Message-passing and the Double Satisfaction Problem in the EUTxO Ledger Model *

Polina Vinogradova[1] [0000−0003−3271−3841]

IOG, firstname.lastname@iohk.io

Abstract. We formalize communication via message-passing among scripts and stateful contracts in the extended UTxO model. Our formalization is made up of a specification and an implementation of a message-passing stateful smart contract using the structured contracts framework. Messages are recorded on the ledger in the form of special NFT tokens, distributed one per UTxO entry. This entry also contains the assets the message is sending. We formalize the notion of sender and receiver outputs for messages. A message is sent (produced) and received (consumed) asynchronously, in separate transactions, with the possibility of multiple messages between multiple outputs to be produced and consumed within a single transaction.
We give two applications of our design. The first is using a message as a record of a successful script computation, similar to memoization. The second application is as an mechanism for asynchronous communication within a structured contract, as well as externally, such as in the case of making payouts to an address, separating contract communication from computation. We also formalize the well-known double satisfaction problem for the EUTxO ledger, which has not been done before. We then show how message-passing can solve the double satisfaction problem for payouts.

Keywords: blockchain · ledger · smart contract · formal verification · specification · transition system · UTxO · message-passing

1 Introduction

Message-passing is the standard for communicating data and assets between contracts in account-based ledgers [8] [34] [17]. Both sending and receiving a message is performed atomically as part of processing a transaction containing that message. In some cases, the transaction itself may be referred to as a message [17]. Communication between scripts in UTxO-based ledgers follows a different architecture. Smart contract enabled UTxO models such as Cardano [19], Algorand [1], and Ergo [13] use scripts only to *check* that certain changes being made by a transaction to the ledger state are allowed. A script may require that another script executes successfully given certain arguments, and within the same transaction. Script interaction and communication is implemented via such constraints on other scripts, which we refer to as *dependencies*.

In this work we present an EUTxO layer-2 implementation of *asynchronous message-passing* as an alternative method of communication between individual scripts as well as the stateful contracts they implement (Section 4). We build this infrastructure on top of the extended UTxO ledger presented in [11], and formalized in terms of the small-step semantics formalism [30] also used to specify the Cardano EUTxO-based ledger [19]. The stateful message-passing contract is an instantiation of the structured contract framework [35]. This framework includes the semantics for specifying and implementing a stateful contract, together with a proof obligation that the specification meets the implementation, formalized in Agda [21].

The state of the message-passing contract is a set of messages. This state is represented on the ledger as a collection special NFT tokens storing message data, which are distributed across separate UTxO entries. Messages specify the output that was spent to validate their production. Thus, they serve as proof artefacts of the validation of specific output-locking contracts, with specific inputs. This gives messages a verified *sender*. Similarly, message tokens can only be spent and burned by an authenticated *recipient*, another contract that must validate with particular inputs when consuming a message. Any script is able to interface with the message-passing contract so long as the user input to the script can be decoded as a list of messages being produced and consumed.

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

The message-passing structured contract we propose addresses some of the challenges of programming on an EUTxO ledger. It does so by defining a way to record authentically-sourced data and assets on an EUTxO ledger, for consumption by a specific recipient, in a highly concurrent way. In addition to providing a way to interpret the notion of message-passing on an EUTxO ledger, our distributed stateful contract design demonstrates the flexibility of the structured contract formalism for implementation of contracts with state contained in multiple UTxO entries. Unlike a centralized design with a single UTxO entry keeping track of all messages, our design allows arbitrary reordering of message-passing transactions whenever one of them does not consume messages (or other outputs) produced by the other. We demonstrate that this scheme can be used in the following ways, addressing the corresponding challenges of EUTxO programming :

(i) as a authenticated output of a function call with specific inputs to the function, similar to the technique called memoization (Section 5.1) ;
(ii) as intermediate storage of assets and data for the purpose of asynchronous communication between stateful contracts (Section 5.2);

Next (Section 6), we present a formalization of an as-of-yet unformalized, but widely discussed problem in EUTxO programming : the *double satisfaction problem* (DSP) [18] [32] [4] [28]. DSP may occur when multiple scripts within the same transaction share the same constraint, which is satisfied. For example, by making a *single* payout to an address as required by two separate contracts, even though the intent of each of the contracts was that the address should have received a separate payment, for a total of two. Using the structured contract formalism, we formalize when a constraint of a contract is vulnerable to such a situation. We then demonstrate that making payouts via messages is not susceptible to double satisfaction.

The main contributions of this work are :

(i) the specification and implementation of a message-passing structured contract ;
(ii) two applications of the message-passing contract : messages as outputs of function calls, and messages-passing as a means of asynchronous communication of data and transfer of assets between contracts and within a contract ;
(iii) formalization of the double satisfaction problem ;
(iv) a result stating that if payouts are specified via changes in the state of the message-passing contract, the specification of the required payouts is not susceptible to DS

2 Small-steps Specifications and the Ledger Model

In order to give the specification of the message-passing stateful contract, we first discuss the semantics we use for our specifications, as well as the ledger model underlying our contract implementations.

2.1 Small-steps Specifications

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 [30], and used in [19] [35]. Non-standard notation used here is given in 4.

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 [19].

2.2 Ledger Types

The ledger model we use is an extended UTxO model with native multi-assets, as introduced in [35]. It is based on a previously introduced model [9]. We give an overview of the EUTxO state and transaction structure below, and present the full set of type definitions in Figures 5 and 6.

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 [10]. 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 [9], 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 [20]. It is also used in [9], and in the Cardano implementation [31]. 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 7, and consistent with the ledger rules specified in [9] [35] :

(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 6.

3 The Structured Contract Formalism

The structured contract framework is a formalism for specifying and demonstrating the correctness of the implementation of a stateful contract on LEDGER. A structured contract is specified in terms of small-steps semantics, then a ledger reprsentation is given for its state and input types. The ledger representation consists of a pair of projection functions : (i) one which computes the contract state from a given UTxO state (or fails), and (ii) one which computes the input to the contract from the transaction being applied to the ledger. The environment (slot number) is ignored because scripts are not permitted to inspect it in our model.

For any step in the LEDGER, the representation functions must compute a corresponding valid step in the structured contract any time the starting UTxO state maps to a contract state (rather than fails). This design guarantees that no invalid contract state updates are ever possible on the ledger. Note that it is not necessarily true that for any contract step, a corresponding valid ledger step exists. Specifying when one does exist is the subject of future work.

Definition. Suppose STRUC is small-step transition systems of the following type :

$$\_ \vdash \_ \xrightarrow[\text{STRUC}]{\_} \_ \in \mathbb{P}\ (\star \times \mathsf{State}_{\mathsf{STRUC}} \times \mathsf{Input}_{\mathsf{STRUC}} \times \mathsf{State}_{\mathsf{STRUC}})$$

and let

$$\pi_{\mathsf{STRUC}} : \mathsf{UTxO} \to \mathsf{State}_{\mathsf{STRUC}} \cup \{\star\}$$
$$\pi_{\mathsf{Tx,STRUC}} : \mathsf{Tx} \to \mathsf{Input}_{\mathsf{STRUC}}$$

be some projection functions such that the following implication holds :

$$\sim> \frac{e \vdash (u) \xrightarrow[\text{LEDGER}]{t} (u')}{* \vdash (\pi_{\mathsf{STRUC}}\ u) \xrightarrow[\text{STRUC}]{\pi_{\mathsf{Tx,STRUC}}\ t} (\pi_{\mathsf{STRUC}}\ u')} \tag{2}$$

The triple $(\mathsf{STRUC}, \pi_{\mathsf{STRUC}}, \pi_{\mathsf{Tx,STRUC}})$ is called a *structured contract*, and we denote it by

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

Note that the range of the function π_{STRUC} is $\mathsf{State}_{\mathsf{STRUC}} \cup \{\star\}$, where $\{\star\}$ is a singleton. This indicates that π_{STRUC} is *partial*. Note also that the block-level data is never exposed to user-defined scripts in this model, so that the context of a structured contract is necessarily of type $\{\star\}$.

4 Message-passing specification

Conceptually, a message is data associated with a specific sender and recipient, that has been sent from the sender to the recipient [3]. In our design, a message is a data structure of type Msg encoded on the ledger, which also includes a sender, receiver, and some data. The content of $m \in \mathsf{Msg}$ is encoded as the TokenName of an NFT with the minting policy msgsTT. Such an NFT can only be minted and burned under the certain conditions on the carrying transaction. A message consists of the following fields, see Figure 8

(i) an output reference inUTxO : OutputRef : must be spent when the message token is minted ;
(ii) an index msgIx : Ix : used to uniquely identify a message whenever multiple messages are produced in association with spending a single UTxO entry ;
(iii) an output msgTo : Output : 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 msgFrom : Output : an output that must be spent to validate production of the message, the entry inUTxO $\mapsto$ msgFrom must be in the UTxO set at the time the message is minted, and must be spent to validate the production of the message *by that sender* ;
(v) a value msgValue : the assets being sent from the sender output to the recipient via this message. When a message is minted and placed in an output, this output must *also* contain at least this amount of assets ;
(vi) data msgData : the data being sent via this message, generated using known inputs to the sender script ;

Each message must have a unique identifier to keep track of it. Here, we use the approach to ensuring NFT uniqueness by recording in each message token a particular output reference inUTxO spent from the UTxO by the minting transaction, similar to the thread token mechanism in [9] [35]. A single script may produce multiple messages in the same transaction, associated with a single output reference. So, the identifying output reference inUTxO together with its *index* msgIx uniquely identifies the message.

Sending a list of messages is done by submitting a transaction that (i) mints the NFTs encoding each of the messages, and (ii) executes the script of the sender output with the redeemer containing the list of sent messages. For an output to receive a set of sent messages, a transaction must spend the outputs containing the messages, and burn the message tokens. It must also spend the receiver output supplied with a redeemer containing the message list.

The MSGS transition system specifies the rules for updating a set of messages with a given transaction by adding and removing messages to/from the message state. It has a state type $\mathsf{State} := \mathbb{P}\ \mathsf{Msg}$, and an input type $\mathsf{Input} := \mathsf{Tx}$. The projection function π_{Msg} returns, for a given *utxo*, the messages encoded by the message tokens that exist in the outputs of the UTxO set. It returns $\star$ when one or more messages have been duplicated or incorrectly generated in the *utxo*.

The function msgTkn constructs a message token out of a given message. See Figure 10 for details. Figure 1 gives the transition rule for the MSGS system. The functions fromData, toData, whose types are given in 9, encode and decode data at lists of messages to be sent and received.

$$\text{– (1) compute messages being sent and received by getting all redeemers}$$
$$\text{which decode as pairs of an instruction and message data}$$
$$\mathit{sndMsgs} := [\ (\mathit{msg},\ i)\ \mid\ i \leftarrow\ (\mathsf{toList}\ (\mathsf{inputs}\ \mathit{tx})),\ (\mathit{sr},\ \mathit{msg})\ \leftarrow\ \mathsf{fromData}\ (\mathsf{redeemer}\ i),\ \mathit{sr} = \mathsf{send}\ ]$$
$$\mathit{rcvMsgs} := [\ (\mathit{msg},\ i)\ \mid\ i \leftarrow\ (\mathsf{toList}\ (\mathsf{inputs}\ \mathit{tx})),\ (\mathit{sr},\ \mathit{msg})\ \leftarrow\ \mathsf{fromData}\ (\mathsf{redeemer}\ i),\ \mathit{sr} = \mathsf{receive}\ ]$$

$$\text{– (2) check that no messages new are duplicates}$$
$$\mathsf{noDups}\ (\mathsf{map}\ \mathsf{getMsgRef}\ \mathit{sndMsgs}\ +\!+\ \mathsf{map}\ \mathsf{getMsgRef}\ \mathit{rcvMsgs})$$
$$\mathsf{noDups}\ (\mathsf{map}\ \mathsf{getMsgRef}\ \mathit{sndMsgs}\ +\!+\ \mathsf{map}\ \mathsf{inUTxO}\ \mathit{msgs})$$

$$\text{– (3) compute what new message-containing outputs should be created}$$
$$\text{based on what message tokens are in tx outputs}$$
$$\mathit{newOuts}\ :=\ \{\ (o,\ \mathit{msg})\ \mid\ o\ \in\ \mathsf{outputs}\ \mathit{tx},\ \mathsf{msgTkn}\ \mathit{msg}\ \subseteq\ \mathsf{value}\ o\ \}$$

$$\text{– (4) check that all the messages are correctly defined :}$$
$$\text{correct sender, outputref is spent, one message per output,}$$
$$\text{output with message has correct validator and suffient value, sender has correct redeemer}$$
$$\forall\ (o,\ \mathit{msg})\ \in\ \mathit{newOuts},\ (\mathit{msg},\ (\mathit{inf}, \mathsf{msgFrom}\ \mathit{msg}, \_))\ \in\ \mathit{sndMsgs}$$
$$\wedge\ \mathsf{inUTxO}\ \mathit{msg}\ =\ \mathit{inf}\ \wedge\ \{\ t\ \subseteq\ \mathsf{value}\ o\ \mid\ \mathsf{dom}\ t\ =\ \{\mathsf{msgsTT}\}\ \}\ =\ \mathsf{msgTkn}\ \mathit{msg}$$
$$\wedge\ \mathsf{validator}\ o\ =\ \mathsf{msgsVal}\ \wedge\ \mathsf{value}\ o\ \geq\ \mathsf{msgValue}\ \mathit{msg}$$

$$\text{– (5) compute what message-outputs should be spent}$$
$$\text{based on what message tokens are in tx inputs}$$
$$\mathit{usedInputs} := \{\ (i,\ \mathit{msg})\ \mid\ i\ \in\ \mathsf{inputs}\ \mathit{tx},\ \mathsf{msgTkn}\ \mathit{msg}\ \subseteq\ \mathsf{value}\ (\mathsf{output}\ i)\ \}$$

$$\text{– (6) check that all the messages are correctly consumed :}$$
$$\text{the receiver is correct, and has correct redeemer, and message exists}$$
$$\forall\ (i,\ \mathit{msg})\ \in\ \mathit{usedInputs},\ (\mathit{msg},\ (\_, \mathsf{msgTo}\ \mathit{msg}, \_))\ \in\ \mathit{rcvMsgs}\ \wedge\ \mathit{msg} \in \mathit{msgs}$$

$$\text{– (7) check minting and burning of message tokens :}$$
$$\Sigma_{(\mathit{msg},\ \_)\ \in\ \mathit{sndMsgs}}\ \mathsf{msgTkn}\ \mathit{msg}\ +\ \Sigma_{(\mathit{msg},\ \_)\ \in\ \mathit{rcvMsgs}}\ (-1)\ *\ (\mathsf{msgTkn}\ \mathit{msg})$$
$$=\ \Sigma_{\mathsf{msgsTT} \mapsto \mathit{tkns}\ \in\ \mathsf{mint}\ \mathit{tx}} \mathsf{msgsTT} \mapsto \mathit{tkns}$$

$$\text{Process}\ \frac{}{\vdash\ (\ \mathit{msgs}\ )\ \xrightarrow[\text{MSGS}]{\mathit{tx}}\ (\ (\mathit{msgs} \setminus (\mathbf{map\ fst}\ \mathit{rcvMsgs})) \cup (\mathbf{map\ fst}\ \mathit{sndMsgs})\ )} \quad (3)$$

Fig. 1: Specification of the MSGS transition

The contracts msgsTT and msgsVal, implementing the above specification, are given in Figures 11 and 12. The message minting policy msgsTT performs all the same checks that are in the MSGS specification. Each message token minted by an transaction under policy msgsTT must be placed into a UTxO locked by a special validator, msgsVal. The script msgsVal checks that the message token therein is burned — the minting (i.e. burning) policy ensures the rest of the conditions specified in MSGS. The redeemer type for the validator

script msgsVal is $\{\star\}$. See A for the proof of the relation $\sim>$ between LEDGER and MSGS that ensures that the implementation of MSGS via the msgsTT and msgsVal scripts is according to the MSGS specification. It uses assumption (i) in 4.

Extra assumptions. We must make additional assumptions about allowable ledger states and transactions in order to prove properties of the behaviour of our MSGS program. Under reasonable assumptions about the initial state of the ledger, the following assumptions may not be required, and instead be proved as safety properties [2] resulting from correct construction of subsequent ledger states starting from acceptable initial states. A full treatment of traces and properties is outside the scope of this work.

(i) **Replay protection.** For any *utxo* such that $\pi\ utxo \neq \star$,

$$\forall\ m, m' \in \pi\ utxo,\ (\mathsf{msgIx}\ m = \mathsf{msgIx}\ m') \ \Rightarrow\ \mathsf{fst}\ (\mathsf{inUTxO}\ m) \neq \mathsf{fst}\ (\mathsf{inUTxO}\ m')$$

This states that if two message tokens on the ledger *utxo* have the same index, they could not have been generated as a result of spending the same UTxO entry. This assumption required in order to guarantee non-duplication of message tokens in the UTxO set. Under reasonable assumptions about the initial ledger state, it is a consequence of replay protection together with non-duplication of messages within a single transaction, which is guaranteed by first conjunct of (2) in the rules 1. Replay protection states that the same transaction cannot be applied twice within the same trace. In the EUTxO model, as demonstrated in [9] [20], replay protection is a safety property satisfied by any ledger trace starting from a state satisfying certain properties.

(ii) **Script validation.** Given any *utxo*, $(tx, \_) \mapsto o\ \in\ utxo$,

$$[\![\mathsf{msgsVal}]\!]\ (d,\ r,\ (tx,\ i))$$

where $\mathsf{redeemer}\ i = r, \mathsf{datum}\ (\mathsf{output}\ i)\ =\ d$. This states that the scripts in all the outputs of all transactions recorded on a given ledger state must have validated with their corresponding datum and redeemer inputs. We will require this assumption when demonstrating the applications of message-passing in order to treat existing messages as generated in conjunction with the validation of a sender output.

5 Message-passing usecases

In this section we give two applications of the message-passing 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 smaller functions, each requiring less memory or CPU to run. A script may not trust values pre-computed off-chain, so a proof that a value was correctly computed on-chain is required. The technique we describe in this section for constructing such proofs is similar to a specific kind of caching, also called *memoization* [15], which is also how we refer to our approach.

Consider a function $\mathsf{myFunction}\ :\ \mathsf{MyInType}\ \rightarrow\ \mathsf{MyOutType}$ which performs some computation. We define a script checkMyFunction 2, that wraps the computation done by myFunction into a script that produces a message with data $(fIn,\ fOut)$, such that $\mathsf{myFunction}\ fIn\ =\ fOut)$, and mints a message token proving that it is correct :

Note that msgTo is not constrained by this contract, so that the generated message can be sent to any recipient. The following function decodes the message data as an input-output pair, or fails :

$$\mathsf{fromData}_{IO} : \mathsf{Data} \rightarrow (\mathsf{MyInType}\ \times\ \mathsf{MyOutType}) \cup \{\star\}$$

We define a script useMyFunction, Figure 3, that can consume a message with the redeemer $(\mathsf{receive},\ \mathsf{Msg}\ m)$ when it is addressed to the output locked by useMyFunction. This message serves as a proof that myFunction $(fIn, fOut)$, so, useMyFunction can do some computation checkStuff relying on the fact that myFunction $fIn = fOut$.

We give the result that formalizes the use of message-passing here as proof that myFunction $fIn = fOut$.

```
⟦checkMyFunction⟧ (_, r, (tx, i)) :=
                                        – If r is a send-message redeemer sending m
                                        –and message data is an input-output pair
                                    if fromData r ≠ ⋆ ∧ fromData_IO (msgData m) ≠ ⋆,
                                        – m is uniquely identified by i
                                        ∧ inUTxO m = outputRef i
                                        – m is from this script
                                        ∧ msgFrom m = output i
                                        – no minimum sent value
                                        ∧ msgValue m = 0
                                        – message token with message m is minted
                                        ∧ msgTkn m ⊆ mint tx
                                        – check myFunction computation
                                        ∧ myFunction fIn = fOut
                                         where
                                           [(send, Msg m)] = fromData r
                                           (fIn, fOut) = fromData_IO (msgData m)
                                    else,
                                        False
```

Fig. 2: Script checking input-output pairs for myFunction

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

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

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

5.2 Contracts using message-passing

Stateful contract interaction, or communication, in the EUTxO model is usually implemented via dependencies [9]. A (direct) *dependency* of a script c is a constraint requiring that another script c' must be executed within the same transaction, possibly with specific arguments. *Message-passing* is implemented via dependencies, and is an alternative to the direct dependency approach to implementing interaction and communication of other contracts or scripts. It allows the interacting scripts execute asynchronously, and to depend on the message-passing scripts msgsTT and msgsVal rather than on each other directly. We say that stateful contracts that require the production or consumption of messages to or from scripts implementing the contract *use message-passing*, and we formalize this notion in this section.

Messages serve as intermediate stores of assets whose transfer is authorized by the sender, but before they are accepted by the receiver. The receiver output script may or may not place additional constraints on how the value and data in the message is used. Message-passing is a robust way of making payouts, which are asset transfers *from* a specific payer *to* receiver outputs not included in the structured contract state. We give an example to demonstrate this.

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

```
⟦useMyFunction⟧ (d, r, (tx, i)) :=
    – If r is a receive-message redeemer sending m
    – and m-data from message is an input-output pair
  if fromData r ≠ ⋆ ∧ fromData_IO (msgData m) ≠ ⋆,
    – m is from the script computing myFunction
    ∧ msgFrom m = checkMyFunction
    – m is sent to this script
    ∧ msgTo m = output i
    – message token with message m is burned
    ∧ (−1) ∗ (msgTkn m) ⊆ mint tx
    – use myFunction computation output from message to do more stuff
    ∧ checkStuff d r (tx, i) (fIn, fOut)
      where
        [(receive, Msg m)] = fromData r
        (fIn, fOut) = fromData_IO (msgData m)
  else,
    checkOtherStuff d r (tx, i)
```

Fig. 3: Script that uses memoized output of myFunction

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 must be the complete transaction. Below, we sometimes abuse notation for predicates $\mathsf{P} : A \to \mathbb{B}$, and write $a \in \mathsf{P}$ for $\mathsf{P}\ a$.

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

$$\mathsf{State} := \{i \mapsto o \ \in\ U \mid i \in \mathsf{Input},\ o \in \mathsf{F}, U \in \mathsf{UTxO}, \mathsf{c}\ U\}$$

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

$$\pi_{\mathsf{Tx}} := \mathsf{id}$$

We can use the technique of combining structured contracts [35], to construct the structured contract $\mathsf{STRUC}_{\mathsf{MSGS}}$,

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

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

$$\begin{aligned} \mathsf{getFromSTRUCmsgs} &: \mathbb{P}\ \mathsf{Msg} \to \mathbb{P}\ \mathsf{Msg} \\ \mathsf{getFromSTRUCmsgs}\ msgs &:= \{\ m \mid m \in msgs,\ \mathsf{F}\ (\mathsf{msgFrom}\ m)\ \} \end{aligned}$$

$$\begin{aligned} \mathsf{getToSTRUCmsgs} &: \mathbb{P}\ \mathsf{Msg} \to \mathbb{P}\ \mathsf{Msg} \\ \mathsf{getToSTRUCmsgs}\ msgs &:= \{\ m \mid m \in msgs,\ \mathsf{F}\ (\mathsf{msgTo}\ m)\ \} \end{aligned}$$

We now state a result that says that, for given a F, 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* a specific 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 *to* a specific contract requires, again, 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 $(\star, (s, m), tx, (s'm')) \in \mathsf{STRUC}_{\mathsf{MSGS}}$,

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

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

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

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

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 computations that STRUC performs may be contingent on receiving a specific message. For example, accepting a payment message sent by another contract.

Contracts that use message-passing share two common features that are both necessary and sufficient for a script c implementing the contract 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 require that the corresponding messages are minted/burned. The sent messages must be *from* c, and the received messages must be addressed *to* c, by Lemma 5.2.

We can refer to the messages sent and received by outputs that make up the state of a contract STRUC, i.e. those filtered by F, as a scripts' *communication*. All other checks and state updates are the contract's computations. These computations may still include dependencies on scripts implementing contracts other than MSGS. Specifying when a contract has no non-message dependencies is important for reasoning about when it is able to progress. This is, however, the subject of future work.

A *payout* is a message that is from STRUC, but not addressed to STRUC, and specifies a sent value greater than zero. When a contract makes payouts with messages, it generates at least one payout for some valid step. Note the function that returns all the payouts for a given contract, getPayouts, is a function of the MSGS state only, given that it is applied to a valid step of $\mathsf{STRUC}_{\mathsf{MSGS}}$. To demonstrate using message-passing, we give an example of a message-augmented contract that makes payouts until it runs out of funds. We focus on a payout example because we later present a security result about message payouts.

Message-augmented PAYOUT Suppose NFT is an NFT thread token whose minting policy requires that a specific output reference must be spent from the UTxO, and the token must be placed into the output given by $(\mathsf{payout}, \mathsf{NFT} + \mathsf{a}, \star)$, for some $\mathsf{a} > 0$. This NFT token serves as a unique identifier of the UTxO entry currently containing the datum and value representing part (or all) of the contract state. This approach is defined in the

thread token design pattern in [9], and used in [35], as well as for implementing MSGS. We define the filter F which returns only the outputs containing the NFT specific to the PAYOUT contract implementation, and constraint c stating that there is exactly one NFT in the UTxO :

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

$$\mathsf{c}\ u \ := \ \exists!\ i \mapsto o \in \ u, \mathsf{NFT} \subseteq \mathsf{value}\ o$$

The redeemer type for the payout script is $[(\mathsf{SR} \times \mathsf{Msg})]$ (as with all message-passing scripts), 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. Transitions between distinct states with multiple copies of the NFT (e.g. including more than one UTxO) are never allowed.

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*. Defining the message-augmentation of any message-passing contract makes it possible to constrain the message state update, rather than filtering and constraining messages in the inputs and outputs of a transaction. This is because the function that returns all the payouts for a given contract, getPayouts, is a function of the MSGS state only.

The specification is given by

$$\text{MSGSOnly}\ \frac{\_ \vdash (m) \xrightarrow[\text{MSGS}]{tx} (\boldsymbol{m'}) \qquad (i, o, \_) \notin \mathsf{inputs}\ tx}{\vdash \begin{pmatrix} \{i \mapsto o\} \\ m \end{pmatrix} \xrightarrow[\text{PAYOUT-MSGS}]{tx} \begin{pmatrix} \{i \mapsto o\} \\ \boldsymbol{m'} \end{pmatrix}} \tag{4}$$

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

$$ms \in m' \qquad\qquad ms \notin m \qquad\qquad \mathsf{v} \leq a$$

$$(tx, ix) \mapsto (\mathsf{payout}, \mathsf{NFT} + (a \ - \ \mathsf{v}), \star) \in \mathsf{mkOuts}\ tx$$

$$\exists\ inp \in \mathsf{inputs}\ tx, \mathsf{outputRef}\ inp = i \ \wedge \ \mathsf{redeemer\ inp} = \{(\mathsf{send}, ms)\}$$

$$\text{PayoutV}\ \frac{\_ \vdash (m) \xrightarrow[\text{MSGS}]{tx} (\boldsymbol{m'})}{\vdash \begin{pmatrix} \{i \mapsto (\mathsf{payout}, \mathsf{NFT} + a, \star)\} \\ m \end{pmatrix} \xrightarrow[\text{PAYOUT-MSGS}]{tx} \begin{pmatrix} \boldsymbol{\{(tx, ix) \mapsto (\mathsf{payout}, \mathsf{NFT} + (a \ - \ \mathsf{v}), \star)\}} \\ \boldsymbol{m'} \end{pmatrix}} \tag{5}$$

6 Double Satisfaction

In the EUTxO model, multiple scripts can be executed as part of the validation and application of the transaction which causes them to be run, i.e. the *carrying* transaction. Each script is run when the unique 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).

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

(i) **Authorization tokens :** the transaction must contain in its inputs a special token, the ability to spend which constitutes proof that a particular contract state update is authorized
(ii) **Payouts :** to perform a specified state update, the transaction must make a payout of value v to address a, i.e. contain an output with value v and address a

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 may be that the same transaction makes contract state updates to other contracts, whose implementations also require the check that the specific authorization token was presented. There is no problem with presenting a single authorization token within a transaction updating the state of multiple distinct contracts, each of which requires the presentation of this token. 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 $(a, v, \_)$. 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 one contract. In this case, the double satisfaction is problematic, and a receives less total assets than intended.

Because this issue is one that applies to all scripts that require transactions to make payouts, it is a widely encountered issue with EUTxO programming. It has been discussed in Plutus documentation [32], as well as in the context of contract audits [18] [4] [28], but not formally analyzed. In this work, we discuss the DSP in the context of structured contracts, and argue that expressing payouts as messages is a way to make payments that is resilient to DS.

The examples we presented show that the distinction between DS that is problematic and not problematic 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 a function that returns all pairs of states in valid transitions of a contract 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 STRUC is a subset

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

such that $\mathsf{STRUC} \subseteq \mathsf{C}$. A constraint is *strict* when $\mathsf{STRUC} \subsetneq \mathsf{C}$.

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

Discussion. Definition 6 states that a double satisfaction-vulnerable constraint limits the set of allowable inputs for a given pair of states with a transition between them. This is the kind of constraint that can appear in another contract run within the same transaction, intuitively making them susceptible to DS. A constraint on the pairs of states that have a transition between them, on the other hand, can only be satisfied by another contract if the two contracts *share state*. A constraint on shared contract state and the update thereof does not appear to be what is meant by a single constraint being satisfied by multiple contracts. It is, rather, a different situation, better described in terms of nested or overlapping contracts, and is the subject of future work.

Our definition of DS rules out constraints that can never be satisfied, and therefore all constraints that eliminate certain state transitions entirely. Requiring certain payouts to be in transaction outputs, but not recording them in the contract state, is vulnerable to double satisfaction. We give an example to illustrate the concept of DS vulnerability.

*Example (*TOGGLE *with extra constraint).* Consider the following specification of a TOGGLE contract, with State $= \mathbb{B}$, and Input $= (\mathsf{toggle} \cup \{\star\}) \times \mathsf{Interval}[\mathsf{Slot}]$.

$$\text{DoNothing}\ \frac{}{\vdash (x) \xrightarrow[\text{TOGGLE}]{(\star,\_)} (x)} \tag{6}$$

$$\text{Toggle}\ \frac{5 \leq k < j \leq 9}{\vdash (x) \xrightarrow[\text{TOGGLE}]{(toggle,[j,k])} (\neg\, x)} \tag{7}$$

Now, let us observe the DS vulnerability that occurs in this contract. We define a contract STRUC′ by removing $j \geq 5$ from 7, and define the strict constraint

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

so, STRUC $\subsetneq$ C. The contract STRUC′ has the same set of possible transitions as STRUC, which is : $x \mapsto x$ and $x \mapsto \neg\, x$, and therefore, s STRUC = s STRUC′. So, by our definitions, STRUC = STRUC′ $\cap$ C $\subsetneq$ STRUC′. Therefore, STRUC is vulnerable to DS with respect to 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 $[5, 9]$. This may be difficult to avoid, in practice.

It is possible to define a class of contracts that is never vulnerable to DS constraints, and these contracts have an output state for any pair of an input state and an input :

Lemma (DS-free contracts) A system STRUC is not vulnerable to double satisfaction with respect to any constraint whenever for any (s, i), there exists an s' such that $(\star, s, i, s') \in$ STRUC

We give a proof sketch in A.

MSGS *as payouts.* Payouts may be implemented differently for different structured contracts. We gave one naive approach to payouts in (ii) in 6. This approach is vulnerable to DS, since the constraint requiring a payout to be made is strictly on the input transaction, with no record of the payout in the state. We now argue that, since making payouts via messages can be expressed as a constraint on a pair of message states, rather than on the input transaction, this approach to payouts is not vulnerable to DS for a message-enhanced contract.

Let us consider payouts as specified in the definition of payouts via messages, given in 5.2. Suppose $(\pi_{\mathsf{F},\mathsf{c}},\ \pi_{\mathsf{Tx}},\ \mathsf{STRUC})$. Let $(\pi_{\mathsf{F},\mathsf{c}},\ \pi_{\mathsf{Tx}},\ \mathsf{STRUC}')$ be another structured contract, with s STRUC = s STRUC′, and STRUC $\subseteq$ STRUC′. The two contracts necessarily have the same sets of payout messages for transitions between the same pairs of states.

To formalize this reasoning, we define following constraint on the message-enhanced contract $\mathsf{STRUC}_{\mathsf{MSGS}}$,

$$\begin{aligned}\mathsf{Pays}\ (\star, (s,m), tx, (s'm')) \ :=\ &\forall\, \mathsf{STRUC}' \supseteq \mathsf{STRUC}, \mathsf{s\ STRUC} = \mathsf{s\ STRUC}', \forall (\star, (s,m), tx', (s', m')) \in \mathsf{STRUC}', \\ &\mathsf{getPayouts}_{\mathsf{STRUC}}\ (\star, (s,m), tx, (s', m')) = \mathsf{getPayouts}_{\mathsf{STRUC}'}\ (\star, (s,m), tx', (s', m'))\end{aligned}$$

This constraint states that given any other (more permissive) contract STRUC′, the set of payouts necessary and sufficient for any step of both $\mathsf{STRUC}_{\mathsf{MSGS}}$ and $\mathsf{STRUC}'_{\mathsf{MSGS}}$ is fully determined by the change in the corresponding MSGS state, even when the input transactions are distinct. By definition, $\mathsf{getPayouts}_{\mathsf{STRUC}}$ and $\mathsf{getPayouts}_{\mathsf{STRUC}'}$ do not inspect the states s, s' or the transactions tx, tx' when applied to valid steps in STRUC and STRUC′, respectively, see 5.2. So, we can conclude that the following is trivially true :

$$\forall\, step, \mathsf{Pays}\ step$$

and therefore, STRUC′ $\cap$ Pays = STRUC′. We can conclude that payout messages are not affected by adding DS-vulnerable constraints of STRUC to STRUC′. This gives us the following result, which is a security property of message-passing contracts :

*Lemma (*MSGS*-payouts and DS)* Given a filter F, a UTxO constraint c, and a structured contract $(\pi_{\mathsf{F},\mathsf{c}}, \pi_{\mathsf{Tx}}, \mathsf{STRUC})$, $\mathsf{STRUC}_{\mathsf{MSGS}}$ is not vulnerable to DS with respect to the constraint Pays.

7 Discussion

7.1 Related work

In this work we presented a stateful contract implementing asynchronous message-passing in the EUTxO ledger. Message-passing is the backbone of distributed computing [3] [12]. The π-calculus process calculus has been developed to formalize processes and message-passing between them in a distributed computing scenario [22]. We conjecture that it may be possible to apply this formalism to message-passing between structured contracts, considering them as processes under certain circumstances. However, this is the subject of future work.

The UTxO ledger design (first introduced as the BitCoin ledger [23]), and EUTxO ledger implementations [5] [13] [36] 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 content of the message is constrained as well. The contract MSGS can be viewed as a kind of linear sub-ledger within LEDGER, one that may be used as a tool in specification and verification of properties, such as temporal properties of communicating contracts, or ones needed for the applications we presented, e.g. Lemmas 5.1 5.2.

Some account-based ledger designs [8] [17] [34] use on message-passing as the default way of communication between contracts. However, in all three cases, the message-passing is synchronous, so that both the sender and receiver accounts are updated as part of processing a single transaction. Our goal of separating communication from computation for stateful contracts on the EUTxO ledger is inspired by the Scilla [29] programming language. Even though it was developed for the account-based ledger model, the communicating automata structure it uses to model contracts could potentially be used to describe message-passing structured contracts as well.

CoSplit, presented in [25], is a static analysis tool for implementing *sharding* in an account-based ledger model. Sharding is the practice of separating contract state into smaller fragments that can be affected by commuting operations, usually for the purposes of increasing parallelism and scalability. Our message-passing contract is distributed, so that each message is contained in a distinct output. There is no specially-marked output containing the consolidated message state, whose output reference a transaction must include every time *any* message is produced or consumed. So, our design does not put superfluous constraints on the order in which multiple transactions producing and consuming messages can be applied to the EUTxO ledger. It does not require further state sharding.

A version of asynchronous message-passing is implemented in the ERC-20 Ethereum contract for fungible tokens [14]. 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 authenticate script computation outputs 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 in an EUTxO ledger would significantly reduce the possibility of reordering message-passing transactions.

Formalization of blockchain and ledger functionality forms a foundation for rigorous reasoning about smart contracts security, discussed in the detailed overview [27]. Mathematical models of EUTxO and UTxO ledgers and smart contracts therein, including ours, often specify a simplified version of actual implementations [9] [16] [24] [6] [26]. Languages for writing verifiable contracts such as [7] [29] make programming on a blockchain ledger more amenable to formal verification. In this work we continue this tradition of formalizing ledgers, smart contracts, and communication models between ledger contracts, and formally studying their vulnerabilities. In particular, this work builds on the work done in [35] [9] on formalizing stateful program models and their security properties.

8 Future work

The scheme we presented in Figure 1 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 system design. For example, the msgTo field can be replaced by a constraint on some part of the output. For example, the recipient output must contain a particular NFT token in its value. Making it an option to not even specify the receiver allows the message-passing system to also function as a kind of broadcast system for authenticated data. Messages broadcast in this way can serve as untrusted oracles.

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 constraints other than the spending of the intended recipient output. This would 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 *either* a list of messages *or* some other data can also make MSGS more versatile. It may be useful in enabling a given script to participate in either computation or communication as a result of applying a transaction, depending on the redeemer specified.

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 that specify permissions for updating parts of the ledger which not reflected in the contract state representation. It is often not realistic for a contract to successfully take a step without *any* external contracts validating. For example, paying into a contract is likely to require an "external" output to be spent within the same transaction in order to cover the payment. However, it seems feasible to limit a structured contract's dependencies to message-passing only. Formalizing and proving properties about this subclass of contracts in the future is of interest.

The formalization of the double satisfaction problem we presented does not address the possibility that a script may contain a constraint preventing any other scripts from executing as part of carrying transaction validation. While this seems like an obvious way to prevent the problem, we note that "no other scripts can run during validation of this transaction" is a constraint on the transaction that is in fact itself vulnerable to double satisfaction. It is incidental that no other scripts that may benefit from the satisfaction of this constraint. This is a special case we can address in the future, however, as mentioned earlier, it is not likely that a sensible contract can be guaranteed to progress under the condition that the transaction runs no "external" scripts.

In the future, we intend to mechanize this contract and its applications in Agda, using the EUTxO ledger and structured contracts model [21], as well as in the more realistic Agda-mechanized ledger model [33] of the Cardano ledger [31], which is currently in development.

8.1 Conclusion

As discussed above, some security properties of stateful smart contracts in the EUTxO ledger model have been formalized in existing work. However, in such models, communication between contracts was not a focus, and was incidentally treated the same as any other script dependency. In this work, we focus on formalizing communication of data and assets among scripts as well as the stateful contracts they implement. We encode communication as a special kind of dependency of a script (or structured contract) on a distributed stateful contract keeping track of all unconsumed messages. Moreover, the contract we present is specified in the same small-steps semantic framework as the ledger itself. This allows us to reason about it separately from other dependencies and from script computation, making it more amenable to formal reasoning.

As an example of such reasoning, we present two usecases. The first usecase is a variation on memoization, wherein message tokens serve as proof artefacts of successful script computations. The second usecase formalizes the idea of a script communicating via message-passing, and defines when a message constitutes a "payout". We go on to formalize the pervasive double satisfaction problem as a property of a labelled transition system, and demonstrate the use of message-passing to address this vulnerability in certain cases.

A Appendix

Proof sketch of $\sim>$ *relation for* MSGS. Suppose $(\star, u, tx, u') \in \mathsf{LEDGER}$, and that $m = \pi\ u \neq \star$. Suppose that $\{\} \neq \{\mathsf{msgsTT} \mapsto tkns\} \subseteq \mathsf{mint}\ tx$. Therefore, by 9,

$\star : \{\star\}$	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. 4: Notation

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

Let *sndMsgs*, *rcvMsgs*, *newOuts*, *usedInputs* be defined as in the script msgsTT in 11, which, for the given transaction *tx*, is the same as in the MSGS transition rules 1. By inspection, the constraints (4) and (7) of the MSGS transition rules 1 are satisfied by definition of msgsTT, as they are replicated exactly in msgsTT.

The constraint (6) has a related constraint in msgsTT, call it

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

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

The constraint (2) has a related constraint in msgsTT,

$$\mathsf{noDups}\ (\mathsf{map}\ \mathsf{getMsgRef}\ sndMsgs\ +\!+\ \mathsf{map}\ \mathsf{getMsgRef}\ rcvMsgs)$$

but it does not require that, additionally,

$$\mathsf{noDups}\ (\mathsf{map}\ \mathsf{getMsgRef}\ sndMsgs\ +\!+\ \mathsf{map}\ \mathsf{inUTxO}\ (\pi\ u))$$

that is, that the fields inUTxO *msg* and msgIx *msg* of a message in *sndMsgs* cannot both be the same as those fields of some message *msgs*′ that is already in $\pi\ u$. This guaranteed by assumption (i) in Section 4. We can now use (2), (4), (6), and (7) in the rest of the proof.

Finally, we must demonstrate that

$$\pi\ u' = (\pi\ u \setminus (\mathsf{map}\ \mathsf{fst}\ rcvMsgs)) \cup (\mathsf{map}\ \mathsf{fst}\ sndMsgs)$$

We must show that $\pi\ u' \neq \star$, i.e. msgOutsOK u'. We start by showing that in u', all message tokens are still in separate outputs in quantity 1, with no duplication. All message tokens that are in u but not in inputs or outputs of *tx* still satisfy this property in u'. The newly minted singleton message tokens in mint *tx* correspond to the messages in *sndMsgs*, by (7). The messages in *sndMsgs* are not duplicates of existing messages by (2). The messages in *sndMsgs* must be minted as tokens and placed in separate outputs, locked by UTxO validator msgsVal, by (4). No new message tokens besides those corresponding to *sndMsgs* can be minted, by (7). So, all newly minted message tokens satisfy the required property. All message tokens included in the inputs of *tx* are burned, since by msgOutsOK u, the must be locked by msgsVal. So, all consumed message tokens are burned, and all produced message tokens, as well as all message tokens that are not in outputs or inputs of *tx*, satisfy the required property. This accounts for all message tokens in u' by the global POV [35].

We must now show that all inputs that were spent to mint tokens existing on u' are no longer in u'. By (4), the inUTxO field of each message *msg* in *sndMsgs* must be the output reference an input of *tx*. Therefore, by definition of u' according to LEDGER, inUTxO *msg* must be removed from u and not exist in u'. For all tokens already in u, by msgOutsOK u, this must inUTxO *msg* must not be in u. So, for all newly minted tokens, as well as all existing tokens, inUTxO *msg* must not be in u, as required. Finally, $[\![\mathsf{msgsTT}]\!]\ (\star,\ (tx,\ \mathsf{msgsTT}))$ holds true by rule 9.

Now, we must show that

$$\pi\ u' = (\pi\ u \setminus (\mathsf{map\ fst}\ \mathit{rcvMsgs})) \cup (\mathsf{map\ fst}\ \mathit{sndMsgs})$$

Let $\mathit{msg} \in \pi\ u'$. So, either $\mathsf{msgTkn}\ \mathit{msg}$ is in u or it is in outputs of tx. Note that $\mathsf{msgTkn}\ \mathit{msg}$ cannot be in inputs of tx if it is in u', because by $\mathsf{msgsVal}$ and $\mathsf{msgOutsOK}\ u$, all message tokens in inputs must be locked by $\mathsf{msgsVal}$, and therefore burned when spent. By (4), set $\mathsf{map\ fst}\ \mathit{sndMsgs}$ contains all the message tokens in outputs of tx (which are in $\mathit{newOuts}$), and the set $\pi\ u$ contains all message tokens in u. Therefore,

$$\mathit{msg} \in (\pi\ u \setminus (\mathsf{map\ fst}\ \mathit{rcvMsgs})) \cup (\mathsf{map\ fst}\ \mathit{sndMsgs})$$

To show the inclusion in the other direction, suppose

$$\mathit{msg} \in (\pi\ u \setminus (\mathsf{map\ fst}\ \mathit{rcvMsgs})) \cup (\mathsf{map\ fst}\ \mathit{sndMsgs})$$

If $\mathit{msg} \in (\pi\ u \setminus (\mathsf{map\ fst}\ \mathit{rcvMsgs})) \subseteq \pi\ u$, it is still in $\pi\ u'$. This is by the global POV, and because $\mathsf{msgTkn}\ \mathit{msg}$ is not in $\mathit{usedInputs}$, and therefore not consumed by tx. If $\mathit{msg} \in (\mathsf{map\ fst}\ \mathit{sndMsgs})$, the message token $\mathsf{msgTkn}\ \mathit{msg}$ must be in $\mathit{newOuts}$, and therefore exist in the outputs of tx. By definition of LEDGER, token $\mathsf{msgTkn}\ \mathit{msg}$ must then be in u'. So, msg is necessarily either in $(\pi\ u \setminus (\mathsf{map\ fst}\ \mathit{rcvMsgs}))$ or $(\mathsf{map\ fst}\ \mathit{sndMsgs})$, and we are done.

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

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

such that

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

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

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

so, by definition of the useMyFunction script,

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

Now, all quantities of message tokens in outputs on the ledger are exactly 1. This is implied by $\pi\ u \neq \star$, which calls $\mathsf{msgOutsOK}$ to check this. By the POV rule 5, we can conclude that one entry $p \mapsto w$ containing one token $\mathsf{msgTkn}\ m$ was spent by tx from u. We know it is exactly one entry because, again inspecting $\mathsf{msgOutsOK}$, message tokens are unique on a ledger for which $\pi\ u \neq \star$.

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

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

By inspecting msgsTT, and therefore msgsTT', we see that $(w, m) \in \mathit{newOuts}$ by definition of $\mathit{newOuts}$, and the uniqueness of the message token m (which is guaranteed by the noDups check).

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

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

and

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

By assumption (ii) in Section 4, we have

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

Recall that $(fIn,\ fOut)\ =\ \mathsf{fromData}_{IO}\ (\mathsf{msgData}\ m)$. Inspencting checkMyFunction, we get that

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

which, by definition, is true whenever $\mathsf{myFunctionCompute}\ fIn = fOut$.

Now, tx, the transaction consuming the message m, also requires that the message minting policy validate,

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

By definition of msgsTT, $(w, m) \in usedInputs$ for tx must be such that $(m,\ (\_, \mathsf{msgTo}\ m, \_))\ \in\ rcvMsgs$. Therefore, the spending of a transaction input with output $(\_, \mathsf{msgTo}\ m, \_)$ and redeemer $\mathsf{toData}\ [(\mathsf{receive}, m)]$ must validate. Note that such a $(m,\ (\_, \mathsf{msgTo}\ m, \_))\ \in\ rcvMsgs$ is necessarily unique because the noDups constraint prevents duplication of messages in message-containing redeemers. Since $\mathsf{toData}\ [(\mathsf{receive},\ \mathsf{Msg}\ m)]$ is the redeemer of that unique input with such a redeemer, and we know that the input

$$(\_, (\mathsf{useMyFunction}, v, d), \mathsf{toData}\ [(\mathsf{receive},\ \mathsf{Msg}\ m)])$$

has this redeemer in tx, it follows from $(m,\ (\_, \mathsf{msgTo}\ m, \_))\ \in\ rcvMsgs$ that $\mathsf{msgTo}\ m = (\mathsf{useMyFunction}, v, d)$.

Proof of DS-free lemma. Let STRUC be a contract, and C — a strict constraint of STRUC, so that $\mathsf{STRUC} \subsetneq \mathsf{C}$. Suppose $\mathsf{STRUC} \subseteq \mathsf{STRUC}'$, such that that $\mathsf{STRUC} \subseteq \mathsf{STRUC}' \cap \mathsf{C} \subseteq \mathsf{STRUC}'$, and $\mathsf{s}\ \mathsf{STRUC} = \mathsf{s}\ \mathsf{STRUC}'$. 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,

$$(s, s') \in \mathsf{s}\ \mathsf{STRUC}$$

We also know there is an s'' for the given s, i such that $(\star, s, i, s'') \in \mathsf{STRUC}$.

Since $\mathsf{STRUC} \subseteq \mathsf{STRUC}'$, and all systems are deterministic, s'' must be unique, and $s' = s''$, so that $(\star, s, i, s') \in \mathsf{STRUC}$.

Therefore, $\mathsf{STRUC} = \mathsf{STRUC}' = \mathsf{STRUC}' \cap \mathsf{C}$, and STRUC is not vulnerable to DS with respect to C.

References

1. Algorand Team: Algorand Developer Documentation. https://developer.algorand.org/docs/ (2021)
2. 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
3. Andrews, G.: Foundations of Multithreaded, Parallel, and Distributed Programming. Addison-Wesley (1999)
4. Auditing, V.: Cardano vulnerabilities 1 — double satisfaction (2023), https://medium.com/@vacuumlabs_auditing/cardano-vulnerabilities-1-double-satisfaction-219f1bc9665e
5. Bartoletti, M., Bracciali, A., Lepore, C., Scalas, A., Zunino, R.: A formal model of algorand smart contracts (2021)
6. Bartoletti, M., Bracciali, A., Lepore, C., Scalas, A., Zunino, R.: A formal model of algorand smart contracts (2021)
7. 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)
8. Buterin, V.: Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/en/whitepaper/ (2014)
9. Chakravarty, M.M.T., Chapman, J., MacKenzie, K., Melkonian, O., Müller, J., Peyton Jones, M., Vinogradova, P., Wadler, P.: Native custom tokens in the extended UTXO model. In: 9th International Symposium on Leveraging Applications of Formal Methods (ISoLA). Springer (2020), to appear; https://omelkonian.github.io/data/publications/eutxoma.pdf

10. Chakravarty, M.M.T., Chapman, J., MacKenzie, K., Melkonian, O., Müller, J., Peyton Jones, M., Vinogradova, P., Wadler, P., Zahnentferner, J.: Utxoma: Utxo with multi-asset support. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Applications. pp. 112–130. Springer International Publishing, Cham (2020)
11. Chakravarty, M.M.T., Chapman, J., MacKenzie, K., Melkonian, O., Peyton Jones, M., Wadler, P.: The Extended UTXO model. In: Proceedings of Trusted Smart Contracts (WTSC). LNCS, vol. 12063. Springer (2020)
12. Coulouris, G., Dollimore, J., Kindberg, T.: Distributed Systems: Concepts and Design (International Computer Science). Addison-Wesley Longman, Amsterdam (2005)
13. Ergo Team: Ergo: A Resilient Platform For ContractualMoney. https://whitepaper.io/document/753/ergo-1-whitepaper (2019)
14. Ethereum Team: Ethereum Development Documentation. https://ethereum.org/en/developers/docs/ (2022)
15. Field, A., Harrison, P.: Functional Programming. International computer science series, Addison-Wesley (1988), https://books.google.ca/books?id=nYtQAAAAMAAJ
16. Gabbay, M.J.: Algebras of utxo blockchains. Mathematical Structures in Computer Science **31**(9), 1034–1089 (2021). https://doi.org/10.1017/S0960129521000438
17. Goodman, L.: Tezos—a self-amending crypto-ledger white paper (2014)
18. Group, H.A.S.: Technical review of marlowe : Final report. Tech. rep., Tweag (2023)
19. 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)
20. Melkonian, O.: A formal model of the extended utxo model in agda. https://github.com/omelkonian/formal-utxo (2023)
21. Melkonian, O.: Structured contracts: Small-step-style simulation verification of eutxo smart contracts. https://github.com/omelkonian/structured-contracts (2023)
22. Milner, R.: Communicating and Mobile Systems: The Pi-Calculus. Cambridge University Press, Cambridge, UK (1999)
23. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/en/bitcoin-paper (October 2008)
24. Nester, C.: A foundation for ledger structures. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/OASICS.TOKENOMICS.2020.7, https://drops.dagstuhl.de/opus/volltexte/2021/13529/
25. 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
26. 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
27. 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)
28. Sanchez, F.: A comprehensive guide to marlowe's security: audit outcomes, built-in functional restrictions, and ledger security features (2023), https://iohk.io/en/blog/posts/2023/06/27/a-comprehensive-guide-to-marlowes-security-audit-outcomes-built-in-functional-restrictions-and-ledger-security
29. 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)
30. Team, C.: Small Step Semantics for Cardano. https://github.com/input-output-hk/cardano-ledger/releases/latest/download/small-step-semantics.pdf (2018)
31. Team, C.: Cardano ledger. https://github.com/input-output-hk/cardano-ledger (2023)
32. Team, C.: Double satisfaction (2023), https://plutus.readthedocs.io/en/latest/reference/writing-scripts/common-weaknesses/double-satisfaction.html
33. Team, C.: Formal ledger specifications. https://github.com/input-output-hk/formal-ledger-specifications (2023)
34. Team, T.Z.: The ZILLIQA Technical Whitepaper. https://docs.zilliqa.com/whitepaper.pdf (2017)
35. Vinogradova, P., Chakravarty, M., Wadler, P., Chapman, J., Ferariu, T., Jones, M.P., Krijnen, J., , Melkonian, O.: Structured contracts in the eutxo ledger model (2023), manuscript submitted for publication
36. 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
$\mathsf{TokenName} = [\mathsf{Data}]$	token name
$\mathsf{AssetID} = \mathsf{PolicyID} \times \mathsf{TokenName}$	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

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

Fig. 5: Primitives and basic types for the $\mathrm{EUTxO}_{\mathrm{ma}}$ model

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

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

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

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

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

Fig. 6: Auxiliary functions for entering outputs into the UTxO set

1. **Transaction has at least one input**
$$\mathsf{inputs}\ tx\ \neq\ \{\}$$
2. **The current slot is within the validity interval**
$$slot \in \mathsf{validityInterval}\ tx$$
3. **All outputs have positive values**
$$\forall o \in \mathsf{outputs}\ tx,\ \mathsf{value}\ o > 0$$
4. **All inputs refer to unspent outputs**
$$\forall (oRef,\ o)\ \in\ \{(\mathsf{outputRef}\ i,\ \mathsf{output}\ i)\mid i \in \mathsf{inputs}\ tx\},\ oRef\ \mapsto\ o\ \in\ utxo$$
5. **Value is preserved**
$$\mathsf{mint}\ tx + \sum_{i\in \mathsf{inputs}\ tx,\ (\mathsf{outputRef}\ i)\mapsto o\ \in\ utxo} \mathsf{value}\ o = \sum_{o\in \mathsf{outputs}\ tx} \mathsf{value}\ o$$
6. **No output is double spent**
$$\text{If } i_1, i \in \mathsf{inputs}\ tx \text{ and } \mathsf{fst}\ (\mathsf{outputRef}\ i) = \mathsf{outputRef}\ i \text{ then } \mathsf{fst}\ i\ = i.$$
7. **All inputs validate**
$$\text{For all } i \in \mathsf{inputs}\ tx,\ [\![\mathsf{validator}\ i]\!](\mathsf{datum}\ i,\ \mathsf{redeemer}\ i,\ (tx, i)) = \mathsf{True}$$
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. 7: Validity of a transaction t in the $\mathrm{EUTxO_{ma}}$model

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

State := $\mathbb{P}$ Msg
The MSGS state is a set of messages

Input := Tx
The MSGS input is the full transaction

Fig. 8: Message types

$$\mathsf{toData} : [(\mathsf{SR} \times \mathsf{Msg})] \to \mathsf{Data}$$

encodes a redeemer consisting of a set of messages as Data

$$\mathsf{fromData} : \mathsf{Data} \to [(\mathsf{SR} \times \mathsf{Msg})] \cup \{\star\}$$

decodes a Data redeemer as a set of messages or fails

Fig. 9: Decoding and encoding redeemers for validators of outputs sending and receiving messages

$$\mathsf{msgOutsOK} : \mathsf{UTxO} \to \mathbb{B}$$

$$\begin{aligned}\mathsf{msgOutsOK}\ \mathit{utxo} := \ & \forall\, (i \mapsto o), (j \mapsto p) \in \mathit{utxo},\ i \neq j, \{\ \mathsf{msgsTT} \mapsto \{m \mapsto q\}\} \subseteq \mathsf{value}\ o \Rightarrow \\ & (\{\ \mathsf{msgsTT} \mapsto \{m \mapsto \_\}\} \cap \mathsf{value}\ p = \{\}\ \ \wedge\ \ (q = 1) \\ & \wedge\ \ (\mathsf{inUTxO}\ (\mathsf{fromData}\ m) \mapsto \_ \notin \mathit{utxo}) \\ & \wedge\ \ (\llbracket \mathsf{msgsTT} \rrbracket\ (\star,\ (\mathsf{fst}\ i,\ \mathsf{msgsTT}))))\end{aligned}$$

$$\mathsf{msgTkn} :\ \mathsf{Msg}\ \to\ \mathsf{Value}$$

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

Message token constructor

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

$$\mathsf{SR} := \{\mathsf{send}, \mathsf{receive}\}$$

Tag specifying whether message is being sent or received

$$\mathsf{getMsgRef} : (\mathsf{SR} \times \mathsf{Msg}) \to (\mathsf{OutputRef},\ \mathsf{Ix})$$

$$\mathsf{getMsgRef}(\_, \mathit{msg}) := (\mathsf{inUTxO}\ \mathit{msg},\ \mathsf{msgIx}\ \mathit{msg})$$

Returns unique message identifier

Fig. 10: Projections and auxiliary MSGS functions

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

Fig. 11: Minting policy constructor for message tokens

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

Fig. 12: Minting policy and validator for UTxO containing message tokens