Structured Contracts in the EUTxO Ledger Model *

Polina Vinogradova[1], Manuel Chakravarty[1], Philip Wadler[1,2], James Chapman[1], Tudor Ferariu[1,2], Michael Peyton Jones[1], Jacco Krijnen[1,3], and Orestis Melkonian[1,2]

[1] IOG, firstname.lastname@iohk.io
[2] University of Edinburgh, UK orestis.melkonian@ed.ac.uk, s1408714@sms.ed.ac.uk, wadler@inf.ed.ac.uk
[3] Utrecht University, Netherlands, j.o.g.krijnen@uu.nl

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 formalization of the very notion of correctly implementing stateful programs 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 subsystem 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 extremely versatile by presenting several highly distinct examples. In particular, we demonstrate how the framework allows for building distinct implementations of the same specification. We discuss useful potential applications of this framework to existing problems in smart contract verification.

Keywords: blockchain · ledger · smart contract · formal verification · specification · transition system · UTxO · semantics

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 [7] and Tezos [12], smart contracts are inherently stateful, and their states can be updated by transactions. Smart contracts in the EUTxO model such as [13], 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 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, are multiple existing approaches to implementing and formalizing specific designs of stateful programs running on the ledger [8] [14] [9]. There are also languages, such as BitML [5], 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 using this smart contract model for specifying and implementing stateful programs on the EUTxO ledger. In this work, we propose to re-use the existing specification standard for the ledger model to specify stateful contracts.

Like many prominent platforms [12] [7] [18] [24] [23], the EUTxO ledger is specified as a state transition system. It is given in terms of its operational small-steps semantics describing the application of blocks and

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

transactions to a given ledger state [13]. The reason for this design chice is that the evolution of the ledger takes place in atomic steps corresponding to transaction applications. We propose the *structured contract framework* (SCF) as an extension on the ledger semantics formalization. It enables the users to instantiate a small-steps program specification as a *subsystem of the ledger* via the use of smart contract scripts.

Permission scripts in the EUTxO model control the change of the part of the ledger state for which they specify permissions. Structured contracts make precise what it means to "control a part of the ledger state" by specifying the evolution of that part as stateful program, which happens in accordance with the permissions of the relevant scripts. Generalizing the subsystem design pattern proposed in [8], the SCF formalizes the notion of stateful program running on the EUTxO ledger, and what it means for it to be *implemented correctly*. Moreover, unlike the existing design pattern, our generalization allows a stateful contract to be distributed across multiple UTxOs and tokens, implemented via a suite of scripts. We note that the EUTxO model guarantees transaction commutativity [**?**], making it a natural fit for building contracts with state distributed across different parts of the ledger. These state components be updated independently with predictable outcomes, regardless of the order in which transactions are processed.

We present several examples to showcase the versatility of our approach to contract specification and verification. We discuss how the SCF can be used to formalize and address existing challenges in the field of smart contract verification in the EUTxO model. We argue that the SCF constitutes a new, principled standard approach to stateful smart contract architecture. It allows users to specify and implement behaviour of all ledger subsystems programmable via smart contract script predicates. The main contributions of this paper are :

(i) definition of a simplified EUTxO ledger via a small-steps semantics formalism ;
(ii) definition of the structured contract formalism (SCF) as a ledger subsystem, consisting of a stateful program specification together with a proof obligation ;
(iii) a case study demonstrating the use of the SCF to specify and prove an EUTxO ledger property (the preservation of value), with a related example of expressing a minting policy as a structured contract ;
(iv) a case study demonstrating the use the SCF to define and prove prove the correctness of 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 extended UTxO (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 can be expressed as a labelled transition system, which we define in this section.

The state of the system is a UTxO set, and the transition labels are transactions. As not every transaction is a valid transition for every UTxO set, we additionally formulate the constraints which a transaction must satisfy to be valid in a given UTxO set. Note that while in a realistic system, transactions are applied to the ledger in blocks, block structure and block-specific ledger state data is secondary to the discussion of smart contracts in this work.

2.1 Transition Relation Specification

To give the small-steps semantics which we use to express the transition rules and types, we follow the set-theory based notation outlined in [21], and used in [13] [**?**]. Non-standard notation used here is given in 1.

We denote a transition relation TRANS as 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

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

Given a 4-tuple $(env,\ s,\ i,\ s')$, the role of each component specifying the transition is as follows :

(i) $env \in \mathsf{Env}$ is the fixed *environment* of the state transition update, e.g. the current slot number ;
(ii) $s \in \mathsf{State}$ is the starting state to which an input is applied, e.g. the UTxO set, or a contract state ;
(iii) $i \in \mathsf{Input}$ is the user input, e.g. a transaction, or input to a contract ;
(iv) $s' \in \mathsf{State}$ is the end state obtained from the start state as the result of the application of the input, e.g. the updated UTxO state.

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

Input vs. Environment. Note that there is no formal distinction between the use of input and environment. However, making this distinction creates a useful separation between user-issued input (e.g. a transaction), and block-level data. For example, slot numbers are determined via blocks and a timekeeping mechanism at the consensus level, not specified by users. We adopt this convention from the EUTxO transition system specification [13].

2.2 Ledger Types

The ledger model we use is an extended UTxO model with native multi-assets, as introduced in [**?**]. It is based on a previously introduced model [8]. We give an overview of the EUTxO state and transaction structure below, and present the full set of type definitions in Figures 2 and 3.

Scripts. A Script, or a *smart contract*, is a piece of stateless user-defined code with a boolean output. It is used to specify conditions under which a transaction is allowed to perform a specific action. It is executed whenever a transaction attempts to perform the action for which it specifies the permissions. We do not specify the language in which scripts are written, but we presume Turing-completeness.

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 (e.g. the policy ID or the input), (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 for scripts that control minting of tokens as well as spending by $[\![\_]\!]$, followed by the script arguments.

Value. The type $\mathsf{Value} = \mathsf{FinSup}[\mathsf{PolicyID}, \mathsf{FinSup}[\mathsf{TokenName}, \mathsf{Quantity}]]$ represents bundles of multiple kinds of assets. It uses a nested finite map data structure to associate a $\mathsf{Quantity} = \mathbb{Z}$ to every $\mathsf{AssetID} = \mathsf{PolicyID} \times \mathsf{TokenName}$. That is, the bundle contains a non-zero integral quantity of assets with the asset IDs specified in the domain of the map, and zero quantity of assets with all other IDs. A token bundle containing no tokens is denoted by 0. Value forms a group for which we denote group the operation by $+$, as well as a partial order [**?**]. An AssetID uniquely identifies a class of assets, wherein all assets are fungible. It is made up of :

(i) a PolicyID, which is a script that is executed whenever a transaction mints assets under this policy to determine whether it is allowed to do so ;
(ii) a TokenName, which is some Data a user selects to differentiate between assets under the same policy. For readability, this differs from the TokenName type in [8], where it is defined to be character string.

We construct a Value with a single asset, whose asset ID is given by

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

UTxO set. The UTxO set is a finite map $\mathsf{UTxO} = \mathsf{OutputRef} \mapsto \mathsf{Output}$. The $\mathsf{OutputRef} = \mathsf{Tx} \times \mathsf{Ix}$ is the type of the key of the UTxO finite map, with $\mathsf{Ix} = \mathbb{N}$. An output reference (tx, ix) is made up of (i) a transaction tx that created the output to which it points, and (ii) index of ix, pointing to the place of a particular output in the list of outputs of that transaction. This uniquely specifies an output in a specific transaction.

An output $(a, v, d) \in \mathsf{Output}$ is made up of (i) an address a, which is a Script, and it specifies under what conditions a given output can be spent by a transaction, (ii) an asset bundle v locked by this script, and (iii) an additional piece of data $d \in \mathsf{Datum}$, specified by the transaction that adds this output to the UTxO.

Data. Is a type for encoding data that can be passed as arguments to scripts, similar in structure to a CBOR encoding. For details, see [**?**]. It is also used in [8], and in the Cardano implementation [**?**]. The types Datum (data stored alongside value in an output) and Redeemer (data specified by the user for spending a specific UTxO entry) are both synonyms for the Data type. For each script, conversion functions for both the datum and redeemer are defined to decode and encode those arguments to and from the specific argument types expected by the script. We usually call the decoding function fromData, and the encoding one toData when the context is unambiguous.

Transactions. Tx is a data structure that specifies the updates to be done 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, together with their signatures.

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

2.3 Ledger Transition Semantics

The type of the ledger transition system LEDGER, that updates the UTxO set by applying a given transaction in a specific environment (slot), is denoted as follows :

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

We introduce the function checkTx, which consists of the constraints that must be satisfied by a given tuple in order to constitute a valid ledger update. This function performs the following checks, specified in Figure 4, and consistent with the ledger rules specified in [8] [**?**] :

(i) The transaction has at least one input
(ii) The current slot is within transaction validity interval
(iii) All outputs have positive values
(iv) All output references of transaction inputs exist in the UTxO
(v) Value is preserved
(vi) No output is double-spent
(vii) All inputs validate
(viii) Minting redeemers are present
(ix) All minting scripts validate
(x) All signatures are present

We use the function checkTx to define membership in the LEDGER relation, and call this rule ApplyTx. More specifically, 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 by removing the UTxO entries in $utxo$ corresponding to the output references of the transaction inputs, and adding the outputs of the transaction to the UTxO set with correctly generated output references. The functions used to compute the updated UTxO set are defined in 3.

Unlike the ledger model in [8], which contains a full list of all transactions in the order they have been applied to the initial *empty* ledger, LEDGER does not specify an initial ledger state. For this reason, we sometimes have to make additional assumptions about transactions.

In later sections, when necessary, we assume certain conditions on the starting state of the ledger trace to be true, such as the existence of a specific entry in the UTxO set. Note also that LEDGER has no trivial transitions due to the fact that transactions without inputs are not allowed by rule 1, so the UTxO set is necessarily updated by the transaction by (at minimum) removing its inputs. An arbitrary labelled transition TRANS need not be deterministic, and may allow multiple distinct end states for a given initial state, environment, and input. However, the LEDGER system is *deterministic*, as the end state variable $utxo'$ is uniquely specified by in its only rule, ApplyTx.

3 Simulations and the structured contract formalism

The formal semantics discussed can be used to specify other stateful programs besides LEDGER. The programs we are interested in specifying for the purpose of this work are those that *run on the ledger*. Intuitively, a stateful program runs on the ledger whenever its state is observable within the ledger state, and is always updated by a transaction in accordance with the program's specification. We now formalize the notion of correctly implementing a stateful program on the ledger.

We note that the only tool for implementing programs on the ledger are smart contract scripts. A smart contract script encodes 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. For this reason, we propose a *stateful* model for specifying programs running on the ledger. Such programs are implemented via a one or more interacting scripts controlling the updates of the corresponding parts of 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.

We present several varied examples to demonstrate this. In this section, the first example is not, in fact, a script-implemeneted structured contract, but rather the specification of an existing ledger feature, the preservation of value. We then present an interpretation of an NFT as structured contract, as a way to demonstrate that minting policies constitute a way to control observable parts of the ledger state.

3.1 Simulations.

We to formalize the notion of a program running on the ledger, we first state the definition a *simulation* [17], instantiating it with lebelled state transition systems expressed as small-steps semantics specifications.

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

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

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

$$\_ \vdash \_ \xrightarrow[\mathrm{TRANS}]{\_} \_ \in \mathbb{P}\,(\mathsf{Env}_{\mathsf{TRANS}} \times \mathsf{State}_{\mathsf{TRANS}} \times \mathsf{Input}_{\mathsf{TRANS}} \times \mathsf{State}_{\mathsf{TRANS}})$$

$$\_ \vdash \_ \xrightarrow[\mathrm{STRUC}]{\_} \_ \in \mathbb{P}\,(\mathsf{Env}_{\mathsf{STRUC}} \times \mathsf{State}_{\mathsf{STRUC}} \times \mathsf{Input}_{\mathsf{STRUC}} \times \mathsf{State}_{\mathsf{STRUC}})$$

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

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

such that the following holds :

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

We write $\sim$ instead of $\sim_{\mathsf{TRANS,STRUC}}$ whenever the subscript is unambiguous. Note that 3.1 includes 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, ie. those where the permissions are controlled by user-defined scripts. For this reason, we define a particular class of simulations of LEDGER.

A *structured contract* STRUC is a transition system such that $(\mathsf{STRUC}, \sim_{\mathsf{LEDGER,STRUC}}, \sim_{\rightarrow,\mathsf{LEDGER,STRUC}}) \succeq \mathsf{LEDGER}$, where $\mathsf{Env}_{\mathsf{STRUC}} := \star$, and for some functions $\pi_{\mathsf{STRUC}} : \mathsf{UTxO} \rightarrow \mathsf{State}^{?}_{\mathsf{STRUC}}$ and $\pi_{\mathsf{Tx,STRUC}} : \mathsf{Tx} \rightarrow \mathsf{Input}_{\mathsf{STRUC}}$, the relations $\sim$ and $\sim>$ are given by :

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

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

This definition states that whenever for a given start state *utxo* with a corresponding contract start state $\pi\ utxo \neq \star$, each valid step $(slot, utxo, tx, utxo') \in \mathsf{LEDGER}$ at the ledger level necessarily has a corresponding valid step at the contract level, $(\star, \pi\ utxo, \pi_{\mathsf{Tx}}\ tx, \pi\ utxo') \in \mathsf{STRUC}$. We sometimes denote both $\pi_{\mathsf{Tx,STRUC}}$ and π_{STRUC} by π or π_{Tx} when the context is clear. We also denote the structured contracts by

$$(\mathsf{STRUC},\ \pi_{\mathsf{STRUC}},\ \pi_{\mathsf{Tx,STRUC}}) \succeq \mathsf{LEDGER}$$

There may be transactions that 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 itself be deterministic, however, since the LEDGER is deterministic, given a ledger step, there is a unique step in STRUC corresponding to the ledger one.

The purpose of structured contracts is to represent all ledger simulations that can be defined and implemented via scripts. Since scripts cannot inspect any ledger environment data (i.e. the slot number), structured contracts have a trivial environment. The reason that the π_{State} a (partial) *function* is that it must be possible to extract a *unique* contract state from the ledger state to meaningfully observe the contracts execution trace corresponding to the ledger execution. This function is *partial* because a valid $(slot, utxo, tx, utxo') \in \mathsf{LEDGER}$ may have a start state *utxo* that violates some constraints necessary to obtain a valid corresponding contract state. For example, the *utxo* state may have multiple copies of what was supposed to be an NFT (and therefore unique), and therefore does not map onto the state of a contract when that state is the datum of a UTxO entry containing that NFT.

Consider now the problem constructing a bisimulation [17], which requires that for a given step in a structured contract, there exists a corresponding valid ledger transition step. Finding tuple representing such a step requires ensuring that all constraints in checkTx are satisfied, including the validation of all dependent scripts. Defining a class of structured contracts for which this is possible is a difficult problem, and we leave it for future work.

4 Minting and preservation of value as structured contracts

In this section we give an example of using the structured contract formalism to express global invariants of the ledger that can be derived from local invariants. The EUTxO ledger transition system LEDGER is said to satisfy the *preservation of value (POV)* property. A *local* POV is the property that each valid transaction mints exactly the difference between the value in the UTxOs in spends and those it creates. This is ensured by the ledger rule 5, which is

$$\mathsf{mint}\ tx + \sum_{i \in \mathsf{txins}\ tx,\ (\mathsf{outputRef}\ i) \mapsto o \ \in utxo} \mathsf{value}\ o = \sum_{o \in \mathsf{outs}\ tx} \mathsf{value}\ o$$

Suppose that $(s,\ utxo,\ tx,\ utxo') \in \mathsf{LEDGER}$, and recall that updated UTxO set is defined as follows

$$utxo' \ := \ (\mathsf{txins}\ tx \not\lhd utxo) \cup \mathsf{mkOuts}\ tx$$

The *global* POV, which can be proved from the local POV, states that the sum total of all the tokens recorded on the ledger is changed by exactly the amount minted or burned by the transaction,

$$\sum_{or \mapsto out \in utxo'} \mathsf{value}\ out \ = \ \Big(\sum_{or \mapsto out \in utxo} \mathsf{value}\ out\Big) \ + \ (\mathsf{mint}\ tx)$$

We can express this property via the following structured contract, POV :

Transition type

$$\_ \vdash \_ \xrightarrow[\text{POV}]{\_} \_ \in \mathbb{P}\ (\star \times \mathsf{Value} \times \mathsf{Tx} \times \mathsf{Value})$$

Simulation relations

$$\pi_{\mathsf{Tx,POV}}\ tx \ := \ tx$$

$$\pi_{\mathsf{POV}}\ utxo \ := \ \sum_{\_ \mapsto out \in utxo} \mathsf{value}\ out$$

Transition rule

$$\text{UpdateValTotal}\ \frac{i := \mathsf{mint}\ tx}{\vdash (s) \xrightarrow[\text{POV}]{tx} (s \ + \ i)} \tag{3}$$

The value is changed exactly by the amount specified in the mint field, so that the $\sim>$ proof obligation corresponds exactly to the global POV formulation. To complete the definition, it remains to prove $\sim>$, which requires that, given that $(slot,\ utxo,\ tx,\ utxo') \in \mathsf{LEDGER}$, necessarily $(\star,\ \pi\ utxo,\ \pi\ tx,\ \pi\ utxo') \in \mathsf{POV}$.

So, we must show that

$$\sum_{\_ \mapsto out \in utxo'} \mathsf{value}\ out = \sum_{\_ \mapsto out \in utxo} \mathsf{value}\ out + \mathsf{mint}\ tx$$

See 5 for the proof.

4.1 NFT Minting policy specification.

Re-using the strategy of adding up tokens in existence in the UTxO, we can also define a contract expressing a *specific minting policy*. This example is meant to illustrate the flexibility and broad scope of the idea that a contract "specifies the evolution of some part of the ledger state". This example differs from the general POV contract example in a noteworthy way. The POV contract represents a property of the ledger transition system. Constructing contracts specifying the evolution of the quantity of tokens under a specific policy, on the other hand, is a tool for developing and verifying reliable minting policy code that behaves in the specified way. It can be used to formalize and verify properties of the contract.

For a specific policy, we restrict the tokens summed up in the UTxO to only those under that policy, which we will call myNFTPolicy. Here, the state type is still State := Value, and Input := Tx. Rather than define the policy, we first write a small-steps contract NFT to specify when and what changes can be made to the totality of tokens under that policy,

$$\text{UpdateNFTTotal}\ \frac{i := \{\ \mathsf{myNFTPolicy} \ \mapsto \ tkns \ \in \ \mathsf{mint}\ tx\ \} \qquad \{\} \subseteq s \subseteq s + i \subseteq \mathsf{oneT}\ \mathsf{myNFTPolicy}\ ""}{\vdash (s) \xrightarrow[\text{NFT}]{tx} (s \ + \ i)} \tag{4}$$

The specification states that a transition may only occur whenever both the start and the end state contain either no tokens, or only the NFT oneT myNFTPolicy "", and that value of the start state is necessarily contained in the value of the end state. Note that an NFT defined via this contract can never be burned, and must be the only token under its policy. It also does not require any authentication to be minted. Next, we define the projection function,

$$\pi\ utxo\ :=\ \begin{cases} 0 & \text{if hasRef}\ \ \wedge\ \ s = 0 \\ s & \text{if } s = \text{oneT myNFTPolicy ""}\ \ \wedge\ \ \neg\ \text{hasRef} \\ \star & \text{otherwise} \end{cases}$$

where

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

$$\text{hasRef}\ :=\ (\text{myNFTRef} \mapsto (\text{myNFTScript},\ \_,\ d) \in utxo)$$

Here, $\pi\ utxo$ returns a non-$\star$ result when either (i) the output myNFTRef $\mapsto$ (myNFTScript, _, d) is in the UTxO set and no NFTs under the policy myNFTPolicy have been minted yet, or (ii) the NFT under the myNFTPolicy policy is already minted with the correct token name "" and quantity 1, and there is no myNFTRef in the UTxO set output references.

With these projections, the NFT contract is implemented using two scripts, the minting policy myNFTPolicy, and the UTxO-locking script myNFTScript. We now define scripts such that it possible to prove the required simulation relation $\sim>$,

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

$$\begin{aligned} [\![\text{mkMyNFTPolicy}\ myRef]\!]\ \_\ (tx,\ pid)\ :=\ & \exists(myRef, (\text{myNFTScript},\ \_,\ \_)) \\ & \in\ \{\ (\text{outputRef}\ i, \text{output}\ i)\ \mid\ i \in \text{txins}\ tx\ \} \\ & \wedge \\ & \text{oneT}\ pid\ ""\ =\ \text{mint}\ tx \end{aligned}$$

$$[\![\text{myNFTScript}]\!]\ \_\ \_\ (tx,\ i)\ :=\ \text{oneT}\ (\text{mkMyNFTPolicy}\ (tx,\ i))\ ""\ =\ \text{mint}\ tx$$

The specific NFT

$$\text{myNFT}\ :=\ \text{oneT}\ (\text{mkMyNFTPolicy myNFTRef})\ ""$$

is identified uniquely by the output reference myNFTRef that must be spent to mint it. To make minting possible, the transaction tx must first place the following output into the UTxO, with myNFTRef := (tx, ix),

$$\text{myOut}\ :=\ \text{myNFTRef} \mapsto (\text{myNFTScript},\ \_,\ \_)$$

The reference myNFTRef is chosen by the contract author, and is only valid if it points to myOut in the UTxO. To prove $\sim>$ (see A), we need to make an additional assumption stating that a transaction which adds myOut to the UTxO cannot be valid again later, once the NFT has been minted :

NFT re-minting protection. For any $(slot,\ utxo,\ tx,\ utxo') \in$ LEDGER, with $\pi\ utxo\ =$ myNFT, necessarily $tx \neq$ fst myNFTRef.

This property is a consequence of replay protection. Replay protection is a trace-based safety property [1] of the EUTxO LEDGER, contingent on certain constraints being satisfied by initial trace states [**?**]. A full treatment of traces and properties is outside the scope of this work. So, to ensure correct program behaviour, we introduce the above assumption. A related conjecture, provenance, is also proved in [8] for a ledger structure with a fixed initial state and similar validation rules. We note that this is an assumption that must be made for all contracts sourcing the uniqueness of their NFT identifier from the uniqueness of a an output reference across the entire ledger state history.

NFT *property example.* At most one NFT under the policy myNFTPolicy can ever exist. If one does not exist, one can be minted, and moreover, an existing NFT cannot be burned : for any tx such that $i := \{\ \mathsf{myNFTPolicy} \mapsto tkns \in \mathsf{mint}\ tx\ \}$,

$$\forall\, s,\ i,\ \ \{\} \subseteq s \subseteq s + i \subseteq \mathsf{oneT}\ \mathsf{myNFTPolicy}\ ""$$

This is immediate from the NFT specification. Note that this is a safety propery [1], as it can be checked by inspecting a finite number of individual states in a given trace.

5 Multiple implementations of a single specification example

Script interaction in the UTxO ledger refers the dependency of one script on the successful validation of another that is run within the same transaction. A feature of the structured contract formalism is that it allows for specifying stateful programs that can be implemented with multiple interacting UTxO-locking and minting policy scripts, which we demonstrate in this section.

Using this example, we additionally demonstrate that our formalism allows for different implementations of the same specification. This enables contract authors to compare distinct implementations across any relevant properties, 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 $\{\mathsf{toggle}\} \uplus \star$. The two booleans in the state are interchanged by toggle, and not by $\star$. We define the transition system TOGGLE :

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

$$\text{Toggle}\ \frac{}{\vdash (x,\ y) \xrightarrow[\text{TOGGLE}]{toggle} (y,\ x)} \tag{6}$$

5.2 Toggle implementations

We present two implementations of this specification. The *naive implementation* is one that uses the datum of a single UTxO entry to store a representation of the 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. Even with this basic example, several interacting scripts must be defined to implement its specification. We present pseudocode for the required scripts.

Thread token scripts To define unique identifiers that point to the TOGGLE contract representation in the UTxO, we borrow the design pattern of *thread tokens* introduced in [8]. The idea is that an NFT is minted and stored in a UTxO entry that contains the datum representing (all or part of) the contract state. Since the NFT is script-guaranteed to be unique, it marks the unique authentic state-storing UTxO. When the state is updated, the UTxO containing the thread token is spent, and the thread token is placed in a new entry containing the updated state representation.

Naive implementation. The naive implementation state is stored within a single UTxO, and so requires one thread token to track the evolution of the subsystem state.

An output reference myRef $\in$ OutputRef is selected from those existing in the UTxO to uniquely identify the thread token. It must be spent for the minting policy to validate as a mechanism for ensuring this policy can only validate once, minting exactly one thread token, and placing it into the appropriately constructed output. The minting policy for the thread tokens is constructed as follows :

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

Distributed implementation. The minting policy for the two distributed implementation thread tokens is constructed by :

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

Here, two distinct thread tokens must be minted and placed into separate UTxOs locked with the correct script. This is because each UTxO representing part of the toggle state requires a unique identifier. We distinguish between the tokens by adding "a" to the token name of one, and "b" to the other.

Validator scripts We again require different UTxO-locking scripts for our two distinct implementations. The functions, whose type is specified in 6, encode and decode the datums which the implementation scripts inspect. Datums appears in transactions and UTxOs in the encoded Data format.

Naive implementation. The naive implementation script has two main functions : ensuring that the thread token is propagated correctly into a new UTxO, and that the datum of that UTxO contais the toggled state of the UTxO that previously contained the thread token. The type of the datum of outputs locked by $\mathsf{toggleVal}_N$ is $\mathbb{B} \times \mathbb{B}$. The validator script is given by :

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

Distributed implementation. The distributed implementation script ensures that the boolean in each of the datums of the spent UTxO entries containing the two thread tokens is the negation of the boolean of the newly created UTxOs with each of those tokens, respectively. The validator script is given by :

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

LEDGER - TOGGLE *state and transition relations* Here we first encounter a situation wherein not every UTxO has a corresponding TOGGLE state. In particular, we can only relate UTxO states to toggle states wherein the toggle contract has already been initialized, ie. thread token(s) minted and stored in a correctly-defined UTxO. We fix myRef to some specific output reference. We also instantiate the thread tokens as follows :

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

Naive implementation relations.

In the naive implementation, a ledger state is related to a TOGGLE state whenever (i) the TOGGLE state is a pair booleans where one is the negation of the other, and (ii) there is exactly one output on the ledger

containing the thread token, and locked by the correct validator, and with a datum that decodes to the same pair of booleans.

$$\begin{aligned} utxo \sim (a, b) := \mathsf{myRef} \notin \{ i \mid i \mapsto o \in utxo \} \\ \wedge \exists!\, i \mapsto o \in utxo,\ \mathsf{ttt} = \mathsf{value}\, o \\ \wedge\ \mathsf{validator}\, o = \mathsf{toggleVal}_N\ \mathsf{myRef} \wedge \mathsf{datum}\, o = (a, b) \end{aligned}$$

Next, we define the projections :

$$\pi_{\mathsf{TOGGLE}}\ utxo := \begin{cases} (a,b) & \text{if } utxo \sim (a, b) \\ \star & \text{otherwise} \end{cases}$$

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

Distributed implementation relation.

In the distributed implementation, a ledger state is related to a TOGGLE state whenever there are exactly two outputs on the ledger each containing one of the the two thread tokens, tta and ttb, locked by the correct validator, each with a datum that decodes to the the corresponding boolean in the TOGGLE state.

$$\begin{aligned} utxo \sim (a, b) := \mathsf{myRef} \notin \{ i \mid i \mapsto o \in utxo \} \\ \wedge \\ \exists!\, (i \mapsto o,\ i' \mapsto o') \in utxo,\ \mathsf{tta} = \mathsf{value}\, o \wedge \mathsf{ttb} = \mathsf{value}\, o' \\ \wedge\ \mathsf{validator}\, o = \mathsf{toggleVal}_D\ \mathsf{myRef} = \mathsf{validator}\, o' \wedge \mathsf{datum}\, o = a \wedge \mathsf{datum}\, o = b \end{aligned}$$

The function π_{TOGGLE} is defined in the same way as for the naive implementation, but with $\sim$ for the naive implementation. The transaction projection is :

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

For both implementations, we again need to make a variation on the NFT re-minting protection assumption, For any $(slot,\ utxo,\ tx,\ utxo') \in \mathsf{LEDGER}$, with $\pi\ utxo = (a, b)$, necessarily $tx \neq \mathsf{fst}\ \mathsf{myRef}$.

$\sim>$ relation proof sketch Suppose that $(slot,\ utxo,\ tx,\ utxo')$, $utxo \sim (a, b)$, and $\pi\ utxo = (a, b)$. 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 $utxo \sim \pi\ utxo'$, i.e. that $\pi\ utxo' \neq \star$.

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 :

$utxo \sim (a, b)$ implies that the token already exists in the UTxO set and is unique. The minting policy cannot be satisfied, as the constraint therein that

$$\mathsf{myRef} \in \{ \mathsf{outputRef}\, i \mid i \in \mathsf{txins}\, tx \}$$

cannot be satisfied by the definition of $\sim (a, b)$. So, since thread tokens cannot be minted or burned, and by the POV rule 5, we can make the required conclusion.

Naive implementation.

(i) $\pi\ tx = \star$:

$$\neg\ (\exists!\ i\ \in\ \mathsf{txins}\ tx,\ \mathsf{value}\ (\mathsf{output}\ i) = \mathsf{ttt})$$

since an additional token ttt cannot be minted,

$$\neg\ (\exists\ i\ \in\ \mathsf{txins}\ tx,\ \mathsf{value}\ (\mathsf{output}\ i) = \mathsf{ttt})$$

therefore, the (unique) UTxO containing ttt remains in $utxo'$, and no additional outputs containing ttt appear in $utxo'$, thus guaranteeing the uniqueness of ttt in $utxo'$. As shown above, myRef is also not added to the inputs of $utxo'$. The value, datum, and validator of the UTxO containing ttt are unchanged, so that $\pi\ utxo' = \pi\ utxo \sim (a, b)$. Then,

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

(i) $\pi\ tx = \mathsf{toggle}$:

$$\exists!\ i\ \in\ \mathsf{txins}\ tx,\ \mathsf{value}\ (\mathsf{output}\ i) = \mathsf{ttt}$$

so that the (unique) UTxO containing ttt is spent, and no ttt tokens are minted or burned. Therefore, by POV, the transaction must create a single output in $utxo'$ with that token. By definition of $\mathsf{toggleVal}_N$, that output must have a toggled datum (b, a), no other tokens, and a $\mathsf{toggleVal}_N$ script, so that $\pi\ utxo' \sim (b, a)$. Again, myRef is not added to the inputs of $utxo'$. Then,

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

Distributed implementation.

The proof for the distributed implementation is similar to the one for the naive implementation, except we reason about two inputs and two outputs containing two thread tokens.

TOGGLE *property example.* We can prove a property that the TOGGLE specification upholds, stating that either booleans are switched, or stay the same :

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

And, because we have demonstrated the $\sim>$ relation, a related property can be stated for all valid ledger transactions applied to any state $utxo$ such that $(a, b) = \pi\ utxo$:

$$\forall utxo \in \mathsf{UTxO}, (\_, utxo, tx, utxo') \in \mathsf{LEDGER}\ \ \Rightarrow\ \ \pi\ utxo = \pi\ utxo' \vee \pi\ utxo = ((\pi\ utxo')_2, (\pi\ utxo')_1)$$

Both properties require a constraint to hold for every state, and therefore are safety properties [1].

6 Related work

Code errors and design flaws have been very costly for users since the introduction of smart contracts on the Ethereum platform [2]. Among the most high-profile and costly being the DAO hack [11], and more recently, a faulty NFT contract [6]. Formal methods are being used to find, prevent, and mitigate vulnerabilities on different ledger models, and via different approaches [15].

Scilla [20] is a intermediate-level language for writing smart contracts as state machines on an account-based ledger model. The Scilla authors have used Coq to reason about contracts written in Scilla, proving a variety of temporal properties such as safety, liveness, and others. The goal of this work, however, not to study stateful program behaviour, but rather, it is to formalize the notion of correct implementations of stateful programs on a platform where programs are inherently stateless. The purpose of the properties we discuss is to exemplify how reasoning about a specification trace guarantees the conclusions to hold for any ledger representation inducing a correct its implementation. A full treatment of lifting safety and liveness properties from specification to implementation is the subject of future work. Additionally, we believe that the subsystem approach we present in this work may be used to model on-chain interactions between Scilla contracts.

The Bitcoin Modelling Language (BitML) [5] allows the definition of smart contracts running on Bitcoin by means of a restricted class of state machines. The BitML state machines are less expressive than the class of specifications considered in our model, since we assume that our stateless scripts are written in a Turing-complete language. However, the goal of this language is similar to ours - to guarantee that the behaviour of certain state machines (in the case of BitML, ones defined using this language) if in accordance with the changes made by valid transactions, i.e. soundness. The difference is that we present a framework in which one can define a state transition system with an implementation (both using a Turing-complete language), then prove soundness to achieve a "correct" implementation, whereas BitML allows users to define a sound state machine from smaller class, then compile it to a *specific* implementation. With BitML, LTL formulas can be automatically verified using a dedicated model checker. In the future, we plan to add support for LTL formulas in our framework.

VeriSolid [16] 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, however, an account-based model, rather than the EUTxO model we work with. Moreover, in contrast to the VeriSolid approach, our approach relies on the contract author to themselves to synthesize an implementation that meets the requirements specific to the contract being built, and then provides a proof obligation to show that implementation is correct. This allows for more flexibility in the implementation, as well as in the logic used in checking properties. Here, again, it may be of interest to use an approach similar to the structured contracts framework to model interactions between structured contracts.

CoSplit, presented in [19], 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 compose contracts whose state is distributed across multiple UTxOs and tokens on the ledger, and provides a way to formally guarantee that the update of the full aggregated state is in accordance with the update of its ledger representation. This application of structured contracts serves a similar purpose as for the EUTxO ledger as sharding does for an account-based one, as it can be used to increase parallelism and scalability. We note that the UTxO model is a natural fit for such state separation, since one of the benefits of such a ledger is that all operations either commute or fail [4]. Therefore, any UTxO state representation, with any (correct) implementation, will afford the relevant properties for a given distributed contract.

On the EUTxO ledger, the constraint-emitting machine design pattern [8] makes a formal correctness guarantee similar to the proof obligation we require as part of the definition of smart contracts. However, it is limited to a ledger representation of contract state that is strictly the datum and value of a single UTxO entry, expressing dependencies on other scripts via a limited set of possible constraints on transactions. Our model allows the contract state to be computed from multiple UTxO entries and tokens aggregated across the ledger state, with its evolution coordinated by multiple different scripts. Another notable difference is that the ledger model presented in [8] is a list of UTxO-stype transactions, rather than the UTxO set itself, with a unique initial state (the empty ledger). Here, we are not able to review and reason about the full transaction history, as is the case for existing realistic ledgers.

The K framework [**?**] is a unifying formal semantics framework for all programming languages, which has been used as a tool to perform audits of smart contracts [22]. QuckCheck, a property-testing library, has also been applied for the purposes of auditing existing stateful EUTxO-implemented contracts [3]. Other formal methods audits of individual contracts acknowledge the complexity and uniqueness of Turing-complete code in the EUTxO model [10]. While formal in nature, these individual audits and services do not present an overarching principle of EUTxO smart contract verification in their approaches.

7 Applications

We have presented a framework which can be used to formalize not only the contracts themselves, but also the problems in the domain of smart contract verification. We discuss several applications of this general framework to existing contract verification issues. We give examples of the problems that are most promising to be addressed with this new tool. Each of these requires an in-depth investigation, which is outside the scope of this paper :

Properties. Program verification involves analyzing and providing guaranees about the behaviour of program. In our case, these programs are structured contracts, as well as the ledger itself. Trace-based properties, such as liveness and safety properties, are the standard for analyzing program behaviour [1]. Applying and/or adjusting the definitions and theorems about properties to be used in the context of structured contracts and ledgers will give high-assurance guarantees about the behaviour of both [**?**]. Moreover, the subsystem relation in the SCF allows for establishing a correspondence between ledger and contract properties via the state projection function. This may give additional behaviour guarantees for a contract's ledger representations.

Double satisfaction. Double satisfaction is a situation in which a single action performed by a transaction on the ledger satisfies the constraints of more than one script being executed. This is a very broad, informal description, and the situation it describes may not be a problem. However, there are situations in which it would be - such as when a single payout made by a transaction to a given address satisfies the payout requirements of two distinct scripts run by the transaction.

So far, there has not been any formal treatment of exactly when double satisfaction may be undesirable, even if it is a fairly intuitive answer in most cases. The SCF may present a solution to making precise this distinction : it provides a way to separate transaction *constraint checking* from *state updates*. For example, one may include in a specification state the collection of pay-ins to be consumed by the contract and pay-outs to be collected by the intended recipient. This ensures that any correct implementation will mark such payments as made for or by a *specific contract instance*, thus mitigating problematic double satisfaction.

This approach of marking some data or assets as "for a specific contract, and from a specific contract" is a scheme that can be described as a kind of *message-passing* [**?**]. We can implement it as and instance of the SCF. We can also use this same scheme for the following challenge to tackle via the SCF:

Asynchronous or partial contract execution. Asynchronous contract execution refers to dependence between scripts implemented in a particular way. It is achieved by relaxing the requirement that dependent scripts must be executed within the same transaction, in two distinct ones. We can allowing one script to execute a step, simultaneously constructing a kind of proof artefact of its execution on a given input. In the second transaction, which consumes the artefact, the dependent script executes using the consumption of the artefact as proof of the first script's validation within a prior transaction. This second script can behave as if the first contract was executed within the same transaction with that specific input. This type of scheme may be useful, for example, as a way to transfer assets between contracts without having to run them both in a single transaction.

Another application of this scheme is allowing contracts whose code is "too large" to be run in a single transaction to be split up into what is effectively *function calls*. Since message-passing records the contract that generated a given message, as well as the input that contract was given, messages can be used as artefacts of computations of function calls [**?**].

Eliminating dependencies on opaque scripts. A script may include a constraint requiring another script to be run within the same transaction, e.g. a particular token to be minted, or some UTxO to be spent. Those scripts, may, in turn, contain additional constraints requiring yet more scripts to be run. The SCF could allow us to define ledger subsystems in which guarantees can be made about what scripts will be required to run when the subsystem executes any step. While a good implementation of a structured contract will intuitively not depend on any unnecessary script executions, we may now be able to formalize this property.

Eliminating reliance on the validation of opaque scripts to advance a contract state on the ledger is an important goal, especially in the context of proving liveness properties such as liquidity. Achieving it is a step towards being able to guarantee the existence of a valid ledger transaction corresponding to each step in the specification of a contract state transition.

8 Conclusion

We have presented a novel approach to specifying and reasoning about behaviour of stateful programs running on a EUTxO ledger, which we call the structured contract formalism. Our formalism defines a robust way to relate stateless predicate scripts executed at the ledger level to the corresponding corresponding executions of a specific stateful program. We used the well-established concept of a subsystem to define this

relation, and a small-step semantics style already in use in existing systems (i.e. Cardano) for the specification of the ledger and contracts. This work presents a broadly applicable and principled way of reasoning about stateful programs on the EUTxO ledger.

This paper lays the groundwork for treating ledger-implemented programs as formal subsystems of the ledger. This is done on a level that is very specific to the details of the EUTxO ledger. In the future, we aim to generalize our research to be applicable to other kinds of ledger transition systems. We would also like to better align our findings with existing concepts in the theory of simulation, concurrency, and distributed computation, in order to apply the full gamut of results in those areas for studying stateful programs on the ledger.

Another goal for future work is to find a way to use the findings presented here to verify existing contracts. This may be done by first constructing the ledger and transaction representation projection functions, then building a state transition specification induced by LEDGER for those projections. Properties of the resulting system can then be studied.

A full mechanization in Agda of the results and definitions in this work is currently under way, as it is a natural next step. We also intend to build a mechanization of this work integrated with the Agda mechanization of a more sophisticated and realistic ledger, the Cardano ledger. The Cardano ledger is in a unique position to be amenable to the structured contract framework approach to verification due to the existence of a mechanized small-steps specification in Agda of the entire ledger (currently in development) [**?**]. Agda automation of proof generation for the subsystem proof obligations, such as using an SMT-solver, will be a natural progression of the project.

A Appendix

Notation :

$\star : \{\star\}$	the one-element set, and its one inhabitant
$\mathsf{fst} : (A \times B) \to A$	first projection
$Key \mapsto Value \subsetneq \{\, k \mapsto v \mid k \in Key,\, v \in Value \,\}$	finite map with unique keys
$[\, \mathsf{f}\ b \mid b \leftarrow myList \,] : [C]$	list comprehension, given $\mathsf{f} : B \to C$
$\mathsf{map} : (A \to B) \to [A] \to [B]$	apply map to every element in given list
$\mathsf{map} : (A \to B) \to \mathbb{P}\ A \to \mathbb{P}\ B$	apply map to every element in given set
$MyType \cup \star$	maybe type

Fig. 1: Notation

$\sim>$ *proof sketch for* NFT. Suppose $(slot,\ utxo,\ tx,\ utxo') \in$ LEDGER. $\pi\ utxo = 0$ **:** By hasRef, myOut $\in utxo$. If $i = 0$, the NFT is not minted, and $0 = s = s'$, and indeed $(\star, 0, tx, 0) \in$
$funNFT$. If $i \neq 0$, necessa

myOut is spent, and myOut can only be spent if myNFT is minted. Note here that in fulfilling the proof obligation of the NFT structured contract, we require to show that the output reference myNFTRef can never be added to the UTxO by tx, making it possible to mint a second copy of myNFT.

References

1. Alpern, B., Schneider, F.B.: Defining liveness. Information Processing Letters **21**(4), 181–185 (1985). https://doi.org/https://doi.org/10.1016/0020-0190(85)90056-0, https://www.sciencedirect.com/science/article/pii/0020019085900560

2. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts sok. p. 164–186. Springer-Verlag, Berlin, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_8, https://doi.org/10.1007/978-3-662-54455-6_8
3. Bailly, A.: Model-based testing with quickcheck (2022), https://engineering.iog.io/2022-09-28-introduce-q-d/
4. Bartoletti, M., Galletta, L., Murgia, M.: A theory of transaction parallelism in blockchains. Logical Methods in Computer Science **Volume 17, Issue 4** (nov 2021). https://doi.org/10.46298/lmcs-17(4:10)2021, https://doi.org/10.46298%2Flmcs-17%284%3A10%292021
5. Bartoletti, M., Zunino, R.: BitML: a calculus for Bitcoin smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 83–100. ACM (2018)
6. Bhardwaj, S.: Popular nft launch on ethereum loses 34 million in faulty smart contract (2022), https://www.forbesindia.com/article/crypto-made-easy/popular-nft-launch-on-ethereum-loses-34-million-in-faulty-smart-contract/75675/1
7. Buterin, V.: Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/en/whitepaper/ (2014)
8. Chakravarty, M.M.T., Chapman, J., MacKenzie, K., Melkonian, O., Müller, J., Jones, M.P., Vinogradova, P., Wadler, P.: 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. LNCS, vol. 12478 (2020)
9. Chakravarty, M.M.T., Coretti, S., Fitzi, M., Gazi, P., Kant, P., Kiayias, A., Russell, A.: Hydra: Fast isomorphic state channels. IACR Cryptol. ePrint Arch. **2020**, 299 (2020)
10. Chevrou, F.: A journey through the auditing process of a smart contract (2023), https://www.tweag.io/blog/2023-05-11-audit-smart-contract/
11. Falkon, S.: The story of the DAO – its history and consequences. https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee (2017), medium.com
12. Goodman, L.: Tezos—a self-amending crypto-ledger white paper (2014)
13. Knispel, A., Vinogradova, P.: A Formal Specification of the Cardano Ledger integrating Plutus CoreFormal Spec: Cardano Ledger with Plutus Core. https://github.com/input-output-hk/cardano-ledger/releases/latest/download/alonzo-ledger.pdf (2021)
14. Lamela Seijas, P., Nemish, A., Smith, D., Thompson, S.: Marlowe: Implementing and analysing financial contracts on blockchain. In: Bernhard, M., Bracciali, A., Camp, L.J., Matsuo, S., Maurushat, A., Rønne, P.B., Sala, M. (eds.) Financial Cryptography and Data Security. pp. 496–511. Springer International Publishing, Cham (2020)
15. Margaria, T., Steffen, B. (eds.): Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV, Lecture Notes in Computer Science, vol. 11247. Springer (2018). https://doi.org/10.1007/978-3-030-03427-6, https://doi.org/10.1007/978-3-030-03427-6
16. Mavridou, A., Laszka, A., Stachtiari, E., Dubey, A.: VeriSolid: Correct-by-design smart contracts for Ethereum. In: International Conference on Financial Cryptography and Data Security. pp. 446–465. Springer (2019)
17. Milner, R.: Communicating and mobile systems: the π-calculus. Cambridge University Press (1999)
18. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/en/bitcoin-paper (October 2008)
19. Pîrlea, G., Kumar, A., Sergey, I.: Practical smart contract sharding with ownership and commutativity analysis. p. 1327–1341. PLDI 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3453483.3454112, https://doi.org/10.1145/3453483.3454112
20. Sergey, I., Nagaraj, V., Johannsen, J., Kumar, A., Trunov, A., Hao, K.C.G.: Safer smart contract programming with Scilla. Proceedings of the ACM on Programming Languages **3**(OOPSLA), 185 (2019)
21. Team, C.: Small Step Semantics for Cardano. https://github.com/input-output-hk/cardano-ledger/releases/latest/download/small-step-semantics.pdf (2018)
22. Team, R.V.: Smart contract analysis and verification (2023), https://runtimeverification.com/smartcontract
23. Team, T.Z.: The ZILLIQA Technical Whitepaper. https://docs.zilliqa.com/whitepaper.pdf (2017)
24. Xie, J.: Nervos CKB: A Common Knowledge Base for Crypto-Economy. https://github.com/nervosnetwork/rfcs/blob/master/rfcs/0002-ckb/0002-ckb.md (2018)

BASIC TYPES

$\mathbb{B}, \mathbb{N}, \mathbb{Z}$	the type of Booleans, natural numbers, and integers
$\mathbb{H}$	the type of bytestrings: $\bigcup_{n=0}^{\infty}\{0,1\}^{8n}$
$\mathbb{P}\ T$	the type of (finite) sets over T
$[\mathrm{T}]$	the type of lists over T, with $\_[\_]$ as indexing and $\lvert\_\rvert$ as length
$h :: t$	the list with head h and tail t
$\mathsf{Interval}[A]$	the type of intervals over a totally-ordered set A
$\mathsf{FinSup}[K, M]$	the type of finitely supported functions from a type K to a monoid M

LEDGER PRIMITIVES

$\mathsf{Quantity} = \mathbb{Z}$	an amount of an assets
$\mathsf{TokenName} = [\mathsf{Char}]$	token name string
$\mathsf{AssetID} = \mathsf{PolicyID} \times \mathsf{TokenName}$	unique asset identifier
$\mathsf{Coin} \in \mathsf{AssetID}$	asset ID of the primary currency
Slot	slot number representing chain time
Data	a type of structured data
Script	the (opaque) type of scripts
$[\![\_]\!] : \mathsf{Script} \to \mathsf{Datum} \times \mathsf{Redeemer} \times \mathsf{ValidatorContext} \to \mathbb{B}$	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

$$
\begin{aligned}
\mathsf{Ix} &= \mathbb{N}\\
\mathsf{PolicyID} &= \mathsf{Script}\\
\mathsf{Redeemer} &= \mathsf{Data}\\
\mathsf{Datum} &= \mathsf{Data}\\
\mathsf{Signature} &= \mathsf{pubkey} \mapsto \mathbb{H}\\
\\
\mathsf{Value} &= \mathsf{FinSup}[\mathsf{PolicyID}, \mathsf{FinSup}[\mathsf{TokenName}, \mathsf{Quantity}]]\\
\\
\mathsf{OutputRef} &= (\mathsf{id} : \mathsf{Tx}, \mathsf{index} : \mathsf{Ix})\\
\\
\mathsf{Output} &= (\mathsf{validator} : \mathsf{Script},\\
&\qquad \mathsf{value} : \mathsf{Value},\\
&\qquad \mathsf{datum} : \mathsf{Data})\\
\\
\mathsf{Input} &= (\mathsf{outputRef} : \mathsf{OutputRef},\\
&\qquad \mathsf{output} : \mathsf{Output},\\
&\qquad \mathsf{redeemer} : \mathsf{Redeemer})\\
\\
\mathsf{Tx} &= (\mathsf{txins} : \mathbb{P}\ \mathsf{Input},\\
&\qquad \mathsf{outs} : [\mathsf{Output}],\\
&\qquad \mathsf{validityInterval} : \mathsf{Interval}[\mathsf{Slot}],\\
&\qquad \mathsf{mint} : \mathsf{Value},\\
&\qquad \mathsf{mintScsRdmrs} : \mathsf{Script} \mapsto \mathsf{Redeemer},\\
&\qquad \mathsf{sigs} : \mathsf{Signature})\\
\\
\mathsf{UTxO} &= \mathsf{OutputRef} \mapsto \mathsf{Output}
\end{aligned}
$$

Fig. 2: Primitives and basic types for the $\mathrm{EUTxO}_{\mathrm{ma}}$ model

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

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

$$
\begin{aligned}
\mathsf{getORefs} &: \mathsf{Tx} \to \mathbb{P}\ \mathsf{OutputRef} \\
\mathsf{getORefs}\ tx &= \{\ \mathsf{outputRef}\ i\ \mid\ i\ \in\ \mathsf{txins}\ tx\ \} \\
\mathsf{getORefs}_U\ tx &= \{\ \mathsf{outputRef}\ i\ \mid\ i\ \in\ \mathsf{txins}\ tx,\ U\ (\mathsf{outputRef}\ i\ \mapsto\ \mathsf{output}\ i)\ \}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{ValidatorContext} &= (\mathsf{Tx}, (\mathsf{Tx},\ \mathsf{Input})) \\
\mathsf{PolicyContext} &= (\mathsf{Tx}, \mathsf{PolicyID})
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{oneT} &: \mathsf{PolicyID} \to \mathsf{TokenName} \to \mathsf{Value} \\
\mathsf{oneT}\ p\ n &:= \{\ p \mapsto \{\ n\ \mapsto 1\ \}\ \}
\end{aligned}
$$

Fig. 3: Auxiliary functions for entering outputs into the UTxO set

1. **Transaction has at least one input**
$$\mathsf{txins}\ tx\ \neq\ \{\}$$
2. **The current slot is within the validity interval**
$$slot \in \mathsf{validityInterval}\ tx$$
3. **All outputs have positive values**
$$\forall o \in \mathsf{outs}\ tx,\ \mathsf{value}\ o > 0$$
4. **All inputs refer to unspent outputs**
$$\forall (oRef,\ o)\ \in\ \{(\mathsf{outputRef}\ i,\ \mathsf{output}\ i)\mid i \in \mathsf{txins}\ tx\},\ oRef\ \mapsto\ o\ \in\ utxo$$
5. **Value is preserved**
$$\mathsf{mint}\ tx + \sum_{i \in \mathsf{txins}\ tx,\ (\mathsf{outputRef}\ i) \mapsto\ o\ \in\ utxo} \mathsf{value}\ o = \sum_{o \in \mathsf{outs}\ tx} \mathsf{value}\ o$$
6. **No output is double spent**
$$\text{If } i_1, i \in \mathsf{txins}\ tx \text{ and } \mathsf{fst}\ (\mathsf{outputRef}\ i) = \mathsf{outputRef}\ i \text{ then } \mathsf{fst}\ i\ = i.$$
7. **All inputs validate**
$$\text{For all } i \in \mathsf{txins}\ tx,\ [\![\mathsf{validator}\ i]\!](\mathsf{datum}\ i,\ \mathsf{redeemer}\ i,\ (tx, i)) = \mathsf{True}$$
8. **Minting redeemers present**
$$\forall\ pid\ \in \mathsf{supp}(\mathsf{mint}\ tx),\ \exists(pid, \_)\ \in\ \mathsf{mintScsRdmrs}\ tx$$
9. **All minting policy scripts validate**
$$\text{For all } (s, rdmr) \in \mathsf{mintScsRdmrs}\ tx,\ [\![s]\!](rdmr, (tx, s)) = \mathsf{True}$$
10. **All signatures are correct**
$$\text{For all } (pk \mapsto s) \in \mathsf{sigs}\ tx,\ \mathsf{checkSig}(tx, pk, s) = \mathsf{True}$$

Fig. 4: Validity of a transaction t in the $\mathrm{EUTxO_{ma}}$model

$$
\begin{aligned}
\sum_{or \mapsto out \in utxo'} \mathsf{value}\ out &= \sum_{or \mapsto out \in ((\mathsf{getORefs}\ tx) \lhd utxo) \cup \mathsf{mkOuts}\ tx} \mathsf{value}\ out \\
&= \sum_{or \mapsto out \in utxo} \mathsf{value}\ out\ - \sum_{i \in \mathsf{txins}\ tx,\ (\mathsf{outputRef}\ i \mapsto out) \in utxo} \mathsf{value}\ out\ + \sum_{out \in \mathsf{outs}\ tx} \mathsf{value}\ out \\
&= \sum_{or \mapsto out \in utxo} \mathsf{value}\ out\ - \sum_{i \in \mathsf{txins}\ tx,\ (\mathsf{outputRef}\ i \mapsto out) \in utxo} \mathsf{value}\ out\ +\ (\mathsf{mint}\ tx) \\
&\quad + \sum_{i \in \mathsf{txins}\ tx,\ (\mathsf{outputRef}\ i \mapsto out) \in utxo} \mathsf{value}\ out \\
&= \sum_{or \mapsto out \in utxo} \mathsf{value}\ out\ +\ (\mathsf{mint}\ tx)
\end{aligned}
$$

Fig. 5: Proof of the $\sim>$ constraint for POV

$$
\begin{aligned}
\mathsf{toData}_N &: (\mathbb{B} \times \mathbb{B}) \to \mathsf{Data} \\
\mathsf{toData}_D &: \mathbb{B} \to \mathsf{Data} \\
\mathsf{fromData}_N &: \mathsf{Data} \to (\mathbb{B} \times \mathbb{B}) \\
\mathsf{fromData}_D &: \mathsf{Data} \to \mathbb{B}
\end{aligned}
$$

Fig. 6: Encoding and decoding TOGGLE script datums