsf{Input} &= (\mathsf{outputRef} : \mathsf{OutputRef}, \\
&\quad \mathsf{output} : \mathsf{Output}, \\
&\quad \mathsf{redeemer} : \mathsf{Redeemer}) \\
\\
\mathsf{Tx} &= (\mathsf{inputs} : \mathbb{P}\ \mathsf{Input}, \\
&\quad \mathsf{outputs} : [\mathsf{Output}], \\
&\quad \mathsf{validityInterval} : \mathsf{Interval}[\mathsf{Slot}], \\
&\quad \mathsf{mint} : \mathsf{Value}, \\
&\quad \mathsf{mintScsRdmrs} : \mathsf{Script} \mapsto \mathsf{Redeemer}, \\
&\quad \mathsf{sigs} : \mathsf{Signature}) \\
\\
\mathsf{ValidatorContext} &= (\mathsf{Tx}, (\mathsf{Tx},\ \mathsf{Input})) \\
\mathsf{PolicyContext} &= (\mathsf{Tx}, \mathsf{PolicyID})
\end{aligned}
$$

Figure 4 Primitives and basic types for the $\mathrm{EUTxO_{ma}}$ model

already exists in the UTxO set, and the minting policy cannot be satisfied. Which, in turn, is because

$$\mathsf{myRef} \in \{\ \mathsf{outputRef}\ i \mid i \in \mathsf{inputs}\ tx\ \}$$

contradicts $\pi\ utxo = (a,\ b)$. So, thread tokens are not being minted or burned, and, by rule (v) in Section 2.3, we can make the required conclusion.

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

Naive implementation.

(i) $\pi\ tx = \star$: We have that $\neg\ (\exists\ i\ \in\ \mathsf{inputs}\ tx,\ \mathsf{value}\ (\mathsf{output}\ i) = \mathsf{ttt})$. Since an additional token ttt cannot be minted or burned, we also conclude $\neg\ (\exists\ o\ \in\ \mathsf{outputs}\ tx,\ \mathsf{value}\ o = \mathsf{ttt})$. By $\pi\ utxo = (a, b)$, the $utxo$ state contains a unique output with token ttt, datum (a, b), and $\mathsf{toggleVal}_N$ myRef validator. By $\pi\ tx = \star$, that output was not spent, and still exists in the UTxO set $utxo'$. By assumption in 5.2, since $tx \neq \mathsf{fst}\ \mathsf{myRef}$, the reference myRef is not added to the inputs of $utxo'$. So, that $\pi\ utxo' = \pi\ utxo = (a, b)$. Then,

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

$$
\begin{aligned}
\mathsf{toMap} &: \mathsf{Ix} \to [\mathsf{Output}] \to (\mathsf{Ix} \mapsto \mathsf{Output}) \\
\mathsf{toMap}\ \_\ [\,] &= [\,] \\
\mathsf{toMap}\ ix\ u\ ::\ outs &= \{\ ix \mapsto u\ \} \cup (\mathsf{toMap}\ (ix+1)\ outs)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{mkOuts} &: \mathsf{Tx} \to \mathsf{UTxO} \\
\mathsf{mkOuts}\ tx &= \{\ (tx,\ ix) \mapsto o\ \mid\ (ix \mapsto o) \in \mathsf{toMap}\ 0\ (\mathsf{outputs}\ tx)\ \}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{getORefs} &: \mathsf{Tx} \to \mathbb{P}\ \mathsf{OutputRef} \\
\mathsf{getORefs}\ tx &= \{\ \mathsf{outputRef}\ i\ \mid\ i \in \mathsf{inputs}\ tx\ \}
\end{aligned}
$$

Figure 5 Auxiliary functions for entering outputs into the UTxO set

$$
\begin{aligned}
&\mathsf{toggleTT}_N\ :\mathsf{OutputRef} \to \mathsf{Script} \to \mathsf{Script} \\
[\![\mathsf{toggleTT}_N\ \mathsf{myRef}\ s]\!]\ \_\ (tx,\ pid)\ :=& \mathsf{myRef} \in \{\ \mathsf{outputRef}\ i\ \mid\ i \in \mathsf{inputs}\ tx\ \} \\
&\wedge\ \mathsf{oneT}\ pid\ (\mathsf{encode}\ s)\ =\ \mathsf{mint}\ tx \\
&\wedge\ \exists\, o \in \mathsf{outputs}\ tx, \\
&\quad \mathsf{value}\ o\ =\ \mathsf{oneT}\ pid\ (\mathsf{encode}\ s) \\
&\quad \wedge\ \mathsf{validator}\ o\ =\ s\ \wedge\ \mathsf{fromData}_N\ (\mathsf{datum}\ o)\ \neq\ \star
\end{aligned}
$$

Figure 6 TOGGLE thread token minting policy for the naive implementation

$$
\begin{aligned}
&\mathsf{toggleTT}_D\ :\mathsf{OutputRef} \to \mathsf{Script} \to \mathsf{Script} \\
&[\![\mathsf{toggleTT}_D\ \mathsf{myRef}\ s]\!]\ \_\ (tx,\ pid)\ :=\ \mathsf{myRef} \in \{\ \mathsf{outputRef}\ i\ \mid\ i \in \mathsf{inputs}\ tx\ \}\ \wedge\ tta\ +\ ttb\ =\ \mathsf{mint}\ tx \\
&\qquad \wedge\ \exists\, oa,\ ob \in \mathsf{outputs}\ tx,\ \mathsf{value}\ oa\ =\ tta\ \wedge\ \mathsf{value}\ ob\ =\ ttb \\
&\qquad\quad \wedge\ \mathsf{validator}\ oa\ =\ \mathsf{validator}\ ob\ =\ s \\
&\qquad\quad \wedge\ \mathsf{fromData}_D\ (\mathsf{datum}\ oa)\ \neq\ \star\ \wedge\ \mathsf{fromData}_D\ (\mathsf{datum}\ ob)\ \neq\ \star \\
&\qquad \textbf{where} \\
&\qquad\quad tta := \mathsf{oneT}\ pid\ (\mathsf{encode}\ s\ +\!+\ "a") \\
&\qquad\quad ttb := \mathsf{oneT}\ pid\ (\mathsf{encode}\ s\ +\!+\ "b")
\end{aligned}
$$

Figure 7 TOGGLE thread token minting policy for the distributed implementation

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

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

Distributed implementation. The proof for the distributed implementation is similar

$$\begin{aligned}
\mathsf{ttt} &:= \mathsf{oneT}\ (\mathsf{toggleTT}_N\ \mathsf{myRef})\ (\mathsf{encode}\ (\mathsf{toggleVal}_N\ \mathsf{myRef})) \\
\mathsf{tta} &:= \mathsf{oneT}\ (\mathsf{toggleTT}_D\ \mathsf{myRef})\ (\mathsf{encode}\ (\mathsf{toggleVal}_D\ \mathsf{myRef})\ +\!+\ "a") \\
\mathsf{ttb} &:= \mathsf{oneT}\ (\mathsf{toggleTT}_D\ \mathsf{myRef})\ (\mathsf{encode}\ (\mathsf{toggleVal}_D\ \mathsf{myRef})\ +\!+\ "b")
\end{aligned}$$

$$\pi_n\ utxo\ :=\ \begin{cases}
(a,b) & \text{if myRef } \notin \{\ i \mid i \mapsto o \in utxo\ \} \\
& \wedge \exists!\ i \mapsto o\ \in\ utxo,\ \mathsf{ttt}\ =\ \mathsf{value}\ o \\
& \quad \wedge\ \mathsf{validator}\ o\ =\ \mathsf{toggleVal}_N\ \mathsf{myRef}\ \wedge\ \mathsf{datum}\ o\ =\ (a,\ b) \\
\star & \text{otherwise}
\end{cases}$$

$$\pi_{\mathsf{Tx},n}\ tx\ :=\ \begin{cases}
\mathsf{toggle} & \text{if } \exists\ i\ \in\ \mathsf{inputs}\ tx,\ \mathsf{value}\ (\mathsf{output}\ i) = \mathsf{ttt} \\
\star & \text{otherwise}
\end{cases}$$

Figure 8 TOGGLE thread tokens and naive projections

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

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 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.

4 Vitalik Buterin. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. `https://ethereum.org/en/whitepaper/`, 2014.

5 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.

6 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/`.

7 Ergo Team. Ergo: A Resilient Platform For ContractualMoney. `https://whitepaper.io/document/753/ergo-1-whitepaper`, 2019.

8 Ilya Sergey et al. Safer smart contract programming with Scilla. 3(OOPSLA):185, 2019.

9 Manuel M. T. Chakravarty et al. Native custom tokens in the extended UTXO model. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III*, volume 12478 of *LNCS*, 2020.

10 Manuel M. T. Chakravarty et al. Utxoma: Utxo with multi-asset support. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, pages 112–130, Cham, 2020. Springer International Publishing.

11 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.

12 Ethereum Team. ERC-721 TOKEN STANDARD. `https://ethereum.org/en/developers/docs/standards/tokens/erc-721`, 2023.

13 LM Goodman. Tezos—a self-amending crypto-ledger white paper. 2014.

14 Matthias Güdemann Jared Corduan 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.

15 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`.

16 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.

17 Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. VeriSolid: Correct-by-design smart contracts for Ethereum. In *International Conference on Financial Cryptography and Data Security*, pages 446–465. Springer, 2019.

18 Orestis Melkonian. A formal model of the extended utxo model in agda. `https://github.com/omelkonian/formal-utxo`, 2023.

19 Orestis Melkonian. Structured contracts: Small-step-style simulation verification of eutxo smart contracts. `https://github.com/omelkonian/structured-contracts`, 2023.

20 Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999.

21 S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. `https://bitcoin.org/en/bitcoin-paper`, October 2008.

22 George Pîrlea, 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`.

23 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`.

24 Cardano Team. Small Step Semantics for Cardano. `https://github.com/input-output-hk/cardano-ledger/releases/latest/download/small-step-semantics.pdf`, 2018.

25 Cardano Team. Cardano ledger. `https://github.com/input-output-hk/cardano-ledger`, 2023.

26 Cardano Team. Formal ledger specifications. `https://github.com/input-output-hk/formal-ledger-specifications`, 2023.

27 Runtime Verification Team. Smart contract analysis and verification, 2023. URL: `https://runtimeverification.com/smartcontract`.

28 The ZILLIQA Team. The ZILLIQA Technical Whitepaper. `https://docs.zilliqa.com/whitepaper.pdf`, 2017.

29 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.