# Message-passing in the Extended UTxO Ledger Model

Polina Vinogradova[1] and Orestis Melkonian[1]

Input Output, Global (IOG), firstname.lastname@iohk.io

**Abstract.** A notable problem faced by developers of smart contracts running on an extended UTxO (EUTxO) ledger is *double satisfaction*: interacting contracts that make payouts may validate with insufficient payments made to some recipients.
In this work, we formalize the notion of a stateful contract constraint being vulnerable to double satisfaction. Next, we formalize interaction among scripts and stateful contracts via *message-passing*, consisting of a specification and an implementation of a stateful distributed message-passing contract, together with a proof of the integrity of its implementation. Messages specify sender and receiver outputs, as well as the data and assets being communicated, which are recorded on the ledger in the form of special NFT tokens distributed across UTxO entries that also contain the sent assets.
We give two applications of our design by considering a message: (1) as a record of a successful script computation, akin to memoization, and (2) as a mechanism for asynchronous structured contracts communication that enables a principled separation of contract communication from its computation.
Building on this application of message-passing, we present a result stating that making payouts from stateful contracts using message-passing is not vulnerable to double satisfaction.

**Keywords:** Blockchain · Ledger · UTxO · EUTxO · Smart contract · Formal verification · Small-step operational semantics · Message-passing · Double satisfaction · Simulation relation

## 1 Introduction

Message-passing is the standard for communicating data and assets between contracts in account-based ledgers [4,26,15]. In some cases, the transaction itself may be referred to as a message [15]. An alternative to the account-based ledger model is the EUTxO (extended UTxO) ledger model, implemented by platforms such as Cardano [8,17] and Ergo [11]. It is a smart contract- (or *script*-) enabled UTxO-based ledger model, where user-defined script code is used to specify conditions a transaction must satisfy to be permitted to spend a UTxO entry or mint a token. Communication between scripts in EUTxO-based ledgers follows a different architecture than for account-based models. A script may require that another script executes successfully within the same transaction. Script interaction and communication is implemented using these kinds of *script dependencies*, as well as other constraints on the script-executing transaction.

Relying on unstructured, ad-hoc communication among EUTxO scripts presents some challenges, in particular, in terms of amenability to formal verification. For example, a sequence of dependencies may be formed wherein one script depends on another script, which, in turn, depends on yet another script, etc. Such dependency sequences may be of arbitrary length, and difficult to reason about. Another challenge is tracking the flow of assets and data, including marking certain quantities of assets or data as "from" a particular contract (e.g. a payout to a specific address), or "to" a particular contract (e.g. a pay-in from an address). Asset flow tracking is a special case of the more general *double satisfaction problem* (DSP).

The DSP may occur when multiple scripts within the same transaction share a constraint satisfied by the transaction. Problematic occurrences of DS are due to the lack of a mechanism to associate the fulfillment of a constraint with the script imposing said constraint. For example, a payout to an specific address may be required by two distinct contracts, which place the same constraint on a transaction. The intent of *each* of the contracts was that the payout address should have received a separate payment, for a total of two. However, the payout is made only once, satisfying both contracts. Because many contracts make payouts, this is a widely discussed problem in EUTxO programming. In Section 3, we present a formalization of the DSP, which has not previously been formalized.

Previous work presents principled approaches to building stateful contracts in the EUTxO ledger model, such as the constraint-emitting machine design pattern [7], as well as the more general structured contract framework (SCF) [27]. These have been mechanized in the Agda proof assistant [21] and provide the

conceptual basis for the actual specification of the Cardano ledger specification [8,17], which is formulated with small-step semantics and is also mechanized in Agda [16].

We present an EUTxO layer-2 implementation of *asynchronous message-passing* as a principled approach to communication among individual scripts and the stateful contracts they implement. Our message-passing architecture is a stateful contract constructed as an instance of the structured contract framework.

The state of the message-passing contract is a set of messages. On the ledger, a special NFT token encodes a single message in the contract state, and individual message tokens are distributed across distinct UTxO entries. Each message specifies *sender* and *recipient* UTxO entries, which are authenticated at the time of minting (burning, resp.) of the message token. The message token specifies the data being sent, and must be placed in a UTxO also containing the assets being sent. Any script is able to interface with the message-passing contract so long as (i) the user input to the script can be decoded as a list of messages being produced and consumed, and (ii) the corresponding message NFTs are minted (burned) by the transaction.

Our decentralized stateful contract design constitutes a way to interpret the notion of message-passing communication on an EUTxO ledger. The scripts implementing this design facilitate concurrent updates to asset token balances on the ledger that exist independently of a shared database. We argue that the message-passing approach to script and stateful contract communication addresses some of the challenges of writing scripts to run on an EUTxO ledger. To that end, we present two use cases of message-passing together with formal specification and verification of properties related to the integrity of their behavior. We demonstrate how expressing payouts as messages from a stateful contract can resolve the DSP for payouts. The main contributions of this work are :

(i) formalization of the double satisfaction problem (Section 3);
(ii) specification and implementation of a message-passing structured contract (Section 4);
(iii) an application of the message-passing contract for memoization, including proven properties of its behavior (Section 5.1);
(iv) an application of the message-passing contract as a means of asynchronous communication of data and assets between structured contracts, together with formal properties of its behavior, including resilience of payout messages to the DSP (Section 5.2).

## 2 Background

### 2.1 The EUTxO ledger model

First, we give an overview of the semantics we use for our contract and ledger specifications, introduced in prior work [7,5,27], but included here for the sake of self-containment.

**Ledger types.** For the purposes of self-containment, we include a description of EUTxO ledger model types. We note and justify the (minimal) changes we introduce to the existing model in the description. Additional details are in Appendix B. The types of booleans, natural numbers, and integers are denoted by $\mathbb{B}$, $\mathbb{N}$, and $\mathbb{Z}$, respectively. The type $\mathsf{Ix} := \mathbb{N}$ is used for indexing, e.g. of elements in a list. The type $\mathsf{Slot} := \mathbb{N}$ is used to indicate blockchain time.

The ledger state consists of a UTxO set, which is a collection of unspent outputs, each associated with a unique identifier. An output is a triple $(a, v, d) \in \mathsf{Output}$, where $a \in \mathsf{Script}$ is the address of the output, $v \in \mathsf{Value}$ is the collection of assets at this address, and $d \in \mathsf{Datum}$ is data specified by the user at the time of constructing the transaction which creates this output (we give more details on these three types below). The type of the UTxO set is $\mathsf{UTxO} := \mathsf{OutputRef} \mapsto \mathsf{Output}$, which is a finite key-value map with keys of type $\mathsf{OutputRef}$. We denote a single element in a finite key-value map $u$ (such as the UTxO set) by $i \mapsto o \in u$. An output reference $(tx, ix) \in \mathsf{OutputRef} := \mathsf{Tx} \times \mathsf{Ix}$ pointing to an output $o$ in a UTxO set consists of the transaction $tx$, which added $(tx, ix) \mapsto o$ to the UTxO set, and the index $ix$, which is the position of output $o$ in the list of outputs of $tx$.

The type $\mathsf{Value}$ is used to represent bundles of multiple kinds of assets (see Fig. 8). Each type of asset in the bundle $v \in \mathsf{Value}$ has a unique asset ID, $a \in \mathsf{AssetID} := \mathsf{Policy} \times \mathsf{TokenName}$, which identifies a class of fungible tokens. Associated to the asset ID of each type of asset in a bundle is a a quantity $q \in \mathsf{Quantity} := \mathbb{Z}$, specifying the amount of the asset with the given ID in $v$. When $v$ contains 0 of a given asset type, its asset ID is not included in $v$. An asset bundle containing one kind of asset with asset ID $(\mathsf{policy}, \mathsf{tokenName})$ of quantity one is denoted by $\{\ \mathsf{policy} \mapsto \{\mathsf{tokenName} \mapsto 1\}\ \}$.

An asset with ID $(p, t)$ has the minting (and burning) policy $p \in \mathsf{Policy} := \mathsf{Script}$. When an asset under this policy is minted or burned, the policy script is executed to determine whether the transaction is allowed to perform this action. The token name $t$ is specified by the user at the time of constructing the minting transaction. It is used to differentiate between assets under the same policy. Unlike previous work [5,27], where the token name is a string, we take $\mathsf{TokenName} := \mathsf{Data}$. $\mathsf{Value}$ forms a group under addition $(+)$ with a zero element $(0)$ and a partial order $(\leq)$ [6].

A script $s \in \mathsf{Script}$ is a piece of user-defined code that is executed as part of transaction validation, applied to specific inputs. Script code is stateless and produces a boolean output. Scripts are executed as part transaction validation to check that a transaction is permitted to do the action with which the script is associated. Scripts are used to specify permissions for two kinds of actions: spending an output (these are referred to as "validators", or sometimes the "address" of the output), and minting or burning tokens (these are called minting policies).

We denote script application by $[\![\_]\!]$, followed by the script arguments (see Fig. 8). At the time of evaluation, the arguments supplied to a script consist of transaction data (of the transaction executing it), as well as the data about the specific action for which the script specifies permission (i.e. the output being spent, or the tokens being minted under the policy). An extra piece of data $d \in \mathsf{Redeemer}$, associated with the particular action being validated, is specified by the user at the time transaction construction.

On-chain data of variable type, including $\mathsf{Datum}$, $\mathsf{Redeemer}$, and $\mathsf{TokenName}$, are all type synonyms for $\mathsf{Data}$ (see Fig. 8); for the sake of brevity, we will omit explicit calls to the corresponding encoding/decoding functions as these will be obvious from the types involved, so any time a value is used as $\mathsf{Data}$ presupposes that decoding is successful.

Updates to the ledger state are specified in the form of a $\mathsf{Tx}$ (transaction) data structure. A transaction $tx \in \mathsf{Tx}$ contains (i) a set of *inputs*, each containing an output reference, an output, and the associated redeemer, (ii) a list 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 $\mathsf{Value}$ 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, alongside their signatures.

**Small-step specifications.** We formulate the transitions of ledgers and contracts in the form of small-step operational semantics [22], as exemplified by the official specification of the Cardano ledger [8,17].

In our specifications and contract implementation pseudocode, we follow standard set-theory notation, and clarify any non-standard notation usage in the Appendix (see Fig. 7).

A transition relation $\mathsf{TRANS} \subseteq (\mathsf{Env} \times \mathsf{State} \times \mathsf{Input} \times \mathsf{State})$ is a collection of 4-tuples. A member $(env,\ s,\ i,\ s')$ of this relation is also denoted by :

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

The variable $env \in \mathsf{Env}$ is the environment of the state transition, $s \in \mathsf{State}$ is the starting state, $i \in \mathsf{Input}$ is the input, and $s' \in \mathsf{State}$ is the end state. The system $\mathsf{TRANS}$ is a labelled transition system. For a given transition $(env,\ s,\ i,\ s') \in \mathsf{TRANS}$, the pair $(env,\ i)$ of an environment and an input make up the *label* of this transition from $s$ to $s'$. Conventionally [8], *env* is block-level data, such as blockchain time, whereas $i$ is specified by the user, e.g. a transaction.

**Ledger transition semantics.** The ledger semantics on top of which we build the results of this paper are found in existing work [7,5,27], but we include them here for self-containment and in order to introduce appropriate notation. The ledger transition system is given by the subset $\mathsf{LEDGER} \subseteq \mathsf{Slot} \times \mathsf{UTxO} \times \mathsf{Tx} \times \mathsf{UTxO}$. Membership in this subset is specified by a single transition rule $\mathsf{ApplyTx}$, which ensures that $(slot,\ utxo,\ tx,\ utxo') \in \mathsf{LEDGER}$ whenever $\mathsf{checkTx}\ (slot,\ utxo,\ tx) = \mathsf{True}$, and $utxo'$ is given by $(\{\ i \mapsto o \in utxo \mid i \notin tx.\mathsf{outputRefs}\ \}) \cup tx.\mathsf{outputs}$. This is expressed in rule ApplyTx below, where any unbound variables are implicitly considered as universally quantified.

$$utxo' := (\{\ i \mapsto o \in utxo \mid i \notin tx.\mathsf{outputRefs}\ \}) \cup tx.\mathsf{outputs}$$

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

The function $\mathsf{checkTx} : \mathsf{Slot} \times \mathsf{UTxO} \times \mathsf{Tx} \to \mathbb{B}$ checks the predicates specified in Fig. 9, which are consistent with the EUTxO model on which this work builds [5]. This includes executing all required validator and minting policy scripts with the appropriate inputs. The projection $tx.\mathsf{outputRefs}$ returns a UTxO set containing an entry $k \mapsto o$ for each input $i$ of $tx$, where the key of the entry is the output reference $k$ of $i$, and its value is the output $o$ of $i$. The value $utxo'$ is calculated by removing the UTxO entries in $utxo$ corresponding to those in $tx.\mathsf{outputRefs}$, and adding the entries constructed by $tx$.

### 2.2 Structured contracts

The structured contract framework [27] is a formalism for specifying and demonstrating the integrity of the implementation of a stateful contract on LEDGER. We give the definition here for self-containment and in order to introduce the appropriate notation. A structured contract includes a small-steps semantics specification, as well as a ledger representation of its state and input. The ledger representation is a pair of functions: one which computes the contract state from a given UTxO state (or fails), and another which computes the input to the contract for a given transaction.

For a given valid LEDGER step, the representation functions must compute a valid step in the structured contract specification given that the starting UTxO state corresponds to a contract state. This integrity constraint is expressed as a proof obligation for the instantiation of a structured contract. This design guarantees that no invalid contract state updates are ever possible on the ledger.

Suppose $\mathsf{STRUC} \subseteq (\{\star\} \times \mathsf{State} \times \mathsf{Input} \times \mathsf{State})$ is a small-step transition system. Let $\pi : \mathsf{UTxO} \to \mathsf{State} \cup \{\star\}$ and $\pi_{\mathsf{Tx}} : \mathsf{Tx} \to \mathsf{Input}$ be functions such that :

$$\frac{\pi\, u \;\neq\; \star \qquad e \vdash (u) \xrightarrow[\text{LEDGER}]{t} (u')}{(\pi\, u' \;\neq\; \star) \;\wedge\; \star \vdash (\pi\, u) \xrightarrow[\text{STRUC}]{\pi_{\mathsf{Tx}}\, t} (\boldsymbol{\pi\, u'})}$$

The triple $(\mathsf{STRUC}, \pi, \pi_{\mathsf{Tx}})$ is called a *structured contract*, and we denote it by $(\mathsf{STRUC}, \pi, \pi_{\mathsf{Tx}}) \succeq \mathsf{LEDGER}$. Note that $\pi$ is function with output type $\mathsf{State} \cup \{\star\}$, where $\{\star\}$ is a singleton. When $\pi\, u = \star$, there is no contract state corresponding to the ledger state $u$. The block-level data is never exposed to user-defined scripts in this model, so that the context of a structured contract is necessarily $\star \in \{\star\}$.

## 3 The problem of double satisfaction

In the EUTxO model, multiple scripts can be executed as part of validation of the transaction which causes them to be run, i.e. the *carrying* transaction. Each script is run when the action it is associated with is performed by the carrying transaction, such as spending a particular UTxO or minting tokens under a specific policy. Scripts contain constraints on the data of the carrying transaction, and *multiple scripts* may place the *same constraint* on the data of a given transaction. The issue with certain undesirable instances of this situation is called the *double satisfaction problem* (DSP). The DSP has been discussed in Plutus documentation,[1] and in the context of contract audits,[2][3] but has not yet been formally analyzed. It applies to scripts and structured contracts that require transactions to make payouts, so it is frequently encountered in EUTxO script programming.

Consider the following examples of constraints a structured contract with $\mathsf{Input} := \mathsf{Tx}$ may place on its input transaction $tx$ :

(i) **Authorization tokens :** the transaction must contain in its inputs a special token, the presence of which constitutes proof that a particular contract state update is authorized
(ii) **Payouts :** the transaction must make a payout to address $a : \mathsf{Script}$ by including an output containing value $v$, with address $a$

[1] https://plutus.readthedocs.io/en/latest/reference/writing-scripts/common-weaknesses/double-satisfaction.html
[2] https://medium.com/@vacuumlabs_auditing/cardano-vulnerabilities-1-double-satisfaction-219f1bc9665e
[3] https://github.com/tweag/tweag-audit-reports/blob/main/Marlowe-2023-03.pdf

The ability of a transaction author to present an authorization token, as in (i), is treated as proof that some action the transaction performs is authorized. It is possible that a single transaction makes contract state updates to multiple contracts whose implementations require that a specific authorization token is presented. There is no problem with a transaction updating the state of multiple distinct contracts, and presenting a single authorization token to all contracts it executes which require it. This is not an example of problematic double satisfaction, since the intended meaning of this constraint is upheld by its implementation.

When two structured contract implementations both place the constraint (ii) on a transaction, it may be satisfied by a single transaction output $(a, v, \_)$. Note here that we use the notation _ to represent a term whose value is not relevant to the computation in which it appears. One may speculate that the authors of each of the scripts with constraint (ii) intended for the output $(a, v, \_)$ to be somehow associated with the step of the particular contract they care about. This means that two distinct outputs would be required, each associated with a specific contract. In this case, double satisfaction is problematic, and $a$ receives less total assets than intended.

The examples we presented show that the distinction between examples of DS that are problematic and those that are not is strictly in the *intent of the script author*. For this reason, we can only judge whether a constraint is *vulnerable* to double satisfaction, i.e. it is not associated exclusively with a particular structured contract. Let us assume that all systems discussed in this section are deterministic, and define the following function, which returns all pairs of states in all valid transitions of a given structured contract $\mathsf{STRUC}$ :

$$\mathsf{s}\ \mathsf{STRUC} = \{\ (s, s')\ \mid\ \exists\, i,\ (\star, s, i, s') \in \mathsf{STRUC}\ \}$$

We now define what a constraint is, as well as double satisfaction (DS).

*Definition (transition constraint).* A *constraint* of a transition system $\mathsf{STRUC}$ is a subset

$$\mathsf{C} \subseteq \{\star\} \times \mathsf{State} \times \mathsf{Input} \times \mathsf{State}$$

such that $\mathsf{STRUC} \subseteq \mathsf{C}$.

*Definition (double satisfaction).* A structured contract $(\pi, \pi_{\mathsf{Tx}}, \mathsf{STRUC})$ is *vulnerable to double satisfaction* with respect to a constraint $\mathsf{C}$ whenever there exists another contract $(\pi, \pi_{\mathsf{Tx}}, \mathsf{STRUC}')$, with $\mathsf{STRUC} \subseteq \mathsf{STRUC}'$ and $\mathsf{s}\ \mathsf{STRUC} = \mathsf{s}\ \mathsf{STRUC}'$, such that $\mathsf{STRUC}' \cap \mathsf{C} \subsetneq \mathsf{STRUC}'$.

*Example (*TOGGLE *with extra constraint).* Consider a $(\pi, \pi_{\mathsf{Tx}}, \mathsf{TOGGLE})$ contract, with $\mathsf{State} := \mathbb{B}$ (i.e. Boolean), and $\mathsf{Input} := (\mathsf{toggle} \cup \{\star\}) \times \mathsf{Interval}[\mathsf{Slot}]$.

$$\textsc{DoNothing}\ \frac{}{\vdash (x) \xrightarrow[\textsc{TOGGLE}]{(\star, \_)} (x)} \qquad \textsc{Toggle}\ \frac{5 \leq j < k \leq 9}{\vdash (x) \xrightarrow[\textsc{Toggle}]{(toggle, [j,k])} (\neg x)}$$

We define a contract $\mathsf{STRUC}'$ by removing $5 \leq j < k \leq 9$ from rule TOGGLE, assume it has the same projections $\pi, \pi_{\mathsf{Tx}}$ as $\mathsf{TOGGLE}$, and define the constraint

$$\mathsf{C}(\star, \_, (t, [j, k]), \_) := (t = \mathsf{toggle}) \Rightarrow (5 \leq j < k \leq 9)$$

The contracts $\mathsf{STRUC}'$ and $\mathsf{STRUC}$ transition between the same states: $\mathsf{s}\ \mathsf{STRUC} = \mathsf{s}\ \mathsf{STRUC}' = x \mapsto x,\ x \mapsto \neg\, x$. Note that $\mathsf{STRUC} = \mathsf{STRUC}' \cap \mathsf{C} \subsetneq \mathsf{STRUC}'$, hence $\mathsf{STRUC}$ is vulnerable to DS with respect to $\mathsf{C}$. If such a vulnerability is deemed problematic by the contract author, then it is likely that it is important to them that *no other contracts* execute on-chain in the interval $[5, 9]$, which may be difficult to avoid in practice.

**Discussion.** Vulnerability to the DSP is related to the challenge of associating an *action of a transaction* (and therefore constraints on it) with a *specific structured contract* (or script implementing it) that constrains this action. Intuitively, any update to a contract state is associated with that contract. No scripts other than those implementing a given structured contract can place constraints on the update of that contract's state. So, only the parts of a transaction used to specify the updated state can be constrained in a way that is not vulnerable to double satisfaction.

Definition 3 states that for a given pair of states $(s, s')$ with at least one transition between them, a double satisfaction-vulnerable constraint $\mathsf{C}$ reduces the set of inputs $i$ such that $(\star, s, i, s') \in \mathsf{STRUC}$. Since all valid state update pairs are already specified by the set $\mathsf{s\ STRUC}$, any additional constraints on allowable inputs for a transition between $(s, s')$ are not relevant in the computation of the state update. Therefore, such additional constraints are not specific to $\mathsf{STRUC}$ and can be present in other scripts, making them vulnerable to DS.

**DSP mitigation.** The conventional way of addressing the DSP (even before giving it a formal definition) is to include a constraint in the implementing script(s) that forces them to fail if *any other scripts* are being run by the transaction. This effectively mitigates negative consequences of potential vulnerabilities of the given contract's constraints to the DSP. This is likely not a practical solution in many cases, however, as it is too restrictive. We note that, like the above example, this constraint is not on the contract state update, but rather, on the transaction. This means that it is itself vulnerable to DS. However, vulnerability of this constraint to DS will likely not be deemed to be a problem by script authors, as the purpose of introducing it is to mitigate the negative consequences of other constraints' vulnerabilities.

It is possible to define classes of contracts that are never vulnerable to DS. Deterministic contracts that have an end state for any pair of an input state and an input constitute such a class.

*Lemma (DS-free contracts).* A deterministic structured contract $(\pi, \pi_{\mathsf{Tx}}, \mathsf{STRUC})$ is not vulnerable to double satisfaction with respect to any constraint whenever for any $(s, i)$, there exists an $s'$ such that $(\star, s, i, s') \in \mathsf{STRUC}$.

*Proof.* Let $\mathsf{STRUC}$ be a contract, and $\mathsf{C}$ — a constraint of $\mathsf{STRUC}$, so that $\mathsf{STRUC} \subseteq \mathsf{C}$. Suppose $\mathsf{STRUC} \subseteq \mathsf{STRUC}'$, such that that $\mathsf{STRUC} \subseteq \mathsf{STRUC}' \cap \mathsf{C} \subseteq \mathsf{STRUC}'$, and $\mathsf{s\ STRUC} = \mathsf{s\ STRUC}'$. Suppose also that for any $(s, i)$ there exists an $s'$ such that $(\star, s, i, s') \in \mathsf{STRUC}$.

Now, let $(\star, s, i, s') \in \mathsf{STRUC}'$. By assumption, $(s, s') \in \mathsf{s\ STRUC}$.

We also know there is a unique (since $\mathsf{STRUC}$ is deterministic) $s''$ for the given $s, i$ such that $(\star, s, i, s'') \in \mathsf{STRUC}$. Since $\mathsf{STRUC} \subseteq \mathsf{STRUC}'$, and such an $s''$ must be unique, and $s' = s''$, we can conclude that $(\star, s, i, s') \in \mathsf{STRUC}$. Therefore, $\mathsf{STRUC} = \mathsf{STRUC}' = \mathsf{STRUC}' \cap \mathsf{C}$, and $\mathsf{STRUC}$ is not vulnerable to DS with respect to $\mathsf{C}$.

In Section 6, we discuss a possible solution to the DSP for payouts.

## 4 Message-passing in EUTxO

Conceptually, a message is data sent from a sender to a recipient [2]. In our design, a message is a data structure of type $\mathsf{Msg}$ encoded on the ledger in a specific way. It also includes a sender, receiver, and some data or assets. The content of $m \in \mathsf{Msg}$ is encoded as the $\mathsf{TokenName}$ of an NFT with the minting policy $\mathsf{msgsTT}$. It is encoded as such in order to maintain certain guarantees about the message's integrity, which are ensured by the NFT minting policy. A message $m \in \mathsf{Msg}$ consists of the following fields (see Fig. 1):

(i) an output reference $\mathsf{inUTxO} : \mathsf{OutputRef}$. An output with this reference must be spent when the message token is minted;
(ii) an index $\mathsf{msgIx} : \mathsf{Ix}$. It is used to uniquely identify a message whenever multiple messages are produced in association with spending a single UTxO entry;
(iii) an output $\mathsf{msgTo} : \mathsf{Output}$. It is an output that must be spent to validate the consumption of the message *by that recipient* (such an output may not be unique);
(iv) an output $\mathsf{msgFrom} : \mathsf{Output}$. It is an output that must be spent to validate production of the message *by that sender*. Specifically, the entry $m.\mathsf{inUTxO} \mapsto m.\mathsf{msgFrom}$ must be spent;
(v) a value $\mathsf{msgValue} : \mathsf{Value}$. It specifies the assets being sent. When a message is minted and placed in an output, this output must *also* contain these assets;
(vi) data $\mathsf{msgData} : \mathsf{Data}$. It is the data being sent via this message.

Each message requires a unique identifier to enable some of the applications we present later. Here, we use an approach based on the thread token mechanism to ensure NFT uniqueness [5]. This mechanism requires

Msg :=(inUTxO : OutputRef,
msgIx : Ix,
msgTo : Output,
msgFrom : Output,
msgValue : Value,
msgData : Data)
*Type of messages*

State := ℙ Msg
*The* MSGS *state is a set of messages*

Input := Tx
*The* MSGS *input is the full transaction*

Fig. 1: Message types

that the thread token's minting policy checks that a particular output reference is spent from the UTxO by the minting transaction, and exactly one token is minted under this policy. To uniquely identify a message NFT we use the output reference inUTxO, together with the message index msgIx. Duplication of unique identifiers is forbidden by the implementing scripts.

Sending a list of messages is done by submitting a transaction that (i) **mints** the NFTs encoding each of the messages, and (ii) for each message, spends the sender output with a redeemer containing the list of messages "from" that output. For an output to receive a list of messages, a transaction must spend the outputs containing the messages, and **burn** the message tokens. It must also spend the receiver output, and supply it with a redeemer containing the messages it is receiving.

The MSGS transition system specifies the rules for sending and receiving messages, see Fig. 5. A state $s \in \mathsf{State}$ of the MSGS contract is a set of messages, and represents messages that have been sent, but not yet received. The input type of MSGS is $\mathsf{Input} := \mathsf{Tx}$.

The function

$$\mathsf{msgTkn}\ msg := \{\ \mathsf{msgsTT} \mapsto \{\ msg \mapsto 1\ \}\ \}$$

encodes a message as a message token, recording the message data as its token name, and msgsTT as its minting policy. According to this policy, each message token minted by a transaction must be placed into a UTxO entry locked by a special validator, msgsVal, which only checks that any message token in that UTxO entry is burned. The message token minting policy msgsTT performs the same checks and assignments (1, 3, 5, 6, 7) that are in the MSGS specification in Fig. 5, with the notable exception of checking the non-duplication of existing messages, as required by (2). This cannot be checked explicitly by msgsTT because it cannot inspect the global set of existing messages under this policy, and must instead be proved as a consequence of the generation of the message's unique identifier. The type of the decoded redeemer for both msgsTT and msgsVal is $\{\star\}$, as they are not used in the implementation. The predicate _#_ takes two lists, returning True if they are disjoint, and $[\mathsf{f}\ a \parallel a \leftarrow as]$ denotes list comprehension. The contracts msgsTT and msgsVal implementing the MSGS specification are given in Fig. 3 and 4. The projection function $\pi_{\mathsf{Msg}}$ returns, for a given $utxo$, all messages encoded in the message tokens that exist in the UTxO set. It returns $\star$ when one or more messages have been duplicated or outputs incorrectly generated in the $utxo$. This is guaranteed by msgOutsOK, see Fig. 2 for the details.

$$\pi_{\mathsf{Msg}}\ utxo := \begin{cases} \{\ m \mid \_ \mapsto o \in utxo, \mathsf{msgTkn}\ m \subseteq o.\mathsf{value}\} & \text{if } \mathsf{msgOutsOK}\ utxo \\ \star & \text{otherwise} \end{cases}$$

See Appendix C for a proof sketch of the simulation relation between LEDGER and MSGS. Recall that this relation ensures the integrity of the implementation, i.e. that the implementation of MSGS via the msgsTT and msgsVal scripts only allows ledger updates that are mapped to *valid* MSGS transitions (by the $\pi$ and $\pi_{\mathsf{Tx}}$ projections). It uses the replay protection assumption.

**Replay protection assumption.** We must make an additional assumption about valid transactions in order to prove properties of the behavior of our MSGS program. This assumption is required for all forthcoming results making use of the fact that $(\mathsf{MSGS}, \pi, \pi_{\mathsf{Tx}})$ is a structured contract.

$$
\begin{aligned}
&\mathsf{msgOutsOK} : \mathsf{UTxO} \to \mathbb{B} \\
&\mathsf{msgOutsOK}\ utxo := \\
&\qquad \forall\, (i \mapsto o) \in utxo, \{\ \mathsf{msgsTT} \mapsto \{m \mapsto q\}\} \subseteq o.\mathsf{value} \Rightarrow \\
&\qquad\quad (q = 1) \\
&\qquad\quad \wedge\ \ (m \neq \star)\ \wedge\ (m.\mathsf{inUTxO} \mapsto \_ \notin utxo) \\
&\qquad\quad \wedge\ \ [\![\mathsf{msgsTT}]\!]\ (\star,\ (i.\mathsf{id},\ \mathsf{msgsTT})) \\
&\qquad\quad \wedge\ \ \forall (i' \mapsto o') \in utxo, i \neq i',\ \{\ \mathsf{msgsTT} \mapsto \{m \mapsto \_\}\} \notin o'.\mathsf{value} \\
&\qquad \wedge\ \forall\, (tx, ix) \mapsto o\ \in\ utxo,\ \forall\, i \in\ tx.\mathsf{inputs}, \\
&\qquad\quad\ [\![i.\mathsf{output.validator}]\!]\ (i.\mathsf{output.datum},\ i.\mathsf{redeemer},\ (tx,\ i)) \\
&\qquad\quad\ \wedge\ \ (ix \mapsto o) \in\ tx.\mathsf{outputs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{SR} := \{\mathsf{send}, \mathsf{receive}\} \\
&\quad \textit{Tag specifying whether message is being sent or received}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{getMsgRef} : \mathsf{Msg} \to (\mathsf{OutputRef},\ \mathsf{Ix}) \\
&\mathsf{getMsgRef}\ msg := (msg.\mathsf{inUTxO},\ msg.\mathsf{msgIx}) \\
&\quad \textit{Returns unique message identifier}
\end{aligned}
$$

Fig. 2: Projections and auxiliary MSGS functions

$$
\begin{aligned}
&\mathsf{msgsTT}' : \mathsf{Script} \to \mathsf{Script} \\
&[\![\mathsf{msgsTT}'\ mv]\!]\ (\_,\ (tx,\ pid)) :=\ \\
&\quad [\ \mathsf{getMsgRef}\ m \mid (\_,\ m) \leftarrow newOuts\ ]\ \#\ [\ \mathsf{getMsgRef}\ m \mid (\_,\ m) \leftarrow usedInputs\ ] \\
&\quad \wedge\ \forall\, (o,\ msg)\ \in\ newOuts, \\
&\qquad\ (msg,\ (msg.\mathsf{inUTxO}, msg.\mathsf{msgFrom}, \_))\ \in\ sndMsgs \\
&\qquad \wedge\ \{\ t\ \subseteq\ o.\mathsf{value}\ \mid\ \mathrm{dom}\ t\ =\ \{pid\}\ \} = \mathsf{msgTkn}\ msg \\
&\qquad \wedge\ o.\mathsf{validator}\ =\ mv\ \wedge\ o.\mathsf{value}\ \geq\ msg.\mathsf{msgValue} \\
&\quad \wedge\ \forall\, (i,\ msg)\ \in\ usedInputs,\ (msg,\ (\_, msg.\mathsf{msgTo}, \_))\ \in\ rcvMsgs \\
&\quad \wedge\ \Sigma_{(\_,msg)\ \in\ newOuts}\ \mathsf{msgTkn}\ msg\ \ +\ \ \Sigma_{(\_,msg)\ \in\ usedInputs}\ (-1)\ *\ (\mathsf{msgTkn}\ msg)\ \ =\ \ \{pid \mapsto tkns\ \in\ tx.\mathsf{mint}\ \} \\
&\quad \textbf{where} \\
&\qquad \mathsf{msgTkn}\ msg\ :=\ \{\ pid\ \mapsto\ \{msg \mapsto 1\}\ \} \\
&\qquad sndMsgs\ \ :=\ [\ (msg,\ i)\ \mid\ i \leftarrow\ tx.\mathsf{inputs},\ (sr,\ msg)\ \leftarrow\ i.\mathsf{redeemer},\ sr = \mathsf{send}\ ] \\
&\qquad rcvMsgs\ \ :=\ [\ (msg,\ i)\ \mid\ i \leftarrow\ tx.\mathsf{inputs},\ (sr,\ msg)\ \leftarrow\ i.\mathsf{redeemer},\ sr = \mathsf{receive}\ ] \\
&\qquad newOuts\ \ :=\ \{\ (o,\ msg)\ \mid\ o\ \in\ tx.\mathsf{outputs},\ \mathsf{msgTkn}\ msg\ \subseteq\ o.\mathsf{value}\ \} \\
&\qquad usedInputs :=\ \{\ (i,\ msg)\ \mid\ i\ \in\ tx.\mathsf{inputs}\ tx,\ \mathsf{msgTkn}\ msg\ \subseteq\ i.\mathsf{output.value}\ \}
\end{aligned}
$$

Fig. 3: Minting policy constructor for message tokens

$$\mathsf{msgsTT} := \mathsf{msgsTT}' \; \mathsf{msgsVal}$$

$$\begin{aligned}&[\![\mathsf{msgsVal}]\!]\ (\_,\ \_,\ (tx,\ i)) \ :=\\ &\quad \forall\, msg \in \{\ m \mid \mathsf{msgsTT}'\ (i.\mathsf{output.validator}) \mapsto \{\ m \mapsto 1\}\} \subseteq i.\mathsf{output.value}\ \},\\ &\quad \{\ \mathsf{msgsTT}'\ (i.\mathsf{output.validator}) \mapsto \{msg \mapsto -1\}\} \subseteq tx.\mathsf{mint}\end{aligned}$$

Fig. 4: Minting policy and validator for UTxO containing message tokens

(1) construct a list of messages encoded in redeemers

$$\begin{aligned}sndMsgs &:= [\ (msg,\ i) \mid i \leftarrow tx.\mathsf{inputs},\ (sr,\ msg) \leftarrow (i.\mathsf{redeemer}),\ sr = \mathsf{send}\ ]\\ rcvMsgs &:= [\ (msg,\ i) \mid i \leftarrow tx.\mathsf{inputs},\ (sr,\ msg) \leftarrow (i.\mathsf{redeemer}),\ sr = \mathsf{receive}\ ]\end{aligned}$$

(2) check that no new messages are duplicates

$$[\ \mathsf{getMsgRef}\ m \mid (\_,\ m) \leftarrow newOuts\ ]\ \#\ [\ \mathsf{getMsgRef}\ m \mid (\_,\ m) \leftarrow usedInputs\ ]\ \#\ [\ \mathsf{getMsgRef}\ m \mid m \leftarrow msgs\ ]$$

(3) compute the set of message token-containing outputs being created

$$newOuts\ :=\ \{\ (o,\ msg)\ \mid\ o\ \in\ tx.\mathsf{outputs},\ \mathsf{msgTkn}\ msg\ \subseteq\ o.\mathsf{value}\ \}$$

(4) check that all the messages are correctly constructed : correct sender output, sender has correct redeemer, output reference is spent, one message per output, output containing message token has correct validator and sufficient value

$$\begin{aligned}&\forall\ (o,\ msg)\ \in\ newOuts,\ (msg,\ (msg.\mathsf{inUTxO}, msg.\mathsf{msgFrom}, \_))\ \in\ sndMsgs\\ &\quad \wedge \{\ t\ \subseteq\ o.\mathsf{value}\ \mid\ \mathsf{dom}\ t\ =\ \{\mathsf{msgsTT}\}\ \} = \mathsf{msgTkn}\ msg\\ &\quad \wedge o.\mathsf{validator}\ =\ \mathsf{msgsVal}\ \wedge\ o.\mathsf{value}\ \geq\ msg.\mathsf{msgValue}\end{aligned}$$

(5) compute the set of message token-containing outputs being spent

$$usedInputs := \{\ (i,\ msg)\ \mid\ i\ \in\ tx.\mathsf{inputs},\ \mathsf{msgTkn}\ msg\ \subseteq\ i.\mathsf{output.value}\ \}$$

(6) check that all messages are correctly consumed : the receiver output is correct, input has correct redeemer, and message exists

$$\forall\ (i,\ msg)\ \in\ usedInputs,\ (msg,\ (\_, msg.\mathsf{msgTo}, \_))\ \in\ rcvMsgs\ \wedge\ msg \in msgs$$

(7) check minting and burning of message tokens :

$$\begin{aligned}&\Sigma_{(\_,msg)\ \in\ newOuts}\ \mathsf{msgTkn}\ msg\ \ +\ \ \Sigma_{(\_,msg)\ \in\ usedInputs}\ (-1)\ *\ (\mathsf{msgTkn}\ msg)\\ &\qquad =\ \{\ \mathsf{msgsTT} \mapsto tkns\ \in\ tx.\mathsf{mint}\ \}\end{aligned}$$

PROCESS

$$\star \vdash (msgs) \xrightarrow[\text{MSGS}]{tx} (\ (msgs \setminus [\ m \mid (\_,\ m) \leftarrow usedInputs\ ]) \cup [\ m \mid (\_,\ m) \leftarrow newOuts\ ]\ )$$

Fig. 5: Specification of the MSGS transition

For any $(slot, utxo, tx, utxo') \in \mathsf{LEDGER}$ such that $\pi\ utxo \neq \star$,

$$((tx, \_) \notin\ tx.\mathsf{outputRefs})\ \ \wedge\ \ (\forall\, m\ \in\ \pi\ utxo,\ \ m.\mathsf{inUTxO.id}\ \neq\ tx)$$

The above states that a transaction cannot re-add one of its inputs as a new output, and that an existing message token cannot be associated with the newly added transaction $tx$.

Both assumptions are consequences of *replay protection*: the general UTxO property that disallows the same transaction being valid multiple times within a given trace. Under reasonable conditions on initial ledger states this can be proven as a safety property [1], but this lies outside the scope of this work as we only consider individual steps here.

## 5 Message-passing use cases

In this section we discuss applications of the message-passing structured contract.

### 5.1 Memoization

There may be strict resource use constraints that apply to executing code on a blockchain. It may not be possible for a transaction to run the code of a large contract in its entirety. It may be desirable to divide such code into less memory- and CPU-intensive functions whose outputs are pre-computed for use by an aggregate function. A script may not trust values pre-computed off-chain, so a proof that a value was correctly computed on-chain is required. In this section we describe a technique for constructing such proofs using the MSGS contract. It is similar to a specific kind of caching called *memoization* [13], which is also how we refer to our approach.

Consider a function myFunction : MyInType → MyOutType which performs some computation. We define a script checkMyFunction (Fig. 6a), which wraps the computation done by myFunction. This script mints a message token with data $(fIn, fOut)$, such that myFunction $fIn = fOut$, and a script useMyFunction (Fig. 6b) that can consume a message with the redeemer $[(\mathsf{receive}, m)]$ when $m$ is addressed to an output locked by useMyFunction, and is sent by an output locked by checkMyFunction. This message serves as a proof that myFunction $fIn = fOut$, so, useMyFunction can perform a computation checkStuff relying on the fact that myFunction $fIn = fOut$. Note that msgTo is not constrained by this contract, so that the generated message can be addressed to any recipient. We give the result that formalizes the use of message-passing to prove that

$$
\begin{aligned}
&[\![\mathsf{checkMyFunction}]\!]\ (\_,\ r,\ (tx,\ i)) :=\\
&\quad m.\mathsf{inUTxO} = i.\mathsf{outputRef}\\
&\wedge\ m.\mathsf{msgFrom} = i.\mathsf{output}\\
&\wedge\ m.\mathsf{msgValue} = 0\\
&\wedge\ \mathsf{msgTkn}\ m \subseteq tx.\mathsf{mint}\\
&\wedge\ \mathsf{myFunction}\ fIn = fOut\\
&\ \mathbf{where}\\
&\ [(\mathsf{send},\ m)] = r\\
&\ (fIn,\ fOut) = m.\mathsf{msgData}
\end{aligned}
$$

(a) Script minting message token.

$$
\begin{aligned}
&[\![\mathsf{useMyFunction}]\!]\ (d,\ r,\ (tx,\ i)) :=\\
&\quad (\ \ m.\mathsf{msgFrom} = (\mathsf{checkMyFunction}, \_, \_)\\
&\quad \wedge\ m.\mathsf{msgTo} = i.\mathsf{output}\\
&\quad \wedge\ (-1) * (\mathsf{msgTkn}\ m) \subseteq tx.\mathsf{mint}\\
&\quad \wedge\ \mathsf{checkStuff}\ d\ r\ (tx,\ i)\ (fIn,\ fOut)\ )\\
&\ \vee\ \mathsf{checkOtherStuff}\ d\ r\ (tx,\ i)\\
&\ \mathbf{where}\\
&\ [(\mathsf{receive},\ m)] = r\\
&\ (fIn,\ fOut) = m.\mathsf{msgData}
\end{aligned}
$$

(b) Script using the memoized output.

Fig. 6: Scripts for memoizing the output of myFunction.

myFunction $fIn = fOut$.

*Lemma (Verified input-output pairs).* For any $(s,\ u,\ tx, u') \in \mathsf{LEDGER}$, with $\pi\ u \neq \star$ and $(i,\ (\mathsf{useMyFunction}, v, d),\ r) \in tx.\mathsf{inputs}$, such that

$$
\begin{aligned}
[(\mathsf{receive},\ m)] &= r\\
(fIn,\ fOut) &= m.\mathsf{msgData}\\
m.\mathsf{msgFrom} &= (\mathsf{checkMyFunction}, \_, \_)
\end{aligned}
$$

necessarily myFunction $fIn = fOut$, and $m.\mathsf{msgTo} = (\mathsf{useMyFunction}, v, d)$. See Appendix C for a proof sketch.

### 5.2 Contracts using message-passing

Stateful contract interaction, or communication, in the EUTxO model is implemented via dependencies [5]. A *dependency* of a script $c$ is a constraint requiring that another script $c'$ must be executed within the same transaction, possibly with specific arguments. For example, the validation of the minting policy msgsTT implementing MSGS depends on the validation of the relevant sender and receiver output-locking scripts with

certain redeemers. MSGS introduces structure to the ad-hoc use of dependencies in implementing interaction and communication between contracts or scripts. It allows the interacting scripts to execute asynchronously, and to depend on the message-passing scripts msgsTT and msgsVal, rather than on each other directly. We say that stateful contracts *use message-passing* when they require the production or consumption of messages to or from scripts implementing the contract. We formalize this notion in this section.

Message-passing specification is closely integrated with ledger semantics, and inspects the scripts, redeemers, and datums of the input transaction. Because of this, a message-passing contract must also inspect these in order to correctly construct a message. So, a state projection function for a contract that uses message-passing includes the UTxO entry relevant to the contract state, in full. The contract input is the complete transaction.

Suppose that $\mathsf{F} : \mathsf{Output} \mapsto \mathbb{B}$ is a constraint on outputs, and $\mathsf{c} : \mathsf{UTxO} \to \mathbb{B}$ is a constraint on a valid UTxO state. The contract denoted by $(\pi_{\mathsf{F},\mathsf{c}},\ \pi_{\mathsf{Tx}},\ \mathsf{STRUC})$ is a structured contract with

$$\begin{aligned}
\mathsf{State} &:= \{ i \mapsto o \ \in\ u \ \mid\ u \in \mathsf{UTxO},\ \mathsf{F}\ o \ \} \\
\pi_{\mathsf{F},\mathsf{c}}\ u\ &:= \begin{cases} \{ i \mapsto o \ \in u \ \mid\ \mathsf{F}\ o \} & \text{if } \mathsf{c}\ u \\ \star & \text{otherwise} \end{cases} \\
\pi_{\mathsf{Tx}} &:= \mathsf{id}
\end{aligned}$$

We can combine STRUC and MSGS to construct the structured contract $\mathsf{STRUC}_{\mathsf{MSGS}}$,

$$\begin{aligned}
\pi_{\mathsf{State}-M}\ u\ &:= \begin{cases} (\pi_{\mathsf{F},\mathsf{c}}\ u, \pi_{\mathsf{Msg}}\ u) & \text{if } \ \pi_{\mathsf{F},\mathsf{c}}\ u \ \neq\ \star\ \neq\ \pi_{\mathsf{Msg}}\ u \\ \star & \text{otherwise} \end{cases} \\
\pi_{\mathsf{Tx}-M}\ &:=\ \mathsf{id}_{\mathsf{Tx}} \\
\mathsf{STRUC}_{\mathsf{MSGS}} &:= \{\ (\star, (s, m), tx, (s', m'))\ \mid\ (\star, s, tx, s') \in \mathsf{STRUC}, (\star, m, tx, m') \in \mathsf{MSGS}\ \}
\end{aligned}$$

We call this contract *message-augmentation* of STRUC. We define the following functions that filter messages sent or received by STRUC :

$$\begin{aligned}
\mathsf{getFromSTRUCmsgs}\ \mathit{msgs} &:= \{\ m\ \mid\ m\ \in \mathit{msgs},\ \mathsf{F}\ (m.\mathsf{msgFrom})\ \} \\
\mathsf{getToSTRUCmsgs}\ \mathit{msgs} &:= \{\ m\ \mid\ m\ \in \mathit{msgs},\ \mathsf{F}\ (m.\mathsf{msgTo})\ \}
\end{aligned}$$

We now state a result that says that, for a given $\mathsf{F},\mathsf{c}$, all messages to and from the contract $(\pi_{\mathsf{F},\mathsf{c}},\ \pi_{\mathsf{Tx}},\ \mathsf{STRUC})$ for a given step are generated and consumed only under some appropriate conditions. In particular, messages *from* the contract can only be produced when a script locking an output of this contract "authorizes" the minting of this message by successfully validating with a redeemer containing the message(s) being produced. Consuming messages addressed *to* the contract requires the validation of a script locking the recipient output, given a redeemer containing these messages. This result follows directly from the MSGS specification and implementation.

*Lemma (*STRUC *messages generated correctly).* For any $(\star, (s, m), tx, (s'm')) \in \mathsf{STRUC}_{\mathsf{MSGS}}$,

$$\begin{aligned}
&\forall\ \mathit{msg} \in \mathsf{getFromSTRUCmsgs}\ (m' \setminus m), (\mathit{msg}.\mathsf{inUTxO} \mapsto (\mathit{msg}.\mathsf{msgFrom}) \in s) \\
&\quad \wedge\ \ \mathit{msg}.\mathsf{inUTxO} \mapsto (\mathit{msg}.\mathsf{msgFrom}) \notin s') \\
&\quad \wedge\ \ \forall\ \mathit{inp} \in tx.\mathsf{inputs}, (\mathit{inp}.\mathsf{outputRef} = \mathit{msg}.\mathsf{inUTxO} \Rightarrow\ (\mathsf{send}, \mathit{msg}) \in \mathit{inp}.\mathsf{redeemer})
\end{aligned}$$

$$\begin{aligned}
&\forall\ \mathit{msg} \in \mathsf{getToSTRUCmsgs}\ (m \setminus m'), (_ \mapsto \mathit{msg}.\mathsf{msgTo} \in s) \\
&\quad \wedge\ \ \exists\ \mathit{inp} \in tx.\mathsf{inputs},\ \mathit{inp}.\mathsf{output} = \mathit{msg}.\mathsf{msgTo} \wedge (\mathsf{receive}, \mathit{msg}) \in \mathit{inp}.\mathsf{redeemer})
\end{aligned}$$

*Definition (Uses message-passing).* We say that STRUC *uses message-passing* whenever the set defined by

$$\mathsf{getMSGS}\ (\star, (s, m), tx, (s'm')) \ :=\ \mathsf{getFromSTRUCmsgs}\ (m' \setminus m) \cup \mathsf{getToSTRUCmsgs}\ (m \setminus m')$$

is non-empty for some $(\star, (s, m), tx, (s'm')) \in \mathsf{STRUC}_{\mathsf{MSGS}}$.

We define the set of *payouts* in the step $(\star, (s, m), tx, (s'm')) \in \mathsf{STRUC}_{\mathsf{MSGS}}$ by

$$\mathsf{getPayouts}\ (\star, (s, m), tx, (s'm')) := \{ msg \in \mathsf{getFromSTRUCmsgs}\ (m' \setminus m) \mid msg.\mathsf{msgValue} > 0 \wedge \neg\, (\mathsf{F}\ (msg.\mathsf{msgTo}) \}$$

Whenever this set is non-empty for some step in $\mathsf{STRUC}_{\mathsf{MSGS}}$, we say that it *makes payouts with messages*.

**Discussion.** A contract is said to use message-passing whenever there is a step in $\mathsf{STRUC}_{\mathsf{MSGS}}$ that requires the production or consumption of a non-empty set of messages to or from STRUC. Some computation performed by contracts implementing STRUC may be contingent on receiving a specific message. For example, accepting a payment message sent by another contract.

Contracts that use message-passing share common features that are both necessary and sufficient for a script $c$ implementing the contract to be able to interface with the message-passing contract : (i) the script's redeemer must decode to a list of sent/received messages, and (ii) the script must ensure that the corresponding messages are included in the transaction's mint field.

For a given step $(\star, s, t, s') \in \mathsf{STRUC}$, we refer to the messages sent and received by outputs that make up $s$, i.e. those filtered by F, as a script's *communication*. Calculating $s'$ for the given $(s, t)$ is the STRUC contract's computation. STRUC may still include arbitrary dependencies on scripts implementing contracts other than MSGS. Specifying when a contract has no non-message dependencies is important for determining when it is guaranteed to be able to progress. This is, however, the subject of future work.

## 6 Messages as payouts

A *payout* is a message that is from STRUC, but not addressed to STRUC, and specifies a sent value greater than zero. The function that returns all the payouts for a given contract, getPayouts, is a function of the start and end MSGS states only, given that it is applied to a valid step of $\mathsf{STRUC}_{\mathsf{MSGS}}$.

**Message-augmented** PAYOUT**.** Messages can serve as intermediate stores of assets whose transfer has been authorized by the sender, but before they are accepted by the receiver. We give an example of a message-augmented contract that makes payouts until it runs out of funds.

In the upcoming example we make use of the thread token design pattern [5]. Suppose NFT is a thread token which, upon minting, must be placed into the output given by $(\mathsf{payout}, \mathsf{NFT} + \mathsf{a}, \star)$, for some $\mathsf{a} > 0$. This NFT serves as a unique identifier of the UTxO entry currently containing the datum and value representing part (or all) of the contract state. We define the filter F which returns only the outputs containing the NFT specific to the PAYOUT contract implementation (plus some assets in quantities less than $a$), and a constraint c requiring that there is exactly one NFT in the UTxO :

$$\mathsf{F}\ o := \mathsf{NFT} \subseteq o.\mathsf{value}$$

$$\begin{aligned}\mathsf{c}\ u := {} & \exists!\, i \mapsto o \in u, \mathsf{NFT} \subseteq o.\mathsf{value} \wedge a + \mathsf{NFT} \geq o.\mathsf{value} \\ & \wedge\ (\forall\, i \mapsto o \in u, \mathsf{NFT} \subseteq o.\mathsf{value} \Rightarrow o.\mathsf{validator} = \mathsf{payout})\end{aligned}$$

As with all message-passing scripts, the redeemer type for the payout script is $[(\mathsf{SR} \times \mathsf{Msg})]$, and the datum type is $\{\star\}$, which is never inspected. We give the two transition rules of the message-augmented specification $\mathsf{PAYOUT}_{\mathsf{MSGS}}$. The first, MSGSOnly, applies when the PAYOUT state is not updated, and only the message-passing contract is updated. The second, PayoutV, applies when a payout of value $\mathsf{v} > 0$ is made to the address script recipient $\neq$ payout. Only one payout at a time is possible, and it must have message index 1.

We note that rather than defining PAYOUT first, then $\mathsf{PAYOUT}_{\mathsf{MSGS}}$, we define the latter directly. The reason for this is that this approach allows us to express the requirements on the payout messages as constraints on the MSGS state update, rather than in terms of on the outputs contained in the carrying $tx$. This is important in addressing the DSP using message-passing.

The specification is given by

$$\textsc{MsgsOnly}\ \frac{\_ \vdash (m) \xrightarrow[msgs]{tx} (\boldsymbol{m'}) \qquad (i, o, \_) \notin tx.\mathsf{inputs}}{\vdash \begin{pmatrix} \{i \mapsto o\} \\ m \end{pmatrix} \xrightarrow[\text{PAYOUT-MSGS}]{tx} \begin{pmatrix} \{i \mapsto o\} \\ \boldsymbol{m'} \end{pmatrix}}$$

$$ms := (i, 1, (\mathsf{recipient}, \mathsf{v}, \star), (\mathsf{payout}, \mathsf{NFT} + a, \star), v, \star)$$

$$\mathsf{recipient} \neq \mathsf{payout} \qquad ms \in m' \setminus m \qquad 0 \leq \mathsf{v} \leq a$$

$$\textsc{PayoutV}\ \frac{\_ \vdash (m) \xrightarrow[msgs]{tx} (\boldsymbol{m'})}{\vdash \left(\begin{matrix}\{i \mapsto (\mathsf{payout}, \mathsf{NFT} + a, \star)\} \\ m\end{matrix}\right) \xrightarrow[\text{PAYOUT-MSGS}]{tx} \left(\begin{matrix}\boldsymbol{\{(tx, ix) \mapsto (\mathsf{payout}, \mathsf{NFT} + (a - \mathsf{v}), \star)\}} \\ \boldsymbol{m'}\end{matrix}\right)}$$

It is possible to construct a similar example to track the assets being deposited into a stateful contract using messages, representing pay-ins to that contract.

MSGS **payouts and double satisfaction.** We gave a naive approach to payouts in (ii) in Section 3. This approach is vulnerable to DS, since the constraint requiring a payout to be made is strictly on the input transaction, rather than the state. Naive payout outputs can be produced and consumed by any valid transaction at any time, independently of the state update of any contract. Without a mechanism to *associate a payout with its sender*, is not possible to include naive payouts in a contract's state.

Intuitively, making payouts via messages provides such a mechanism by ensuring that the sender of the payout is recorded in the message token, and that the message token has a unique identifier. Formally, since making payouts via messages can be expressed as a predicate on a pair of message states, rather than on the input transaction, constraints on message payouts are not vulnerable to DS for a message-enhanced contract. Let us consider payouts as specified in the definition of payouts via messages, given in Definition 5.2. We can state the following result:

*Lemma (*MSGS*-payouts and DS)* Suppose $(\pi_{\mathsf{F,c}}, \pi_{\mathsf{Tx}}, \mathsf{STRUC})$ is a structured contract, and $\mathsf{STRUC}_{\mathsf{MSGS}}$ is its message-enhanced version. Let $\mathsf{C} \supset \mathsf{STRUC}_{\mathsf{MSGS}}$ be a constraint expressible in terms of some predicate $C'$ on the set of payout messages,

$$\mathsf{C}\ (\star, (s, m), tx, (s'm')) := \mathsf{C}'\ (\mathsf{getPayouts}_{\mathsf{STRUC}}\ (\star, (s, m), tx, (s', m')))$$

Then, the contract $\mathsf{STRUC}_{\mathsf{MSGS}}$ is not vulnerable to DS with respect to C.

*Proof.* Suppose that $(\pi_{\mathsf{F,c}}, \pi_{\mathsf{Tx}}, \mathsf{STRUC}')$ is another (more permissive) structured contract, with $\mathsf{STRUC}_{\mathsf{MSGS}} \subseteq \mathsf{STRUC}'_{\mathsf{MSGS}}$, and $\mathsf{s}\ \mathsf{STRUC}_{\mathsf{MSGS}} = \mathsf{s}\ \mathsf{STRUC}'_{\mathsf{MSGS}}$. For any $(\star, (s, m), tx, (s', m')) \in \mathsf{STRUC}'_{\mathsf{MSGS}}$, by definition,

$$\begin{aligned}&\mathsf{getPayouts}_{\mathsf{STRUC}'}\ (\star, (s, m), tx, (s', m')) = \\ &\quad \{\ ms \in m' \setminus m \mid \mathsf{F}\ (ms.\mathsf{msgFrom}) \wedge ms.\mathsf{msgValue} > 0 \wedge \neg\ (\mathsf{F}\ (ms.\mathsf{msgTo}))\ \}\end{aligned}$$

which depends only on $F$ (which is the same for STRUC and STRUC′), and $m' \setminus m$. Now, by the assumed preconditions on STRUC′, for any $(\star, (s, m), tx, (s', m')) \in \mathsf{STRUC}'_{\mathsf{MSGS}}$, we can find $(\star, (s, m), tx', (s', m')) \in \mathsf{STRUC}_{\mathsf{MSGS}} \subseteq \mathsf{STRUC}'_{\mathsf{MSGS}}$. Then,

$$\begin{aligned}\mathsf{C}\ (\star, (s, m), tx, (s'm')) &= \mathsf{C}'\ (\mathsf{getPayouts}_{\mathsf{STRUC}'}\ (\star, (s, m), tx, (s', m'))) \\ &= \mathsf{C}'\ (\mathsf{getPayouts}_{\mathsf{STRUC}}\ (\star, (s, m), tx', (s', m'))) \\ &= \mathsf{C}\ (\star, (s, m), tx', (s'm'))\end{aligned}$$

Therefore, any transition in $\mathsf{STRUC}'_{\mathsf{MSGS}}$ must also satisfy C. We get that $\mathsf{STRUC}'_{\mathsf{MSGS}} \cap \mathsf{C} = \mathsf{STRUC}'_{\mathsf{MSGS}}$, meaning that $\mathsf{STRUC}_{\mathsf{MSGS}}$ is not vulnerable to DS with respect to such a C.

# 7 Discussion

## 7.1 Related work

Message-passing is the backbone of distributed computing [2,9]. The $\pi$-calculus process calculus has been developed to formalize message-passing between processes in distributed computing scenarios [18]. We conjecture that it may be possible to apply this formalism to message-passing between structured contracts.

The UTxO ledger model introduced by Bitcoin [19], as well as EUTxO ledger implementations [11], are themselves message-passing schemes, wherein a transaction is a message to a script. Our scheme reinterprets messages in a way that allows them to have a single verified sender output, and a receiver that is also an output. The contract MSGS can be viewed as a kind of linear sub-ledger within LEDGER, which can be used as a tool in specification and verification of properties of communicating contracts.

In account-based ledgers [4,15,26], (synchronous) message-passing is the default mode of communication between contracts. The Scilla programming language [25], with its emphasis on separating communication from computation for stateful contracts on the Zilliqa ledger [26], inspired this work. Even though Scilla was developed for the account-based ledger model, the communicating automata structure it uses to model contracts may be useful in describing message-passing structured contracts as well.

Existing work on rigs [10], which are cryptographic data structures that provide integrity-at-a-distance, presents an approach to maintaining data integrity across potentially multiple state-managing machines. Aspects of this approach are similar in spirit to the thread-token technique we use to uniquely identify messages and ensure non-duplication of message tokens on the ledger; both are based on temporal and causal dependencies of operations on one another.

A version of asynchronous, but centralized, message-passing is implemented in the ERC-20 Ethereum contract for fungible tokens [12]. To transfer an amount of tokens from a sender to a receiver, the total amount being transferred must first be sent and recorded in an intermediate data structure, then received and withdrawn from the data structure. The total amount does not have to be withdrawn in its entirety, which is different from our design, where a message can only be consumed in full. The ERC-20 design is also primarily for asset transfers, whereas ours can be used to communicate authenticated data as well. We also note that in an account-based ledger, transactions interacting with the same stateful contract like ERC-20 can usually be reordered. However, implementing message-passing via a centralized data-storage contract on an EUTxO ledger would significantly reduce the possibility of reordering message-passing transactions, and therefore concurrency.

Formalization of blockchain and ledger functionality forms a foundation for rigorous reasoning about smart contracts security, discussed in the detailed overview [24]. Mathematical models of EUTxO and UTxO ledgers and smart contracts on those ledgers, including ours, often specify a simplified version of actual implementations [7,5,14,20,3,23].

## 7.2 Future work

The scheme we presented in Fig. 5 is such that the outputs that must be spent in order to consume a given message are fully specified (via the msgTo field of the message), including their scripts, values, and datums. In future work, this constraint could be relaxed for a more permissive and versatile system design.

A *time of expiry* can be added to the message structure and used to specify a time after which a message can be consumed under different constraints. This could allow structured contracts to retract any assets sent via a message but not received by a set time. Changing the type of the message-passing redeemer from a list of messages to a list of messages (for communication) together with some extra data (for computation) can also make MSGS more expressive. It may be useful in more easily enabling a given script to engage in both computation and communication as a result of applying a transaction.

In this work, we did not specify trace-based properties of LEDGER or any structured contract STRUC. This topic, in general, is the subject of future work. Of particular interest are structured contracts that can be guaranteed to take a step without the need for executing "external" scripts, i.e. ones other than those used to implement that contract. It may be unrealistic for a contract to always take a step without *any* external contracts validating, e.g. running a script which locks funds used for paying into the contract. However, it seems feasible to limit a structured contract's dependencies to message-passing only. Formalizing and proving properties about this class of structured contracts in the future is of interest.

In the future, we intend to mechanize this contract and its applications in Agda, building upon the formal EUTxO ledger model [7] and structured contracts framework [27], hoping to eventually integrate our work into the more realistic mechanization of the Cardano ledger [16].

### 7.3 Conclusion

Principled approaches to implementing and reasoning about the behavior of stateful smart contracts in the EUTxO ledger model have already been formalized in existing work. However, such models do not include any special provisions for analyzing communication among contracts. In this work, we focus on formalizing communication of data and assets among scripts as well as the stateful contracts they implement. Our contribution begins with the formalization of a common problem in contract interaction — the double satisfaction problem. The instance of this problem that is encountered most often by script authors is the challenge of performing correct accounting in sending assets to a recipient, i.e. making a payout.

We define what a message data structure is in the context of an EUTxO ledger — a unique identifier associated with the data and assets being sent, as well as its sender and receiver outputs. We then define how to use the ledger asset-minting mechanism to encode messages as tokens which appear in the UTxO set. To track sending and receiving messages, we define a distributed structured contract MSGS. As such, it is a stateful contract specified in the same small-steps semantic framework as the ledger itself, and comes with a proof of the integrity of its implementation. We describe communication between scripts and structured contracts in terms of sending and receiving messages, which facilitates formal reasoning about contract communication. Our message-passing scheme enables asynchronous communication that does not require multiple participants to coordinate transaction construction off-chain to execute dependent scripts.

To give examples of formal reasoning about the message-passing contract and its applications, we present two use cases. The first is a variation on memoization, wherein message tokens serve as proof artifacts of successful script computations. The second formalizes the idea of structured contract communication via message-passing. Any structured contract may participate in message-passing so long as it ensures that the message tokens it mints correspond to messages specified in its redeemer. We formalize when a message constitutes a "payout" from a contract, and then demonstrate how expressing payouts as messages can address vulnerability to the DSP in the case of payouts. A notable limitations of message-passing in general, and, in particular, of using it to address the DSP, is the requirement that the message token minting policy must be run to both send and receive a message. In a realistic setting, this may require the payment of additional script execution fees by the transaction authors.

We presented a contract designed for a minimal EUTxO system. Some changes may be necessary to adjust our contract design to an EUTxO ledger system with additional features or slight variations, e.g. one that uses hashes instead of full preimages of scripts and transactions to reduce on-chain memory use. We believe that message-passing and its applications may be implemented on most EUTxO ledger designs that feature scripts and datums in outputs, redeemers in transactions for spending outputs, transaction summaries passed as arguments to the script evaluator, and the multi-asset functionality relying on user-defined policies for controlling asset minting.

**Acknowledgments.** We would like to thank Manuel Chakravarty for providing inspiration for this work, by being one of the first proponents of the message-passing idiom for concurrency in the EUTxO model. We would also like to thank Philip Wadler for useful discussions.

## 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. Andrews, G.: Foundations of Multithreaded, Parallel, and Distributed Programming. Addison-Wesley (1999)
3. Bartoletti, M., Bracciali, A., Lepore, C., Scalas, A., Zunino, R.: A formal model of Algorand smart contracts (2021)
4. Buterin, V.: Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/en/whitepaper/ (2014)
5. 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: Margaria, T., Steffen, B. (eds.) 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. Lecture Notes in Computer Science, vol. 12478, pp. 89–111. Springer (2020). https://doi.org/10.1007/978-3-030-61467-6_7, https://doi.org/10.1007/978-3-030-61467-6_7
6. Chakravarty, M.M.T., Chapman, J., MacKenzie, K., Melkonian, O., Müller, J., Jones, M.P., Vinogradova, P., Wadler, P., Zahnentferner, J.: $UTXO_{ma}$: UTXO with multi-asset support. In: Margaria, T., Steffen, B. (eds.) 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. Lecture Notes in Computer Science, vol. 12478, pp. 112–130. Springer (2020). https://doi.org/10.1007/978-3-030-61467-6_8, https://doi.org/10.1007/978-3-030-61467-6_8
7. Chakravarty, M.M.T., Chapman, J., MacKenzie, K., Melkonian, O., Peyton Jones, M., Wadler, P.: The extended UTXO model. In: Bernhard, M., Bracciali, A., Camp, L.J., Matsuo, S., Maurushat, A., Rønne, P.B., Sala, M. (eds.) Financial Cryptography and Data Security - FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers. Lecture Notes in Computer Science, vol. 12063, pp. 525–539. Springer (2020). https://doi.org/10.1007/978-3-030-54455-3_37, https://doi.org/10.1007/978-3-030-54455-3_37
8. Corduan, J., Güdemann, M., Vinogradova, P.: A formal specification of the Cardano ledger. https://github.com/input-output-hk/cardano-ledger/releases/latest/download/shelley-ledger.pdf (2019)
9. Coulouris, G., Dollimore, J., Kindberg, T.: Distributed Systems: Concepts and Design (International Computer Science). Addison-Wesley Longman, Amsterdam (2005)
10. Coward, K., Toliver, D.R.: Simple rigs hold fast (2022)
11. Ergo Team: Ergo: A Resilient Platform For ContractualMoney. https://whitepaper.io/document/753/ergo-1-whitepaper (2019)
12. Ethereum Team: ERC-20 TOKEN STANDARD. https://ethereum.org/en/developers/docs/standards/tokens/erc-20 (2023)
13. Field, A., Harrison, P.: Functional Programming. International computer science series, Addison-Wesley (1988), https://books.google.ca/books?id=nYtQAAAAMAAJ
14. Gabbay, M.J.: Algebras of UTxO blockchains. Mathematical Structures in Computer Science **31**(9), 1034–1089 (2021). https://doi.org/10.1017/S0960129521000438
15. Goodman, L.: Tezos—a self-amending crypto-ledger (white paper). https://tezos.com/whitepaper.pdf (2014)
16. Knispel, A., Melkonian, O., Chapman, J., Hill, A., Jääger, J., DeMeo, W., Norell, U.: Formal specification of the Cardano blockchain ledger, mechanized in Agda. https://omelkonian.github.io/data/publications/cardano-ledger.pdf (2024), under submission
17. Knispel, A., Vinogradova, P.: 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)
18. Milner, R.: Communicating and Mobile Systems: The Pi-Calculus. Cambridge University Press, Cambridge, UK (1999)
19. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/en/bitcoin-paper (October 2008)
20. Nester, C.: A foundation for ledger structures. In: Anceaume, E., Bisière, C., Bouvard, M., Bramas, Q., Casamatta, C. (eds.) 2nd International Conference on Blockchain Economics, Security and Protocols, Tokenomics 2020, October 26-27, 2020, Toulouse, France. OASIcs, vol. 82, pp. 7:1–7:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/OASICS.TOKENOMICS.2020.7, https://doi.org/10.4230/OASIcs.Tokenomics.2020.7
21. Norell, U.: Dependently typed programming in Agda. In: International School on Advanced Functional Programming. pp. 230–266. Springer (2008)
22. Plotkin, G.: A structural approach to operational semantics. J. Log. Algebr. Program. **60-61**, 17–139 (07 2004). https://doi.org/10.1016/j.jlap.2004.05.001
23. Rupić, K., Rožić, L., Derek, A.: Mechanized Formal Model of Bitcoin's Blockchain Validation Procedures. In: Bernardo, B., Marmsoler, D. (eds.) 2nd Workshop on Formal Methods for Blockchains (FMBC 2020). Open Access

Series in Informatics (OASIcs), vol. 84, pp. 7:1–7:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2020). https://doi.org/10.4230/OASIcs.FMBC.2020.7, https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.FMBC.2020.7

24. Sánchez, C., Schneider, G., Leucker, M.: Reliable smart contracts: State-of-the-art, applications, challenges and future directions. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice. pp. 275–279. Springer International Publishing, Cham (2018)
25. 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)
26. Team, T.Z.: The ZILLIQA Technical Whitepaper. https://docs.zilliqa.com/whitepaper.pdf (2017)
27. Vinogradova, P., Melkonian, O., Wadler, P., Chakravarty, M., Krijnen, J., Jones, M.P., Chapman, J., Ferariu, T.: Structured contracts in the EUTxO ledger model. https://omelkonian.github.io/data/publications/eutxo-struc.pdf (2024), under submission

## A Notation

Fig. 7 clarifies non-standard notation we use throughout this document.

$$
\begin{array}{rll}
\star : & \{\star\} & \textit{the one-element set, and its one inhabitant} \\
a : & A \cup \{\star\} & \textit{maybe type over } A \\
B \in & \mathbb{P}\, A & \textit{the set of all subsets of } A, \textit{ so } B \subseteq A \\
\mathsf{r.x} : & (A, \ldots, B) \to B & \textit{accessing a (named) field of a record} \\
\mathsf{snd} : & (A \times B) \to B & \textit{second projection} \\
\textit{Key} \mapsto \textit{Value} \subseteq & \{\ k \mapsto v \mid k \in \textit{Key},\ v \in \textit{Value}\ \} & \textit{finite map with unique keys} \\
[a_1; \ldots; a_k] : & [A] & \textit{list over type } A \\
a :: as : & [A] & \textit{list with head } a \textit{ and tail } as \\
as +\!\!+ as' : & [A] & \textit{list concatenation} \\
[\ \mathsf{f}\ a \mid a \leftarrow as\ ] : & [B] & \textit{list comprehension over list } as, \textit{ given } \mathsf{f} : A \to B \\
as \,\#\, as' : & [A] \to [A] \to \mathbb{B} & \textit{check that two lists are disjoint}
\end{array}
$$

Fig. 7: Notation

## B Details of the EUTxO ledger model

Although all the details of Extended UTxO model have been adequately described in previous work [7,5], we include the basic definitions here in order to keep this document self-contained.

Fig. 8 enumerates the basic types of the $\text{EUTxO}_{\text{MA}}$ model: an extension of EUTxO to accommodate native custom assets. Based on those, the validity of a transaction in a $\text{EUTxO}_{\text{MA}}$-based ledger is prescribed by the rules of Fig. 9.

## C Proofs related to messaging

*Proof sketch of simulation relation for* MSGS. Suppose $(\star, u, tx, u') \in \mathsf{LEDGER}$, and that $\pi\ u \neq \star$.

Suppose no message tokens are being minted/burned, so $\{\ \mathsf{msgsTT} \mapsto tkns\ \in\ \mathsf{mint}\ tx\ \} = \{\}$. By $\mathsf{msgOutsOK}\ u$, all message tokens are locked by msgsVal. This script ensures that any spent token must be burned. Since no tokens are burned, no tokens are spent (i.e. in inputs of $tx$). By ledger rule 5 (let us call this rule POV), no tokens are in the outputs of $tx$ either. By the replay protection assumption, $tx$ does not add an output reference to $u$ that is the same as the $msg$.inUTxO of an existing message $msg$. There are no new messages added to $u$, so all messages in $u'$ are still generated according to msgsTT, and still not duplicated. Therefore, the first conjunct of $\mathsf{msgOutsOK}\ u'$ holds. By ledger rule 7, all inputs of $tx$ validate, and all outputs of $tx$ are added to $u$ correctly (by LEDGER), so the second conjunct of $\mathsf{msgOutsOK}\ u'$ holds. So, $\pi\ u' \neq \star$. Now, since the total amount of message also tokens is unchanged, $\pi\ u' = \pi\ u$. This also follows from (7) in the MSGS specification (Fig. 5) whenever neither the outputs nor the inputs of $tx$ contain any message tokens. The other constraints of MSGS are satisfied trivially by an empty set of tokens minted/burned by $tx$, and by non-duplication of existing tokens (required by $\mathsf{msgOutsOK}\ u'$). Therefore, $(\star, \pi\ u, tx, \pi\ u') = (\star, \pi\ u, tx, \pi\ u) \in \mathsf{MSGS}$.

Suppose now that $\{\} \neq \{\mathsf{msgsTT} \mapsto tkns\} \subseteq \mathsf{mint}\ tx$. By rule 9,

$$[\![\mathsf{msgsTT}]\!]\ (\star,\ (tx,\ \mathsf{msgsTT}))$$

We show that all the constraints of the MSGS specification are satisfied by $(\star, \pi\ u, tx, \pi\ u')$. Let *sndMsgs*, *rcvMsgs*, *newOuts*, *usedInputs* be defined as in the script msgsTT in Fig. 3, which, for the given transaction *tx*, is the same as in the MSGS transition rule in Fig. 5. Therefore, the let bindings (1), (3), (5) are the same for both the MSGS transition and in msgsTT. By inspection, the constraints (4) and (7) of the MSGS transition rule 5 are satisfied by definition of msgsTT, as they are replicated exactly in msgsTT.

The constraint (6) is similar to the following constraint in msgsTT,

$$\forall\ (i,\ msg)\ \in\ usedInputs, (msg,\ (\_, msg.\mathsf{msgTo}, \_))\ \in\ rcvMsgs$$

but it does not include the check (present in (6)) that for the given $(i, msg)$, $msg \in \pi\ u$. Input $i$ of $tx$ contains msgTkn *msg*, and by rule 4, all inputs of a transaction refer to unspent outputs, outputRef $i \mapsto$ output $i \in u$. Therefore, $msg \in \pi\ u$.

The constraint (2) is similar to the following constraint in msgsTT,

$$[\ \mathsf{getMsgRef}\ m\ \mid\ (\_,\ m)\ \leftarrow\ newOuts\ ]\ \#\ [\ \mathsf{getMsgRef}\ m\ \mid\ (\_,\ m)\ \leftarrow\ usedInputs\ ]$$

but msgsTT (unlike MSGS) does not require that; additionally,

$$[\ \mathsf{getMsgRef}\ m\ \mid\ (\_,\ m)\ \leftarrow\ newOuts\ ]\ \#\ [\ \mathsf{getMsgRef}\ m\ \mid\ m\ \leftarrow\ \pi\ u\ ]$$

This states that the fields *msg*.inUTxO and *msg*.msgIx of a message in *newOuts* cannot both be the same as those fields of any message already in $\pi\ u$. By msgOutsOK $u$, *msg*.inUTxO of a $msg \in \pi\ u$ cannot also be an output reference in $u$, so no messages in *newOuts* can be generated by spending the same output reference as was spent to generate an existing message in $u$. Also by msgOutsOK $u$, no messages already existing in $u$ could be duplicated. We have just shown that the MSGS constraints are satisfied, and can now use (2), (4), (6), and (7) in the rest of the proof.

Next, we show that $\pi\ u' \neq \star$, i.e. msgOutsOK $u'$. We note that because msgOutsOK $u$, and by $(\star, u, tx, u') \in$ LEDGER, the second conjunct of msgOutsOK $u'$ must hold. By the second conjunct of the assumption in Section 4, $tx$ does not add entries to the UTxO whose output reference values are also the inUTxO values of messages it mints. By the first conjunct of the assumption, $tx$ does not add output references to the $u$ that are the same as inUTxO values of existing message tokens in $u$ either. So, all messages msgTkn *msg* in $u'$ are such that $(msg.\mathsf{inUTxO} \mapsto \_ \notin u')$.

The constraint (2) ensures non-duplication of messages in $\pi\ u'$. All message token-containing UTxOs that are in $u'$ but not in outputs of $tx$ still satisfy the first conjunct of msgOutsOK $u'$, since they remain in the same outputs in $u$ as in $u'$ (by POV and the msgsVal constraint that all message tokens in inputs of $tx$ must be burned). All message tokens in the inputs of $tx$ are burned, and all message tokens in the outputs of $tx$ are minted according to policy msgsTT, and therefore also satisfy the first conjunct of msgOutsOK $u'$. It follows that msgOutsOK $u'$ holds, and $\pi\ u' \neq \star$.

Now, we must show that

$$\pi\ u' = (\pi\ u \setminus [\ m\ \mid\ (\_,\ m)\ \leftarrow\ usedInputs\ ]) \cup [\ m \mid (\_,\ m) \leftarrow newOuts\ ]$$

Let $msg \in \pi\ u'$. We can conclude that $msg \notin [\ m \mid (\_,\ m) \leftarrow usedInputs\ ]$ because by (7) in MSGS, and all message tokens in *usedInputs* must be burned. Since message tokens are unique in the UTxO set by (2) in Fig. 5, *msg* is therefore not in $\pi\ u'$. By POV and LEDGER, either msgTkn *msg* is in $u$ or it is in outputs added by $tx$. If msgTkn *msg* is in $u$, we are done. If msgTkn *msg* is in the outputs added by $tx$, By (4), (7) and the POV, the set $[\ m \mid (\_,\ m) \leftarrow newOuts\ ]$ contains all the message tokens that are in outputs of $tx$. We conclude,

$$msg \in (\pi\ u \setminus [\ m \mid (\_,\ m) \leftarrow usedInputs\ ]) \cup [\ m \mid (\_,\ m) \leftarrow newOuts\ ]$$

To show the inclusion in the other direction, suppose

$$msg \in (\pi\ u \setminus [\ m \mid (\_,\ m) \leftarrow usedInputs\ ]) \cup [\ m \mid (\_,\ m) \leftarrow newOuts\ ]$$

If $msg \in [\ m \mid (\_,\ m) \leftarrow newOuts\ ]$, by definition of *newOuts*, the message token msgTkn *msg* exists in the outputs of $tx$, and therefore in $\pi\ u'$ (by LEDGER). If msgTkn *msg* is not in the inputs of $tx$, by definition of *usedInputs*, it must also not be in $[\ m \mid (\_,\ m) \leftarrow usedInputs\ ]$. If msgTkn *msg* is in $\pi\ u$, by POV, the fact that msgTkn *msg* is not in the inputs of $tx$, and by definition of LEDGER, this token must remain on the ledger after $tx$ is applied, i.e. in $\pi\ u'$. So, *msg* is either in *newOuts*, or already in $\pi\ u$ but not in *usedInputs*, and we are done.

*Proof sketch of verified input-output pairs lemma.* Let $(s,\ u,\ tx, u') \in \mathsf{LEDGER}$, $\pi\ u \neq \star$ and let

$$inp := (i,\ (\mathsf{useMyFunction}, v, d),\ r) \in tx.\mathsf{inputs}$$

such that

$$\begin{aligned}[(\mathsf{receive},\ m)] &= r \\ (fIn,\ fOut) &= m.\mathsf{msgData} \\ m.\mathsf{msgFrom} &= (\mathsf{checkMyFunction}, \_, \_)\end{aligned}$$

From the validity of the step $(s,\ u,\ tx, u')$ and the definition of $inp$, we can conclude that

$$[\![\mathsf{useMyFunction}]\!]\ (d,\ r,\ (tx,\ inp))$$

so, by definition of the useMyFunction script,

$$(-1)\ *\ (\mathsf{msgTkn}\ m)\ \subseteq\ tx.\mathsf{mint}$$

Now, all quantities of message tokens in outputs on the ledger are exactly 1, and no message tokens are duplicated. This is implied by $\pi\ u \neq \star$, which calls msgOutsOK to check this. By the POV rule 5, we can conclude that one UTxO entry $p \mapsto w \in u$ containing one token msgTkn $m$ was spent by $tx$ from $u$. We know it is exactly one because, inspecting msgOutsOK, message tokens are unique on a ledger for which $\pi\ u \neq \star$.

Again, by inspecting msgOutsOK, we can conclude that the token msgTkn $m$ in the value of the entry $p \mapsto w$ on the ledger $u$ must be such that it was minted by some previous transaction $p.\mathsf{id} \in \mathsf{Tx}$, and the following script validated :

$$[\![\mathsf{msgsTT}]\!]\ (\star,\ (p.\mathsf{id},\ \mathsf{msgsTT}))$$

By inspecting msgsTT, we see that $(w, m) \in newOuts$ by definition of $newOuts$, and the uniqueness of the message token $m$ (which is guaranteed by (2) in MSGS).

Therefore, by constraint (4) on $newOuts$ for transaction $p.\mathsf{id}$, some $(m, (q, g, r'))$ is contained in $sndMsgs$, and is such that

$$[(\mathsf{send},\ m)] = r'$$

with

$$g = m.\mathsf{msgFrom}\ =\ (\mathsf{checkMyFunction}, \_, \_)$$

By the last clause of msgOutsOK in Fig. 2, we have

$$[\![\mathsf{checkMyFunction}]\!]\ (\_,\ r',\ (p.\mathsf{id},\ (q, g, r')))$$

Since $(fIn,\ fOut)\ =\ \mathsf{msgData}\ m$, inspecting checkMyFunction, we get that

$$\mathsf{myFunction}\ fIn\ =\ fOut$$

Now, $tx$ must burn the message $m$, since $(i,\ (\mathsf{useMyFunction}, v, d),\ r) \in tx.\mathsf{inputs}$, and useMyFunction requires that a message token encoding $m$ is burned. So, the message minting policy must validate,

$$[\![\mathsf{msgsTT}]\!]\ (\star,\ (tx,\ \mathsf{msgsTT}))$$

By definition of msgsTT, $(w, m) \in usedInputs$ for $tx$ must be such that $(m,\ (\_, m.\mathsf{msgTo}, \_)) \in rcvMsgs$. Therefore, the transaction spends the output $(\_, m.\mathsf{msgTo}, \_)$ with redeemer $[(\mathsf{receive}, m)]$. Inspecting useMyFunction, we see that for redeemer $[(\mathsf{receive},\ m)]$ and input $(i,\ (\mathsf{useMyFunction}, v, d),\ r)$, it requires the minting of message $m$ with $m.\mathsf{msgTo} = (\mathsf{useMyFunction}, v, d)$.

BASIC TYPES

| | |
|---|---|
| $\mathbb{B}, \mathbb{N}, \mathbb{Z}$ | *the type of Booleans, natural numbers, and integers* |
| $\mathbb{H}$ | *the type of bytestrings:* $\bigcup_{n=0}^{\infty}\{0,1\}^{8n}$ |
| $\mathsf{Interval}[A]$ | *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* |
| $\mathsf{TokenName} = [\mathsf{Data}]$ | *token name* |
| $\mathsf{AssetID} = \mathsf{Policy} \times \mathsf{TokenName}$ | *unique asset identifier* |
| $\mathsf{Slot}$ | *slot number representing chain time* |
| $\mathsf{Data}$ | *a type of structured data* |
| $\mathsf{Script}$ | *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

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

Fig. 8: Basic definitions of the $\mathrm{EUTxO}_{\mathrm{MA}}$ model

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

Fig. 9: Validity of a transaction $t$ in the $\mathrm{EUTxO}_{\mathrm{MA}}$ model