Simulating accounts via smart contracts in a UTxO Ledger

Polina Vinogradova[1], IOG folk, and Univerity of Edinburgh folk

[1] IOG, `firstname.lastname@iohk.io`
[2] University of Edinburgh, `orestis.melkonian@ed.ac.uk, wadler@inf.ed.ac.uk`

Keywords: blockchain · smart contracts · accounts

1 Introduction

Unlike the inherently stateful contracts of account-based ledger models, the EUTxO model does not have immediately obvious best practices for implementing stateful contracts. The goal of this work is to address this issue by designing a small-step-semantics-based formalism that makes it possible to specify stateful contract behaviour first, separately from implementing this behaviour using the EUTxO model's structure together with its Boolean-output, stateless contracts. In particular, the formal approach we propose specifies the relationship between ledger state update and stateful contract state update that we call *implementation*. From here on, we will refer to an EUTxO approximation of stateful contract behaviour as a *structured contract*, to avoid conflating the notion of statefulness with the reality of the functioning of the EUTxO ledger (we present a more precise definition later on in this work).

We additionally define and give examples of several other relationships between structured contract specifications, such as being an instance of, weak implementation, features, and implementation via a state-mechanism. The purpose is to demonstrate the versatily of applying the small-steps framework to analyzing structured contracts in a modular way, as well as the ease of relating functionality of smart contracts (written in the same style as the ledger specifications themselves) to the ledger specification which implemenets it.

For example, our approach gives a formal way to demonstrate that multiple implementations meet a given specification, and to specify the EUTxO mechanism via which the contract's state updates are tracked and represented on-chain (which may or may not be the same one for distinct implementations). This allows users to compare implementations across multiple parameters, such as parallelism, memory and CPU consumption, etc., while enjoying a formal property stating that the specified behaviour is indeed implemented across all of them.

In particular, we

(i) specify a simple example of a structured contract (simulating the behaviour of accounts) using a small-step semantics relation (see Sections 2.1, 2.2, and **??**),
(ii) define the notion of a structured contract on the EUTxO ledger via a small-step semantic specification relating the ledger update rule to an update of the structured contract, as well as what constitutes an implementation of a specific contract (ie. an instance of the structured contract relation), Section **??**,
(iii) show that the account simulation specification is an instance of the structured contract formalism (Section 5.1)
(iv) specify the existing NFT-based, constraint-emitting state machine mechanism (NFTCE), and show that it is an instance of the structured contract formalism (Section 6)
(v) give four specific implementations of the account simulation specification, and compare them

polina: todo, need to do all 4 implementations

(vi) specify the message-passing approach (Section 4.1) and demonstrate the use of it in the accounts example specification, Section 2.2, as an alternative to using off-chain communication to coordinate on-chain structured contract communication
(vii) demonstrate that the collection of messages itself constitutes a structured contract, for which producing and consuming messages are the transitions (Section 4.1)
(viii)

polina: todoooo :

give several additional usecases for the message-passing scheme, including addressing the double-satisfaction proble, implementing "eval", and limiting contract dependency graph size by limiting dependencies to message-type contracts only

(ix)

polina: todo :

properties to prove about structured contracts and message-passing

Thus, this approach gives users a way to modularly tackle (specify and prove properties of) the following three tasks : contract behaviour specification, implementation, and inter-contract communication - with a specified set of properties required to ensure specified behaviour is correctly implemented.

Notation :

$$
\begin{array}{rlr}
\mathit{set} \lhd \mathit{map} & = \{k \mapsto v \mid k \mapsto v \in \mathit{map},\ k \in \mathit{set}\} & \text{domain restriction} \\
\mathit{set} \not\lhd \mathit{map} & = \{k \mapsto v \mid k \mapsto v \in \mathit{map},\ k \notin \mathit{set}\} & \text{domain exclusion} \\
M \underrightarrow{\cup} N & = (\operatorname{dom} N \not\lhd M) \cup N & \text{union override right} \\
M \cup_{+} N & = (M \triangle N) \cup \{k \mapsto v_1 + v_2 \mid k \mapsto v_1 \in M \land k \mapsto v_2 \in N\} & \text{union override plus} \\
& & \text{(for monoidal values)} \\
M \cup_{-} N & = (M \triangle N) \cup \{k \mapsto v_1 - v_2 \mid k \mapsto v_1 \in M \land k \mapsto v_2 \in N\} & \text{union override minus} \\
& & \text{(for group values)} \\
\mathbb{P}\ T^* & & \\
[\![s]\!] & = \mathsf{hash}\ s & \text{hash of } s \\
& & \text{power-multi-set of type } T
\end{array}
$$

Fig. 1: Non-standard map operators

There is existing work on message-passing state machines, such as using the Scilla language, which runs on account-based architecture [3]. There are also other UTxO blockchains with smart contracts that could implement a version of our message-passing mechanism [4] [1], as well as the account-based Tezos [2].

2 Account simulation specification

In this section, we use the small steps semantics to specify basic accounts functionality.

2.1 Types and accessors for simulating accounts

In Figure 2 we give the abstract and concrete types, as well as accessor functions, used in the account simulation specification.

Account simulation abstract types

AccID	account identifier
AccState	state of an account
TxInfo	summary of transaction data a script is allowed to see
OArgs	arguments to the Open transition
CArgs	arguments to the Close transition
DArgs	arguments to the Deposit transition
WArgs	arguments to the Withdraw transition
TArgs	arguments to the Transfer transition

Account simulation concrete types

$\mathsf{Accts} = \mathsf{AccID} \mapsto \mathsf{AccState}$
collection of accounts

$\mathsf{AccInput} = \mathsf{OArgs} \mid \mathsf{CArgs} \mid \mathsf{DArgs} \mid \mathsf{WArgs} \mid \mathsf{TArgs} \mid \mathsf{TTArgs} \mid \mathsf{TFArgs}$
account transitions

Accessor functions

$\mathsf{pk} : ((\mathsf{AccID}, \mathsf{AccState}) \mid \mathsf{OArgs}) \rightarrow \mathsf{PubKey}$
public key in control of the account

$\mathsf{val} : (\mathsf{AccState} \mid \mathsf{DArgs} \mid \mathsf{WArgs} \mid \mathsf{TArgs}) \rightarrow \mathsf{Value}$
assets stored in the account

$\mathsf{id} : (\mathsf{OArgs} \mid \mathsf{CArgs} \mid \mathsf{DArgs} \mid \mathsf{WArgs}) \rightarrow \mathsf{AccID}$
account ID specified in input

Fig. 2: Specification of an account simulation (functions and types)

2.2 Account simulation specification

polina: we need to address initial states in this and the SM spec

In Figure 3 we give the type of the account simulation transition.

$$\_ \vdash \_ \xrightarrow[\text{ACCNT}]{\_} \_ \subseteq \mathbb{P}\,(\mathsf{TxInfo} \times \mathsf{AccState} \times \mathsf{AccInput} \times \mathsf{AccState})$$

Fig. 3: Account state transition type

In Figure 4, we give the specification of the rules for opening and closing an account.

$$
\text{Open}\;\frac{\begin{array}{c} accIn \in \mathsf{OArgs} \\[6pt] \mathsf{pk}\ accIn \in \mathsf{txInfoSignatories}\ txInfo \qquad id\ :=\ \mathsf{id}\ accIn \qquad id\ \notin\ \mathsf{dom}\ accts \\[6pt] \mathsf{pk}\ (id,\ newAcct)\ =\ \mathsf{pk}\ accIn \qquad\qquad \mathsf{val}\ newAcct\ =\ \mathsf{zero} \end{array}}{txInfo \vdash \left(accts\right) \xrightarrow[\text{ACCNT}]{accIn} \left(\boldsymbol{accts \cup \{id\ \mapsto\ newAcct\}}\right)} \tag{1}
$$

$$
\text{Close}\;\frac{\begin{array}{c} accIn \in \mathsf{CArgs} \\[6pt] \mathsf{id}\ accIn\ \mapsto\ acntToClose \in\ accts \\ \mathsf{pk}\ accIn \in \mathsf{txInfoSignatories}\ txInfo \qquad \mathsf{val}\ acntToClose\ =\ \mathsf{zero} \end{array}}{txInfo \vdash \left(accts\right) \xrightarrow[\text{ACCNT}]{accIn} \left(\boldsymbol{(\mathsf{id}\ accIn) \ntriangleleft accts}\right)} \tag{2}
$$

Fig. 4: Specification of the account simulation via small-step semantics (open and close)

In Figure 5, we give the specification of the rules for withdrawing and depositing.

$$
\text{Deposit}\;\frac{\begin{array}{c} accIn \in \mathsf{DArgs} \\[6pt] id := \mathsf{id}\ accIn \qquad \mathsf{val}\ accIn \geq \mathsf{zero} \qquad id\ \mapsto oldAcct\ \in\ accts \\[6pt] pk := \mathsf{pk}\ (id,\ oldAcct) \qquad\qquad pk \in \mathsf{txInfoSignatories}\ txInfo \\[6pt] \mathsf{pk}\ (id,\ changedAcct)\ =\ pk \qquad \mathsf{val}\ changedAcct\ =\ \mathsf{val}\ oldAcct\ +\ \mathsf{val}\ accIn \end{array}}{txInfo \vdash \left(accts\right) \xrightarrow[\text{ACCNT}]{accIn} \left(\boldsymbol{accts\ \underset{\rightarrow}{\cup}\ \{id\ \mapsto\ changedAcct\}}\right)} \tag{3}
$$

$$
\text{Withdraw}\;\frac{\begin{array}{c} accIn \in \mathsf{WArgs} \\[6pt] id := \mathsf{id}\ accIn \qquad\qquad id\ \mapsto oldAcct\ \in\ accts \\ \mathsf{val}\ oldAcct \geq \mathsf{val}\ accIn \geq \mathsf{zero} \\[6pt] pk := \mathsf{pk}\ (id,\ oldAcct) \qquad\qquad pk \in \mathsf{txInfoSignatories}\ txInfo \\[6pt] \mathsf{pk}\ (id,\ changedAcct)\ =\ pk \qquad \mathsf{val}\ changedAcct\ =\ \mathsf{val}\ oldAcct\ +\ \mathsf{val}\ accIn \end{array}}{txInfo \vdash \left(accts\right) \xrightarrow[\text{ACCNT}]{accIn} \left(\boldsymbol{accts\ \underset{\rightarrow}{\cup}\ \{id\ \mapsto\ changedAcct\}}\right)} \tag{4}
$$

Fig. 5: Specification of the account simulation via small-step semantics (deposit and withdraw)

In Figure 6, we give the specification of the rules for transferring.

$$
\text{Transfer}\ \frac{\begin{array}{c}
accIn \in \mathsf{TArgs} \\[1em]
\mathsf{idFrom}\ accIn \lhd accts \ = \ idFrom \mapsto oldAcctFrom \\
\mathsf{idTo}\ accIn \lhd accts \ = \ idTo \mapsto oldAcctTo \\[1em]
pkFrom := \mathsf{pk}\ (idFrom,\ oldAcctFrom) \qquad pkTo := \mathsf{pk}\ (idTo,\ oldAcctTo) \qquad idFrom \neq idTo \\
pkFrom,\ pkTo \in \mathsf{txInfoSignatories}\ txInfo \qquad \mathsf{val}\ oldAcctFrom \geq \mathsf{val}\ accIn \geq \mathsf{zero} \\[1em]
\mathsf{pk}\ (idFrom,\ changedAcctFrom) \ = \ pkFrom \qquad \mathsf{pk}\ (idTo,\ changedAcctTo) \ = \ pkTo \\[1em]
\mathsf{val}\ changedAcctFrom \ = \ \mathsf{val}\ oldAcctFrom \ - \ \mathsf{val}\ accIn \\
\mathsf{val}\ changedAcctTo \ = \ \mathsf{val}\ oldAcctTo \ + \ \mathsf{val}\ accIn
\end{array}}{txInfo \vdash \left(accts\right) \xrightarrow[\text{ACCNT}]{accIn} \left(\boldsymbol{accts \underrightarrow{\cup} \{idFrom \ \mapsto \ changedAcctFrom,\ idTo \ \mapsto \ changedAcctTo\}}\right)} \tag{5}
$$

Fig. 6: Specification of the account simulation via small-step semantics (transfer)

2.3 Versions of account simulations

Multi-operation account simulation transition In Figure 7, we give the transition type for a list of account operations applied in sequence. In Figure 8, we give the base and the inductive rule for applying a list of account operations.

Note that the *txInfo* context is a free variable in the ACCNTS transition rule $\mathsf{Seq} - \mathsf{accnts} - \mathsf{ind}$ precondition that is the ACCNTS transition with signal Γ. This is because different transactions may be performing each step of the multi-step ACCNTS transition. If $txInfo = txInfo'$ in that rule, the same transaction is performing the two adjacent steps.

$$\_ \vdash \_ \xrightarrow[\text{ACCNTS}]{\_} \_ \subseteq \mathbb{P}\,(\mathsf{TxInfo} \times \mathsf{AccState} \times [\mathsf{AccInput}] \times \mathsf{AccState})$$

Fig. 7: Account state - sequence of operations transition type

$$\text{Seq-accnts-base}\,\frac{}{txInfo \vdash accs \xrightarrow[\text{ACCNTS}]{\epsilon} accs} \tag{6}$$

$$\text{Seq-accnts-ind}\,\frac{txInfo' \vdash accs \xrightarrow[\text{ACCNTS}]{\Gamma} accs' \qquad txInfo \vdash accs' \xrightarrow[\text{ACCNT}]{accIn} accs''}{txInfo \vdash accs \xrightarrow[\text{ACCNTS}]{\Gamma;\, accIn} \boldsymbol{accs''}} \tag{7}$$

Fig. 8: Rules for applying a sequence of account operations

2.4 Hoare-style specification

polina: Clean this up

3 Structured Contracts

3.1 Plutus-implemented structured contracts on the ledger

An EUTxO *structured contract* is given by specifying the following data

(i) some set of inference rules and concrete types $(\mathsf{Env}, \mathsf{State}, \mathsf{Input})$ that specify the transition of type SMUP in Figure 10

(ii) a surjective function

$$\pi_{\mathsf{State}} \in \mathsf{LState} \to \mathsf{State}$$

(iii) a surjective function

$$\pi_{\mathsf{Input}} \in \mathsf{TxInfo} \to \mathsf{Input}$$

(iv) a surjective function

$$\pi_{\mathsf{Env}} \in \mathsf{TxInfo} \to \mathsf{Env}$$

(v) a proof that the StatefulStep and the StatefulNoStep properties in Figure 11 is satisfied by the data in (i), (ii), and (iii)

(vi) language Lang in which the implementation is done

We say that the functions in (i) and (ii), and the transition in (iii), *implement* a structured contract. Note that since a transaction does not necessarily update the state of a structured contract, the contract input will usually have a trivial input type in the top level disjunction.

polina: maybe bring back StatefulNoStep - for when isValid is false

In Figure 11, the first two arguments to txInfo, which are EI and SysSt, are system constants, and Lang is the language the implementation is written in (eg. PlutusV1 or PlutusV2). We need to specify the language in order to be able to compute *txInfo*.

polina: This property need only hold in an epoch for which the given EpochInfo (the EI variable passed to txInfo)

Note here that these projection functions are not implemented on-chain, however, they can be instantiated off-chain (eg. within a proof assistant) to observe the state transition of the contract being implemented, and used to prove properties of the on-chain behaviour of implemented contract.

polina: how do we make sure that we don't have trivial implementation? Or trivial rule state updates? is it enough that the projections are surjective?

polina: TxInfo is only consistent across the implementation if the language in which the implementation is written is the same for all Plutus contracts executed as part of the implementation - how would I make this formal?

polina: If we want to implement something that involves data outside the scope of the context or state of the LEDGER transition, eg. any block-level updates, we can adjust the notion of implementation to accommodate that?

Structured contracts, valid and initial states A valid ledger state is one that is either the initial state, or any state reachable from an initial state by the application of a trace of valid blocks. Additionally, for the purposes of reasoning about contracts, we consider valid ledger states to be those reachable by a sequence of blocks, followed by the application of a block header and some prefix of a list of transactions in a given block.

A definition of a valid contract state is required to be able to specify properties of contracts, as it does not make sense to make any claims about invalid states. We will later discuss in more detail safety and liveness properties of contracts, and their relation to the evolution of ledger state. First, however, we must specify what kinds of states we will be referring to in such properties.

Instead of studying contract states reachable from the contract state at initial ledger state, it is more practical and convenient to pick any recent ledger state, and prove properties about structured contract states reachable from that state. Subsequently, we will use the following terminology in stating properties of structured contracts :

(i) Given some observed ledger state lsi, we refer to the state $s_{lsi} := \pi_{\mathsf{State}}\ lsi$ of a structured contract as the *lsi-initial state*
(ii) All contract states reachable from some lsi-initial state s via the application of sequence of inputs $i_1, ..., i_k$ are *lsi-valid states*

For example, some approaches to intoducing a natural notion of initiality to a structured contract exist, such identifying entering an initial contract state with some event that can occur at most once in the evolution of ledger state. To accommodate reasoning about such natural notions of initiality, we would pick a ledger state lsi to be one where the once-in-a-lifetime ledger event has not ever occured.

Proposition. For a given lsi, let V denote the collection of lsi-valid states for a structured contract S. Let L denote all ledger states reachable from lsi. Then,

$$\{\ \pi_{\mathsf{State}}\ ls \ \mid\ ls\ \in\ L\ \}\ \subseteq\ V$$

polina: prove this

Account simulation as an instance of a structured contract. A *specific instance of a structured contract* is given by concrete State and Input types, and concrete projection functions. For examle, a concrete implementation of the account simulation is an instance of a structured contract, where

$$\mathsf{State} := \mathsf{AccState},\ \ \mathsf{Input} := \mathsf{AccInput}$$

and the specific rules of the transition relation SMUP are given by the account simulation transition rules, ACCNT. Alternatively, we may also simulate multiple steps being perfomed by one transaction if we set $\mathsf{Input} := [\mathsf{AccInput}]$ and use the rules of the ACCNTS transition.

Features of a structured contract. Rather of showing that an implementation of a structured contract specification (eg. the account simulation) itself is an instance of a structured contract, we can instead assess whether a structured contract *has a certain feature*. The data specifying a feature of a structured contract is very similar to the data specifying an implementation of a structured contract, except the feature is implemented by a structured contract with state and input $(\mathsf{State}, \mathsf{Input})$ and transition SMUP, rather than the ledger state update LEDGER, as follows :

(i) some set of inference rules and concrete types ($\mathsf{FState},\ \mathsf{FInput}$) that specify the transition of type FSMUP
(ii)
$$\pi_{\mathsf{FState}} \in \mathsf{State} \to \mathsf{FState}$$
(iii)
$$\pi_{\mathsf{FInput}} \in \mathsf{Input} \to \mathsf{FInput}^{?}$$
(iv)
$$\pi_{\mathsf{FEnv}} \in \mathsf{Env} \to \mathsf{FEnv}$$

(v) a proof that the $\mathsf{FeatureStep}$, $\mathsf{FeatureNoStep}$ properties in Figure 11 is satisfied by the data in (i), (ii), and (iii) (see Figure 12)

Note that the state of a feature may not be affected by applying the input of the structured contract to its state, that is why a feature is beholden to two contraints, $\mathsf{FeatureStep}$, $\mathsf{FeatureNoStep}$, to allow for both trivial and non-trivial feature updates.

polina: does this (and weak version) need to be a bisimulation? is this the right direction of simulation, if not?

Weakly-implemented feature It is possible that for a given structured contract with transition SMUP, we want to allow multiple steps of this transition to implement a single step of a feature transition FSMUP.

To do this, we first define a transition SMUPS in Figure 13 to be the recursively defined application of a list of SMUP transition inputs to a given state (similar to the recursive definition in Figure 8).

We then replace the simulation property in Figure 12 by a weaker property in Figure 14, and require the existence of the following function such that the property holds :

$$\mathsf{getSteps}\,\mathit{finp} \in \mathsf{FInput} \to [\mathsf{Input}]$$

We say that FSMUP is *weakly implemented by* SMUP.

polina: SPECIFY SMUP as it applies each input to each txInfo!!

3.2 Properties of contracts and their implementations

Realizability . Both contracts specifications and their implementations can have properties specified about them. A property, which we call *realizability*, that can be expressed for any structured contract, is as follows (recall here that the structured contract projection functions $\pi_{\mathsf{State}}, \pi_{\mathsf{Input}}$ are surjective) :

Given some initial state $s \in \mathsf{State}$ and a ledger state l such that $\pi_{\mathsf{State}}\ l = s$, for any $s' \in \mathsf{State}$ reachable from s via a trace of $[i_1, ..., i_k] \in [\mathsf{Input}]$, there exists a trace $[(env_1,\ tx_1), ..., (env_k,\ tx_k)]$ such that $\pi_{\mathsf{Input}}\ txInfo_j = i_j$ for $1 \leq j \leq k$, and $env_j\ \in\ \mathsf{Slot} \times \mathsf{Ix} \times \mathsf{PParams} \times \mathsf{Coin}$, where

$$txInfo_j = \mathsf{txInfo\ El\ SysSt\ Lang}\ pp_j\ (utxo_j)\ tx_j$$

Here, pp_j are the protocol parameters (PParams) in env_j, and $utxo_j$ is the UTxO set in LState.

polina: The initial state s is supposed to be constrained to be an actual state that appeared on-chain at some point? or something like that

Weaker properties . In practice, such a property is difficult to prove in the general case. Instead, the approach we take is to specify weaker propeties about the existence of traces at the ledger level. First, however, we consider the different kinds of properties that can be expressed at the structured contract level, and how they relate to realizability.

Let us here fix a real ledger state lsi, and write ”valid” instead of lsi-valid, for brevity.

(i) Safety properties, which are invariants about tuples (env, s, i, s'), where s is a valid state, and i is a valid transition from s to s' in context env, state that ”a specific bad thing will never happen”. A safety property that holds for some collection V of such tuples (env, s, i, s') will also hold for the collection L of state

$$(env, s, i, s') \in V$$

polina: pi-input : LState -¿ Input
(s) -¿ (s′) in LEDGER

Formally, given a for a property $\mathsf{P} \in\rightarrow \mathsf{Bool}, \mathsf{P}$ hold for every apply to every structured contract state and transition reachable from into are not violated by any transition that starts at a reachable ledger state ls

$$\forall (env,\ tx,\ ls),\ env \vdash ls \xrightarrow[\text{LEDGER}]{tx} ls',$$

$$\exists s\ s'\ i,\ \pi_{\mathsf{State}}\ ls = s, \pi_{\mathsf{State}}\ ls' = s'$$

(ii) Liveness properties.

Double Satisfaction .

A transaction is made up of parts that change the state of the ledger, and those that are only used to check validity. For example, the validity interval is not used in updating the ledger state, it is only used to check if the update by the transaction is allowed in the current slot. That is, we can define a subtype

$$\mathsf{LCTx} = [\mathsf{TxInInfo}] \times [\mathsf{TxOut}] \times \mathsf{Value} \times \mathsf{DCert}$$

and an embedding/retraction pair $\mathsf{lc} \circ \mathsf{embedLC} = \mathsf{id}_{\mathsf{LCTx}}$

$$\mathsf{lc} \; : \; \mathsf{TxInfo} \to \mathsf{LCTx} \; : \; \mathsf{embedLC}$$

so that given $(env, ls, tx, ls') \; \in \; \mathsf{LEDGER}$, such that $\mathsf{isValid}\ tx$, and

$$txInfo := \mathsf{txInfo}\ \mathsf{EI}\ \mathsf{SysSt}\ \mathsf{Lang}\ pp\ (\mathsf{getUTxO}\ ls)\ tx$$

ls' is defined strictly in terms of ls, $\mathsf{lc}\ txInfo$, and $\mathsf{txId}\ tx$.

polina: this needs to be made precise and usable

Given a specifications,

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

And its implementations (satisfying the $\mathsf{StatefulStep}$ constraint),

$$(\pi_{\mathsf{Env}}, \pi_{\mathsf{State}}, \pi_{\mathsf{Input}})$$

We say that the SMUP is *not succeptible to double satisfaction* whenever in the rules of SMUP, all predicates on the ledger-altering parts of the *txInfo* transaction data (ie. LCTx) correspond to state changes in the structured contract.

More specifically,

$\forall\ \mathsf{P} : \mathsf{LCTx} \to \mathsf{Bool}$,
$\exists \mathsf{P}' : (\mathsf{State} \times \mathsf{State}) \to \mathsf{Bool}$, such that $\forall\ txInfo$,

$(\pi_{\mathsf{Env}}\ txInfo,\ s,\ \pi_{\mathsf{Input}}\ txInfo,\ s') \; \in \; \mathsf{SMUP}$,

$\mathsf{P}'\ (s, s') \; \Rightarrow \; \mathsf{P}'\ (\mathsf{lc}\ txInfo)$

We can interpret this as "no other structured contract that does not share state with SMUP that is also being executed in the same transaction is being satisfied by the changes the transaction makes".

A common approach to addressing double satisfaction for a script s is to preclude other scripts from reading (ie. using as input) data that is used in as input in the validation of s. One way to do this without ledger-level changes is to include a constraint $\mathsf{noOtherScriptsAreRun}$ *scriptContext* in s verifying that the transaction executing it cannot also contain other scripts. This does not, in fact, work perfectly - note here that double satisfaction is not necessarily only a problem of a transaction action satisfying two different boolean predicate scripts. One can imagine that if a user already happened to be paying to some public key k for some off-chain exchange of goods, for example, and some badly designed DEX-type script requires a payout to k, the user who is already paying k can benefit by also executing the DEX script in the same transaction where they make their payment, and save money on the DEX execution.

This situation can be addressed by marking any payments that are made to satisfy the structured contract encompassing the DEX by placing a special token in the UTxO containing the payment, so as to mark that payment as part of the DEX structured contract state.

Example.

An example of a predicate P on the ledger-updating parts of TxInfo which that does not care about the contract state is :

$$(txInfo, s, i, s') \in \mathsf{P} \Leftrightarrow \mathit{assetID} \mapsto 1 \in \mathsf{txInfoMint}\ (\mathsf{lc}\ \mathit{txInfo})$$

The mint field updates the ledger, and is included in $\mathsf{lc}\ txInfo$.

Let us consider the accts transition, and pretend (for no good reason) that closing an account requires minting a token with some *assetID*, that is constant and not dependent on the account being closed.

This intuitively makes for potential for doule satisfaction. Did the author of the contract mean that for each account being closed such a token must be minted? That's not what the specification requires. We can easily define a predicate

$$\mathsf{P}'\ \mathit{ti}\ :=\ \mathit{assetID} \mapsto 1 \in \mathsf{txInfoMint}\ \mathit{ti}$$

which implies the P above, since it already does not depend on the contract state, but does depend on $ti = \mathsf{lc}\ txInfo$.

One can attempt to address this by ensuring a state update reflecting the constraints imposed by P' in order to be specific about what the intent of the check was. For example, if a single token must be minted each time a single account is closed, the account state should be updated to keep track of all tokens with ID $assetID$ that exist on the ledger,

$\mathsf{AccState} \times \mathsf{TotalAssetID}$

This, by itself, is not enough, since the $\mathsf{StatefulStep}$ property will no longer be satisfied, as the total tokens with that asset ID can be increased or decreased while no satisfying the ACCTS transition. The policy of $assetID$ can, instead, for example be written to reflect that it can *only* be minted when an account is closed.

$$\text{Close}\ \frac{\begin{array}{c} \mathit{accIn} \in \mathsf{CArgs} \\[1ex] \mathsf{id}\ \mathit{accIn}\ \mapsto\ \mathit{acntToClose} \in\ \mathit{accts} \\ \mathsf{pk}\ \mathit{accIn} \in \mathsf{txInfoSignatories}\ \mathit{txInfo} \qquad \mathsf{val}\ \mathit{acntToClose}\ =\ \mathsf{zero} \\[1ex] \mathit{assetID} \mapsto 1 \in \mathsf{txInfoMint}\ \mathit{txInfo} \end{array}}{\mathit{txInfo} \vdash \left(\mathit{accts}\right) \xrightarrow[\text{ACCNT}]{accIn} \left(\,(\mathbf{id}\ \boldsymbol{accIn}) \ntriangleleft \boldsymbol{accts}\,\right)} \tag{13}$$

While intuitively, this works - no other s

- Global solution - for ANY state, we can have a guarantee that a transaction will have a certain effect??

4 Message-passing

4.1 Message-passing specification

Figure 16 gives the functions required for message-passing. Suppose a transition SMACC is a simplified accounts relation (as in Section **??**). We say that it *is a message-passing contract* whenever there is a projection function from the input type,

$$\pi_{\mathsf{MsgIn}} \in \mathsf{SimplInput} \to \mathsf{MsgIn}$$

and the transition type in Figure 19, is specified by the rules in Figure 17.

Message-passing implementation as a structured contract An implementation of message passing is itself a structured contract with its projection functions specified in Figure 18, and a single transition rule Mint of the MSGS transition. In the same figure, we specify the (previously left abstract in the specification) functions required for an instance of the message passing transition.

polina: Can we produce one message for two scripts - this would be bad, check that for each input there is a message if the redeemer is a message

polina: Full messages and list of full messages as token names should be replaced with hashes

polina: Specify the preimage of msgPolHash, do proofs required for implementation, and properties here

polina: example property - the Msgs signal has to have no duplicates - this isnt true currently in the spec

4.2 Accounts with message-passing

Message-passing can be part of a structured contract without fixing an implementation of message-passing itself (example implementation is, however, given in Section 4.1). In this section, we demonstrate an example of this by specifying a contract MPACCNT, which

(i) weakly implements the account-simulation transition ACCNT

(ii) is an instance of a message-passing contract

First, we specify SimplInput to be AccMsgs, as follows

$$\mathsf{AccMsgs} := \mathsf{MsgIn} \uplus \mathsf{OArgs} \uplus \mathsf{CArgs} \uplus \mathsf{DArgs} \uplus \mathsf{WArgs}$$

and the projection function

$$\pi_{\mathsf{MsgIn}}\ inp = \begin{cases} inp & \text{if } inp \in \mathsf{MsgIn} \\ ([], []) & \text{otherwise} \end{cases}$$

We make concrete the transition type for MPACCNT in Figure 19). We specify the transition rules for this type, which are made up of rules previously specified for non-message passing accoutnts, as well as two new rules Transfer − From and Transfer − To, which replace the non-message passing transition of Transfer.

Open Figure 4

Close Figure 4

Withdraw Figure 5

Deposit Figure 5

Transfer − From Figure 20

Transfer − To Figure 21

polina: We need to prove (i) and (ii)

polina: Also open/close an account from a message?

4.3 Wallets and message-passing

In this work, for simplicity, we identify a wallet with a unique public key. We can then define wallets as instances of simplified accounts, as specified by the projection and auxiliary functions in Figure 23.

polina: This is an extremely simplified idea of a wallet, which should be at some point compared agains the real wallet spec

Note that not all data in txInfo is used in the update of wallets' accounts, but we do not discard unused data in the projection, as this seems simpler. We specify the wallet account update transition with message passing in Figure 22.

In the UpdateWallets rule, we check that every message token being minted is signed by the sender (if it is a public key), and every message token being consumed is signed by the receiver (whenever it is a public key).

The π_{MsgIn} function specified the projection from TxInfo to a list of produced and consumed messages, as needed for a message-passing contract.

5 Account simulation implementations

5.1 Example implementations of account simulations

To implement account simulation as a structured contract directly on the ledger we can simply set $\mathsf{SMUP} := \mathsf{ACCNT}$ and specify the surjective projections functions, then, prove the property StatefulStep.

However, we will instead demonstrate implementations of account simulation via the two structured contract mechanisms we presented earlier.

We choose the language to be PlutusV2, so that $\mathsf{Lang} :=:= \mathsf{PlutusV2}$ for all the upcoming examples.

Naive implementation via NFTCE The simplest version of the account simulation is one that we call the *naive implementation*, and we build by specifying the required data for the NFTCE state machine implementation, rather than the structured contract formalism directly. Recall here that the reason for this is that we have already showed that an NFTCE specification gives us an instance of a structured contract. We specify the required data in Figure 24. We must also specify the utxoNFT UTxO input which must be spent to start the state machine, as well as the initState state, in which the state machine starts.

Multi-threaded direct transfer

5.2 Emitting constraints vs. message-passing

Emitting constraints ...

6 Stateful mechanisms

A *structured contract mechanism* is a a state machine transition relation SMECH that abstracts away the details of interaction with the ledger for some contract SMSPEC, effectively implementing/mediating its ledger interaction and representation aspect. Both transition types are specified in Figure 25. In order to be a structured contract mechanism for the SMSPEC contract, the SMECH transition, given surjective projections $\pi_s \in \mathsf{State} \to \mathsf{State}'$ and $\pi_i \in \mathsf{Input} \to \mathsf{Input}'$,

(i) must be a structured contract (as defined in Figure 11)
(ii) must satisfy the MechanismSim property in Figure 25

When two transition relations SMSPEC and SMECH satisfy (i) and (ii) above, we say that SMSPEC *is implemented via* SMECH.

A contraint-emitting state machine implementation of the structured contract formalism The existing structured contract formalism is based on the idea of implementing a constraint-emitting state machine via propagating an NFT through a dependent sequence of UTxO entries, which makes up the state machine graph. We will call this type of state machine NFTCE, for NFT-based and constraint-emitting.

For a given UTxO set, the state of the state machine is the datum of the only entry containing the NFT associated with the state machine (see the function π_{State} that projects to the state, Figure 26). The input of the transition function is given by the redeemer associated with the transaction input containing the NFT, computed by π_{Input} from the TxInfo of the transaction.

In order to specify an instance of a NFTCE machine, the following data is required (see Figure 26 for details) :

(i) The State and Input types

(ii) The $\mathsf{buildConstraits}$ function which specifies which constraints are emitted when

(iii) The $\mathsf{utxoNFT}$ UTxO input which must be spent to start the state machine

(iv) The $\mathsf{initState}$ state, in which the machine must start (including both the state and the value specified)

The transition relation NFTCE (with transition rules $\mathsf{MintsNFT}$ and $\mathsf{PropagatesNFT}$, specified in 27), together with the projection functions π_{State} and π_{Input}, give an implementation of the SMUP structured contract transition.

It remains to check that the structured contract instance NFTCE indeed satisfies the $\mathsf{StatefulStep}$ property in Figure 11, for any contract SMSPEC being implemented. Note here that this result will allow us to specify the data for an NFTCE, rather than the data required for a structured contract, and obtain an instance of one via the NFTCE mechanism, as implied by this property

$$\mathsf{PolicyID} =$$

polina: actually check these properties

polina: need script context here

polina: need to add invariant that there is at most 1 thread token on the ledger as a precondition for the rules

polina: can prove that invariant for all valid states starting from an initial state with certain properties

Multi-threaded NFT implementation

References

1. Ergo Team: Ergo: A Resilient Platform For ContractualMoney. https://whitepaper.io/document/753/ergo-1-whitepaper (2019)
2. Goodman, L.: Tezos—a self-amending crypto-ledger white paper (2014)
3. 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)
4. 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)

Open

$$\begin{array}{ll} \text{Arguments}: \ \mathsf{OArgs} & \text{arguments type} \\ \quad \mathsf{getID} \ : \ \mathsf{OArgs} \ \rightarrow \ \mathsf{AccID} & \text{returns the ID of the account to be opened} \end{array}$$

$$\begin{array}{rl} \text{Authentication}: & \mathsf{hasSig}\ \mathit{txinfo}\ \mathit{args} \ = \ (\mathsf{getPK}\ \mathit{accts}\ \mathit{args},\ \_)\ \mathsf{in}\ \mathit{txinfo.sigs} \\ \text{Preconditions}: & \mathsf{notOpen}\ \mathit{accts}\ \mathit{args} = \ (\mathsf{getID}\ (\mathit{args})) \ \notin \ (\mathsf{dom}\ \mathit{accts}) \\ & \mathit{acctsOld} \ := \ \mathit{accts} \\ \text{Postconditions}: & \mathsf{isOpen}\ \mathit{accts}\ \mathit{args} \ = \ (\mathsf{getID}\ (\mathit{args}) \ \mapsto \ \mathsf{zero}) \ \in \ \mathit{accts} \\ & \forall\ \mathit{id} \ \in \ \mathsf{dom}\ \mathit{accts} \ \cup \ \mathsf{dom}\ \mathit{acctsOld},\ \mathit{id} \ \neq \ \mathsf{getID}\ \mathit{args}, \\ & \quad \mathit{id} \ \mapsto \ s \ \in \ \mathit{accts} \ \Leftrightarrow \ \mathit{id} \ \mapsto \ s \ \in \ \mathit{acctsOld} \end{array}$$

Close

$$\begin{array}{ll} \text{Arguments}: \ \mathsf{CArgs} & \text{arguments type} \\ \quad \mathsf{getID} \ : \ \mathsf{CArgs} \ \rightarrow \ \mathsf{AccID} & \text{returns the ID of the account to be closed} \end{array}$$

$$\begin{array}{rl} \text{Authentication}: & \mathsf{hasSig}\ \mathit{txinfo}\ \mathit{args} \ = \ (\mathsf{getPK}\ \mathit{accts}\ \mathit{args},\ \_)\ \mathsf{in}\ \mathit{txinfo.sigs} \\ \text{Preconditions}: & \mathsf{isOpen}\ \mathit{accts}\ \mathit{args} \ = \ (\mathsf{getID}\ (\mathit{args}) \ \mapsto \ \mathsf{zero}) \ \in \ \mathit{accts} \\ & \mathit{acctsOld} \ := \ \mathit{accts} \\ \text{Postconditions}: & \mathsf{notOpen}\ \mathit{accts}\ \mathit{args} = \ (\mathsf{getID}\ (\mathit{args})) \ \notin \ (\mathsf{dom}\ \mathit{accts}) \\ & \forall\ \mathit{id} \ \in \ \mathsf{dom}\ \mathit{accts} \ \cup \ \mathsf{dom}\ \mathit{acctsOld},\ \mathit{id} \ \neq \ \mathsf{getID}\ \mathit{args}, \\ & \quad \mathit{id} \ \mapsto \ s \ \in \ \mathit{accts} \ \Leftrightarrow \ \mathit{id} \ \mapsto \ s \ \in \ \mathit{acctsOld} \end{array}$$

Deposit

$$\begin{array}{ll} \text{Arguments}: \ \mathsf{DArgs} & \text{arguments type} \\ \quad \mathsf{getID} \ : \ \mathsf{DArgs} \ \rightarrow \ \mathsf{AccID} & \text{returns the ID of the account to deposit into} \\ \quad \mathsf{val} \ : \ \mathsf{DArgs} \ \rightarrow \ \mathsf{Value} & \text{returns the assets to be deposited} \end{array}$$

$$\begin{array}{rl} \text{Authentication}: & \mathsf{hasSig}\ \mathit{txinfo}\ \mathit{args} \ = \ (\mathsf{getPK}\ \mathit{accts}\ \mathit{args},\ \_)\ \mathsf{in}\ \mathit{txinfo.sigs} \\ \text{Preconditions}: & \mathsf{nonNegV}\ \mathit{args} \ = \ \mathsf{val}\ \mathit{args} \ \geq \ 0 \\ & v_c \ := \ \mathsf{hasV}\ \mathit{accts}\ \mathit{args} \ = \ \mathsf{val}\ (\mathit{accts}\ (\mathsf{getID}\ (\mathit{args}))) \\ & \mathit{acctsOld} \ := \ \mathit{accts} \\ \text{Postconditions}: & \mathsf{addV}\ \mathit{accts}\ \mathit{args} = \ \mathsf{val}\ (\mathit{accts}\ (\mathsf{getID}\ (\mathit{args}))) \ == \ v_c \ + \ v \\ & \forall\ \mathit{id} \ \in \ \mathsf{dom}\ \mathit{accts} \ \cup \ \mathsf{dom}\ \mathit{acctsOld},\ \mathit{id} \ \neq \ \mathsf{getID}\ \mathit{args}, \\ & \quad \mathit{id} \ \mapsto \ s \ \in \ \mathit{accts} \ \Leftrightarrow \ \mathit{id} \ \mapsto \ s \ \in \ \mathit{acctsOld} \end{array}$$

Withdraw

$$\begin{array}{ll} \text{Arguments}: \ \mathsf{WArgs} & \text{arguments type} \\ \quad \mathsf{getID} \ : \ \mathsf{WArgs} \ \rightarrow \ \mathsf{AccID} & \text{returns the ID of the account to withdraw from} \\ \quad \mathsf{val} \ : \ \mathsf{WArgs} \ \rightarrow \ \mathsf{Value} & \text{returns the assets to be withdrawn} \end{array}$$

$$\begin{array}{rl} \text{Authentication}: & \mathsf{hasSig}\ \mathit{txinfo}\ \mathit{args} \ = \ (\mathsf{getPK}\ \mathit{accts}\ \mathit{args},\ \_)\ \mathsf{in}\ \mathit{txinfo.sigs} \\ \text{Preconditions}: & \mathsf{nonNegV}\ \mathit{args} \ = \ \mathsf{val}\ \mathit{args} \ \geq \ 0 \\ & v_c \ := \ \mathsf{plusV}\ \mathit{accts}\ \mathit{args} \ = \ \mathsf{val}\ (\mathit{accts}\ (\mathsf{getID}\ (\mathit{args}))) \\ & \mathsf{vIsEnough} \ = \ v_c \ \geq \ v \\ & \mathit{acctsOld} \ := \ \mathit{accts} \\ \text{Postconditions}: & \mathsf{minusV}\ \mathit{accts}\ \mathit{args} = \ \mathsf{val}\ (\mathit{accts}\ (\mathsf{getID}\ (\mathit{args}))) \ == \ v_c \ - \ v \\ & \forall\ \mathit{id} \ \in \ \mathsf{dom}\ \mathit{accts} \ \cup \ \mathsf{dom}\ \mathit{acctsOld},\ \mathit{id} \ \neq \ \mathsf{getID}\ \mathit{args}, \\ & \quad \mathit{id} \ \mapsto \ s \ \in \ \mathit{accts} \ \Leftrightarrow \ \mathit{id} \ \mapsto \ s \ \in \ \mathit{acctsOld} \end{array}$$

Transfer

$$\begin{array}{ll} \text{Arguments}: \ \mathsf{TArgs} & \text{arguments type} \\ \quad \mathsf{getFromID} \ : \ \mathsf{TArgs} \ \rightarrow \ \mathsf{AccID} & \text{returns the ID of the account to withdraw from} \\ \quad \mathsf{getToID} \ : \ \mathsf{TArgs} \ \rightarrow \ \mathsf{AccID} & \text{returns the ID of the account to deposit into} \\ \quad \mathsf{val} \ : \ \mathsf{TArgs} \ \rightarrow \ \mathsf{Value} & \text{returns the assets to be transferred} \end{array}$$

$$\begin{array}{rl} \text{Authentication}: & \mathsf{hasSig}\ \mathit{txinfo}\ \mathit{args} \ = \ (\mathsf{getPK}\ \mathit{accts}\ \mathit{args},\ \_)\ \mathsf{in}\ \mathit{txinfo.sigs} \\ \text{Preconditions}: & \mathsf{nonNegV}\ \mathit{args} \ = \ \mathsf{val}\ \mathit{args} \ \geq \ 0 \\ & v_{from} \ := \ \mathsf{hasVFrom}\ \mathit{accts}\ \mathit{args} \ = \ \mathsf{val}\ (\mathit{accts}\ (\mathsf{getFromID}\ (\mathit{args}))) \\ & v_{to} \ := \ \mathsf{hasVTo}\ \mathit{accts}\ \mathit{args} \ = \ \mathsf{val}\ (\mathit{accts}\ (\mathsf{getToID}\ (\mathit{args}))) \\ & \mathsf{vIsEnough}\ v_{from} \ \geq \ v \\ & \mathit{acctsOld} \ := \ \mathit{accts} \\ \text{Postconditions}: & \mathsf{minusV}\ \mathit{accts}\ \mathit{args} = \ \mathsf{val}\ (\mathit{accts}\ (\mathsf{getFromID}\ (\mathit{args}))) \ == \ v_{from} \ - \ v \\ & \mathsf{plusV}\ \mathit{accts}\ \mathit{args} = \ \mathsf{val}\ (\mathit{accts}\ (\mathsf{getToID}\ (\mathit{args}))) \ == \ v_{to} \ + \ v \\ & \forall\ \mathit{id} \ \in \ \mathsf{dom}\ \mathit{accts} \ \cup \ \mathsf{dom}\ \mathit{acctsOld},\ \mathsf{getToID}\ \mathit{args} \ \neq \ \mathit{id} \ \neq \ \mathsf{getFromID}\ \mathit{args}\ , \\ & \quad \mathit{id} \ \mapsto \ s \ \in \ \mathit{accts} \ \Leftrightarrow \ \mathit{id} \ \mapsto \ s \ \in \ \mathit{acctsOld} \end{array}$$

Fig. 9: Specification of an account simulation (transitions)

$$\_ \vdash \_ \xrightarrow[\text{SMUP}]{\_} \_ \subseteq \mathbb{P}\,(\mathsf{Env} \times \mathsf{State} \times \mathsf{Input} \times \mathsf{State})$$

Fig. 10: Structured contract transition type

$$\text{StatefulStep}\ \frac{\begin{array}{c} \mathit{txInfo} := \mathsf{txInfo\ EI\ SysSt\ Lang}\ \mathit{pp}\ (\mathsf{getUTxO}\ \mathit{lState})\ \mathit{tx} \\ \mathsf{isValid}\ \mathit{tx} = \mathsf{True} \\ \begin{array}{c} \mathit{slot} \\ \mathit{txIx} \\ \mathit{pp} \\ \mathit{account} \end{array} \vdash (\,\mathit{lState}\,) \xrightarrow[\text{LEDGER}]{\mathit{tx}} (\,\mathit{lState}'\,) \end{array}}{\pi_{\mathsf{TxInfo}}\ \mathit{txInfo} \vdash (\pi_{\mathsf{State}}\ \mathit{lState}) \xrightarrow[\text{SMUP}]{\pi_{\mathsf{Input}}\ \mathit{txInfo}} (\pi_{\mathsf{State}}\ \mathit{lState}')} \tag{8}$$

$$\text{StatefulNoStep}\ \frac{\begin{array}{c} \mathit{txInfo} := \mathsf{txInfo\ EI\ SysSt\ Lang}\ \mathit{pp}\ (\mathsf{getUTxO}\ \mathit{lState})\ \mathit{tx} \\ \mathsf{isValid}\ \mathit{tx} = \mathsf{False} \\ \begin{array}{c} \mathit{slot} \\ \mathit{txIx} \\ \mathit{pp} \\ \mathit{account} \end{array} \vdash (\,\mathit{lState}\,) \xrightarrow[\text{LEDGER}]{\mathit{tx}} (\,\mathit{lState}'\,) \end{array}}{\pi_{\mathsf{TxInfo}}\ \mathit{txInfo} \vdash (\pi_{\mathsf{State}}\ \mathit{lState}) \xrightarrow[\text{SMUP}]{\pi_{\mathsf{Input}}\ \mathit{txInfo}} (\pi_{\mathsf{State}}\ \mathit{lState})} \tag{9}$$

Fig. 11: Structured contract implementation property

$$\_ \vdash \_ \xrightarrow[\text{FSMUP}]{\_} \_ \subseteq \mathbb{P}\,(\mathsf{FEnv} \times \mathsf{FState} \times \mathsf{FInput} \times \mathsf{FState})$$

$$\text{FeatureNoStep}\ \frac{\begin{array}{c} \pi_{\mathsf{FInput}}\ \mathit{inp} = \diamond \\ (\,\pi_{\mathsf{FEnv}}\ \mathit{txInfo}\,) \vdash (\pi_{\mathsf{FState}}\ \mathit{state}) \xrightarrow[\text{FSMUP}]{\diamond} (\pi_{\mathsf{FState}}\ \mathit{state}) \end{array}}{\mathit{txInfo} \vdash (\,\mathit{state}\,) \xrightarrow[\text{SMUP}]{\diamond} (\,\mathit{state}'\,)} \tag{10}$$

$$\text{FeatureStep}\ \frac{\begin{array}{c} \pi_{\mathsf{FInput}}\ \mathit{inp} \neq \diamond \\ (\,\mathit{txInfo}\,) \vdash (\pi_{\mathsf{FState}}\ \mathit{state}) \xrightarrow[\text{FSMUP}]{\pi_{\mathsf{FInput}}\ \mathit{inp}} (\pi_{\mathsf{FState}}\ \mathit{state}') \end{array}}{\mathit{txInfo} \vdash (\,\mathit{state}\,) \xrightarrow[\text{SMUP}]{\mathit{inp}} (\,\mathit{state}'\,)} \tag{11}$$

Fig. 12: Structured contract feature property

$$\_ \vdash \_ \xrightarrow[\text{SMUPS}]{\_} \_ \subseteq \mathbb{P}\,([\mathsf{Env}] \times \mathsf{State} \times [\mathsf{Input}] \times \mathsf{State})$$

Fig. 13: Structured contract transition type

$$\text{WeakStep}\ \frac{\mathsf{map}\ \pi_{\mathsf{FEnv}}\ \mathit{txInfos} \vdash (\mathit{state}) \xrightarrow[\text{SMUPS}]{\mathsf{getSteps}\ \mathit{finp}} (\mathit{state}')}{\mathit{txInfo} \vdash (\pi_{\mathsf{FState}}\ \mathit{state}) \xrightarrow[\text{FSMUP}]{\mathit{finp}} (\pi_{\mathsf{State}}\ \mathit{state}')} \tag{12}$$

Fig. 14: Feature weak implementation property

$$\_ \vdash \_ \xrightarrow[\text{MSGS}]{\_} \_ \subseteq \mathbb{P}\,(\mathsf{TxInfo} \times \mathsf{Msgs} \times \mathsf{MsgIn} \times \mathsf{Msgs})$$

Fig. 15: Message-passing transition type

Accessor functions

$\mathsf{mFrom} : \mathsf{Msg} \to \mathsf{AccID}$
ID (credential) of the sender of the message

$\mathsf{mTo} : \mathsf{Msg} \to \mathsf{AccID}$
Credential of the intended recepient of the message

$\mathsf{mData} : \mathsf{Msg} \to \mathbb{B}$
The bytestring data contents of the message

$\mathsf{mValue} : \mathsf{Msg} \to \mathsf{Value}$
The value being sent in the message

Abstract types

Msg message type

Concrete types

$\mathsf{Msgs} = [\mathsf{Msg}]$
power-multiset of messages

$\mathsf{MsgIn} = \mathsf{Msgs} \times \mathsf{Msgs}$
Messages being produced and consumed

Functions

$\mathsf{updateM} \in \mathsf{Accts} \to \mathsf{Msg} \to \mathsf{Msgs} \to \mathsf{Value}$
$\mathsf{updateM}\ cstate\ msg\ msgs = \mathsf{val}\,(cstate\,(\mathsf{mFrom}\ msg))) - \Sigma_{m \in msgs,\ \mathsf{mFrom}\ m = \mathsf{mFrom}\ msg} \mathsf{mValue}\ m$
compute the updated value in the message-sending account of *msg*

$\mathsf{updateB} \in \mathsf{Accts} \to \mathsf{Msg} \to \mathsf{Msgs} \to \mathsf{Value}$
$\mathsf{updateB}\ cstate\ msg\ msgs = (\mathsf{val}\,(cstate\,(\mathsf{mTo}\ msg))) + \Sigma_{m \in msgs,\ \mathsf{mTo}\ m = \mathsf{mTo}\ msg} \mathsf{mValue}\ m$
compute the updated value in the message-receiving account of *msg*

Fig. 16: Message passing types and functions

$$(produce, consume) := mint$$
$$mint = \pi_{MsgIn}\ \pi_{CState}\ txInfo$$

$$cstate' := cstate \underset{\rightarrow}{\cup} \{(\mathsf{mFrom}\ msg) \mapsto (s',\ \mathsf{updateM}\ cstate\ msg\ msgs) \mid msg \in msgs\}$$

$$cstate'' := cstate' \underset{\rightarrow}{\cup} \{(\mathsf{mTo}\ msg) \mapsto (s',\ \mathsf{updateB}\ cstate'\ msg\ msgs) \mid msg \in msgs\}$$

$$\text{Mint}\ \frac{\begin{array}{c} txInfo \vdash (\,cstate\,) \xrightarrow[\text{SMACC}]{\mathsf{Send}\ produce} (\,\boldsymbol{cstate'}\,) \\ txInfo \vdash (\,cstate'\,) \xrightarrow[\text{SMACC}]{\mathsf{Receive}\ consume} (\,\boldsymbol{cstate''}\,) \end{array}}{txInfo \vdash (\,msgs\,) \xrightarrow[\text{MSGS}]{mint} (\,\boldsymbol{msgs \cup produce - consume}\,)} \tag{14}$$

Fig. 17: Specification of messages

$$\begin{aligned}
\mathsf{utxoScript} &\in \mathsf{Datum} \to \mathsf{Redeemer} \to \mathsf{ScriptContext} \to \mathsf{Bool} \\
\mathsf{utxoScript} &= \mathsf{hash\ utxoScript}
\end{aligned}$$

$$\begin{aligned}
\mathsf{msgAddress} &\in \mathsf{Credential} \\
\mathsf{msgAddress} &= \mathsf{hash\ utxoScript}
\end{aligned}$$

$$\begin{aligned}
\pi_{\mathsf{Msgs}}\ \mathit{utxoSt} = \{\ & \mathit{tn} \mid _ \mapsto \mathit{out} \in (\mathsf{getUTxO}\ \mathit{utxoSt}),\ \mathsf{msgAddress} = \mathsf{addressCredential}\ (\mathsf{txOutAddress}\ \mathit{out}), \\
& (\mathsf{msgPolHash},\ \mathit{tn}) \mapsto (1) \in \mathsf{txOutValue}\ \mathit{out}\ \}
\end{aligned}$$

$$\begin{aligned}
\pi_{\mathsf{MsgIn}}\ \mathit{txinfo} = (\{\ & \mathit{rdm} \mid \mathsf{msgPolHash} \mapsto \mathit{lsRdm} \in \mathsf{txInfoRedeemers}\ \mathit{txInfo}, \\
& \mathit{rdm} \in \mathit{lsRdm},\ (\mathsf{msgPolHash},\ \mathit{rdm}) = \mathit{aid}, \\
& \mathit{aid} \mapsto q \in \mathsf{txInfoMint}\ \mathit{txinfo},\ q \geq 1\ \}, \\
& \{\ \mathit{rdm} \mid \mathsf{msgPolHash} \mapsto \mathit{lsRdm} \in \mathsf{txInfoRedeemers}\ \mathit{txInfo}, \\
& \mathit{rdm} \in \mathit{lsRdm},\ (\mathsf{msgPolHash},\ \mathit{rdm}) = \mathit{aid}, \\
& \mathit{aid} \mapsto q \in \mathsf{txInfoMint}\ \mathit{txinfo},\ q \leq (-1)\ \})
\end{aligned}$$

Fig. 18: Implementation of messages

$$\_ \vdash \_ \xrightarrow[\text{MSGS}]{\_} \_ \subseteq \mathbb{P}\ (\mathsf{TxInfo} \times \mathsf{AccState} \times \mathsf{AccMsgs} \times \mathsf{AccState})$$

Fig. 19: Account with message-passing transition type

$$\text{Transfer-From}\ \frac{\begin{array}{c}
\mathit{accIn} \in \mathsf{MsgIn} \\[4pt]
\mathsf{idFrom}\ \mathit{accIn} \mapsto \mathit{oldAcctFrom} \in \mathit{accts} \qquad \mathit{pkFrom} := \mathsf{pk}\ (\mathit{idFrom},\ \mathit{oldAcctFrom}) \\[4pt]
\mathsf{val}\ \mathit{oldAcctFrom} \geq \mathsf{val}\ \mathit{accIn} \geq \mathsf{zero} \\
\mathsf{val}\ \mathit{changedAcctFrom} = \mathsf{val}\ \mathit{oldAcctFrom} - \mathsf{val}\ \mathit{accIn} \\[4pt]
\mathsf{pk}\ (\mathsf{idFrom}\ \mathit{accIn},\ \mathit{changedAcctFrom}) = \mathit{pkFrom} \\
\mathit{pkFrom} \in \mathsf{txInfoSignatories}\ \mathit{txInfo} \\[4pt]
\mathit{txInfo} \vdash \mathit{msgs} \xrightarrow[\text{MSGS}]{\mathit{accIn}} \mathit{msgs}'
\end{array}}{\mathit{txInfo} \vdash \left(\mathit{accts}\right) \xrightarrow[\text{ACCNT}]{\mathit{accIn}} \left(\boldsymbol{\mathit{accts}} \underset{\rightarrow}{\cup} \{\boldsymbol{\mathit{idFrom}} \mapsto \boldsymbol{\mathit{changedAcctFrom}}\}\right)} \quad (15)$$

Fig. 20: Specification of account simulation transfer via message-passing (transfer-from)

$$\text{Transfer-To}\ \frac{\begin{array}{c}
\mathit{accIn} \in \mathsf{TFArgs} \\[4pt]
\mathsf{idTo}\ \mathit{accIn} \mapsto \mathit{oldAcctTo} \in \mathit{accts} \qquad \mathit{pkTo} := \mathsf{pk}\ (\mathit{idTo},\ \mathit{oldAcctTo}) \\[4pt]
\mathsf{val}\ \mathit{accIn} \geq \mathsf{zero} \\
\mathsf{val}\ \mathit{changedAcctTo} = \mathsf{val}\ \mathit{oldAcctTo} + \mathsf{val}\ \mathit{accIn} \\[4pt]
\mathsf{pk}\ (\mathsf{idTo}\ \mathit{accIn},\ \mathit{changedAcctTo}) = \mathit{pkTo} \\
\mathit{pkTo} \in \mathsf{txInfoSignatories}\ \mathit{txInfo} \\[4pt]
\mathit{txInfo} \vdash \mathit{msgs} \xrightarrow[\text{MSGS}]{\mathit{accIn}} \mathit{msgs}'
\end{array}}{\mathit{txInfo} \vdash \left(\mathit{accts}\right) \xrightarrow[\text{ACCNT}]{\mathit{accIn}} \left(\boldsymbol{\mathit{accts}} \underset{\rightarrow}{\cup} \{\boldsymbol{\mathit{idTo}} \mapsto \boldsymbol{\mathit{changedAcctTo}}\}\right)} \quad (16)$$

Fig. 21: Specification of account simulation transfer via message-passing (transfer-from)

$$\text{UpdateWallets}\;\frac{\begin{array}{c} \mathit{ins} := [\,\mathsf{txInInfoResolved}\ \mathit{ri} \mid \mathit{ri} \leftarrow \mathsf{txInfoInputs}\ \mathit{txInfo}\,] \\ \mathit{outs} := (\mathsf{txInfoOutputs}\ \mathit{txInfo}) \\[1em] \forall\,(\mathsf{msgPolHash} \mapsto (\mathsf{getPreim}\ \mathit{txInfo}\ \mathit{msg}) \mapsto 1) \in \mathsf{txInfoMint}\ \mathit{txInfo},\ \mathsf{mFrom}\ \mathit{msg} \in \mathsf{PubKey}, \\ \mathsf{mFrom}\ \mathit{msg} \in \mathsf{txInfoSignatories}\ \mathit{txInfo} \\[1em] \forall\,(\mathsf{msgPolHash} \mapsto (\mathsf{getPreim}\ \mathit{txInfo}\ \mathit{msg}) \mapsto -1) \in \mathsf{txInfoMint}\ \mathit{txInfo},\ \mathsf{mTo}\ \mathit{msg} \in \mathsf{PubKey}, \\ \mathsf{mTo}\ \mathit{msg} \in \mathsf{txInfoSignatories}\ \mathit{txInfo} \\[1em] \mathit{accts}' := \mathit{accts} \cup_{+} \{\, \mathit{pk} \mapsto \Sigma_{((\mathit{pk},\_),v,\_) \in \mathit{outs}} v \mid ((\mathit{pk},\_),\_,\_) \in \mathit{outs} \,\} \\ \cup_{-} \{\, \mathit{pk} \mapsto \Sigma_{((\mathit{pk},\_),v,\_) \in \mathit{ins}} v \mid ((\mathit{pk},\_),\_,\_) \in \mathit{ins} \,\} \end{array}}{\mathit{txInfo} \vdash (\,\mathit{accts}\,) \xrightarrow[\text{ACCNT}]{\mathit{accIn}} (\,\boldsymbol{\mathit{accts}'}\,)} \tag{17}$$

Fig. 22: Specification of wallet accounts update

Types

$$\mathsf{AccState} := \mathsf{PubKey} \mapsto \mathsf{Value}$$

Functions

$\pi_{\mathsf{AccState}} \in \mathsf{UTxOState} \to \mathsf{AccState}$

$\pi_{\mathsf{AccState}}\ \mathit{utxoSt} = \{\mathit{pk} \mapsto (\Sigma_{i \mapsto ((\mathit{pk},\_),v,\_) \in \mathsf{getUTxO}\ (\mathit{utxo})} v) \mid i \mapsto ((\mathit{pk},\_),v,\_) \in \mathsf{getUTxO}\ (\mathit{utxo})\}$

Projects the wallet (public key) account state from the UTxO state

$\pi_{\mathsf{SimplInput}} \in \mathsf{TxInfo} \to \mathsf{TxInfo}$

$\pi_{\mathsf{SimplInput}} := \mathsf{id}_{\mathsf{txInfo}}$

Projection from TxInfo to wallet accounts input is the identity

$\pi_{\mathsf{MsgIn}} \in \mathsf{TxInfo} \to \mathsf{MsgIn}$

$\pi_{\mathsf{MsgIn}} := ([\mathit{msg} \mid (\mathsf{msgPolHash} \mapsto \mathsf{hash}\ \mathit{msg} \mapsto 1) \leftarrow \mathsf{txInfoMint}\ \mathit{txInfo}, \mathsf{mFrom}\ \mathit{msg} \in \mathsf{PubKey}],$
$\quad [\mathit{msg} \mid (\mathsf{msgPolHash} \mapsto \mathsf{hash}\ \mathit{msg} \mapsto -1) \leftarrow \mathsf{txInfoMint}\ \mathit{txInfo}, \mathsf{mTo}\ \mathit{msg} \in \mathsf{PubKey}])$

Projection from TxInfo to wallet sender/receiver messages

$\mathsf{getPreim} \in \mathsf{TxInfo} \to \mathbb{B} \to \mathsf{TokenName}^?$

$$\mathsf{getPreim}\ \mathit{txInfo}\ h = \begin{cases} \mathit{msg} & \text{if } h \mapsto \mathit{msg} \in \mathsf{txInfoRedeemers}\ \mathit{txInfo} \\ \diamond & \text{otherwise} \end{cases}$$

Returns the preimage of a hash of a message from the redeemer of the message minting policy

$\mathsf{getMint} \in \mathsf{TxInfo} \to \mathsf{MsgIn}$

$\mathsf{getMint}\ \mathit{txi} = ([\,\mathsf{getPreim}\ \mathsf{txInfo}\ \mathit{tn} \mid (\mathit{tn}, 1) \leftarrow \mathsf{toList}\ ((\mathsf{txInfoMint}\ \mathit{txi})\ \mathit{msgPolHash})\,],$
$\quad [\,\mathsf{getPreim}\ \mathsf{txInfo}\ \mathit{tn} \mid (\mathit{tn}, -1) \leftarrow \mathsf{toList}\ ((\mathsf{txInfoMint}\ \mathit{txi})\ \mathit{msgPolHash})\,])$

Returns the collection of message tokens being produced and consumed by the transaction

Fig. 23: Wallet accounts functions

$$
\begin{aligned}
\mathsf{State} &= \mathsf{PubKey} \mapsto \mathsf{Value} \\
\mathsf{Input} &= \mathsf{PubKey} \mapsto \mathsf{Value} \\
\mathsf{State} &= \mathsf{PubKey} \mapsto \mathsf{Value} \\
\mathsf{buildConstraits} &= \\
\mathsf{utxoNFT} &= \text{can be any unspent output reference}
\end{aligned}
$$

Fig. 24: Naive implementation account simulation instance

$$
\_ \vdash \_ \xrightarrow[\text{SMECH}]{\_} \_ \subseteq \mathbb{P}\ (\mathsf{TxInfo} \times \mathsf{State} \times \mathsf{Input} \times \mathsf{State})
$$

$$
\_ \vdash \_ \xrightarrow[\text{SMSPEC}]{\_} \_ \subseteq \mathbb{P}\ (\mathsf{Context} \times \mathsf{State}' \times \mathsf{Input}' \times \mathsf{State}')
$$

$$
\text{MechanismSim}\frac{\mathit{txInfo} \vdash \pi_s\ s \xrightarrow[\text{SMPEC}]{\pi_i\ i} \boldsymbol{\pi_s\ s'}}{\mathit{txInfo} \vdash s \xrightarrow[\text{SMECH}]{i} \boldsymbol{s'}} \tag{18}
$$

Fig. 25: Stateful mechanism property

NFTCE projection functions and types

$$\pi_{\mathsf{State}} \in \mathsf{UTxOState} \to \mathsf{State}^?$$

$$\pi_{\mathsf{State}}\ utxoSt = \begin{cases} \mathsf{head}\ (\mathsf{toList}\ smDatVal) & \text{when } smDatVal \neq \Diamond \\ \Diamond & \text{otherwise} \end{cases}$$

where

$smDatVal := \{\ (d,\ v)\ \mid\ ([\![\mathsf{utxoNFTPolicy}\ \mathsf{utxoNFT}]\!],\ vh) \mapsto \_ \ \in\ \mathsf{txOutValue}\ t,\ \_ \mapsto\ t\ \in\ (\mathsf{getUTxO}\ u$

$v = \mathsf{txOutValue}\ t\ -\ (([\![\mathsf{utxoNFTPolicy}\ \mathsf{utxoNFT}]\!],\ vh) \mapsto 1)\ \}$

Returns the state of the SM being implemented

$$\pi_{\mathsf{Input}} \in \mathsf{TxInfo} \to \mathsf{Input}^?$$

$$\pi_{\mathsf{Input}}\ txInfo = \begin{cases} \mathsf{head}\ (\mathsf{toList}\ smRedeemer) & \text{when } smRedeemer \neq \Diamond \\ \mathsf{Initialize} & \text{when } nftMint\ =\ 1 \\ \Diamond & \text{otherwise} \end{cases}$$

where

$smRedeemer := \{\ r\ \mid\ ([\![\mathsf{utxoNFTPolicy}\ \mathsf{utxoNFT}]\!],\ \_) \mapsto \_\ \in\ \mathsf{txOutValue}\ t,$

$i \mapsto\ t\ \in\ (\mathsf{getUTxO}\ utxoSt),\ i \mapsto\ r\ \in \mathsf{txInfoRedeemers}\ txInfo\ \}$

$nftMint\ :=\ (\mathsf{txInfoMint}\ txInfo)\ [\![\mathsf{utxoNFTPolicy}\ \mathsf{utxoNFT}]\!]$

Returns the input of the SM being implemented

$(\mathsf{NFTSMState} \times\ \mathsf{Value})^? := \mathsf{State}$

The state of the SM being implemented

$\mathsf{NFTSMInput} := \mathsf{Input}$

The input of the SM being implemented

$$\mathsf{utxoNFTPolicy} \in \mathsf{TxOutRef} \to \mathsf{PolicyID}$$

$$\mathsf{utxoNFTPolicy}\ txin\ context\ rdmr = \begin{cases} \mathsf{True} & \text{when } (txin,\ \_) \in \mathsf{txInfoInputs}\ (\mathsf{scriptContextTxInfo}\ context) \\ \mathsf{False} & \text{otherwise} \end{cases}$$

The input used to generate the unique thread token

NFTCE abstract functions and parameter values

$\mathsf{buildConstraits} \in \mathsf{TxInfo} \to \mathbb{P}\ \mathsf{cnstr} \in \mathsf{State} \times \mathsf{Input} \to \mathsf{TxConstraint}$

Function returning emitted contraints for the state machine

$\mathsf{utxoNFT} \in \mathsf{TxOutRef}$

The input used to generate the unique thread token

$\mathsf{initState} \in \mathsf{State}$

Initial state of the SM

Transition type for SM being implemented by NFTCE

$$\_ \vdash \_ \xrightarrow[\text{SMSPEC}]{\_} \_ \subseteq \mathbb{P}\ (\mathsf{TxInfo} \times (\mathsf{NFTSMState} \times\ \mathsf{Value}) \times \mathsf{NFTSMInput} \times (\mathsf{NFTSMState} \times\ \mathsf{Value}))$$

Fig. 26: Specification of an account simulation (functions and types)

$$\text{DoesNothingNFT}\ \frac{smi = \diamond}{txInfo \vdash smsval \xrightarrow[\text{NFTCE}]{smi} \begin{array}{c} sms \\ val \end{array}} \tag{19}$$

$$\text{MintsNFT}\ \frac{\begin{array}{c} smi \in \mathsf{Initialize} \\ \text{puts thread token NFT into UTxO with correct validator} \end{array}}{txInfo \vdash (\diamond) \xrightarrow[\text{NFTCE}]{smi} \begin{array}{c} \mathbf{\mathsf{fst\ initState}} \\ \mathbf{\mathsf{snd\ initState}} \end{array}} \tag{20}$$

$$\text{PropagatesNFT}\ \frac{\begin{array}{c} \forall p \in \mathsf{buildConstraits}\ txInfo,\ p\ sms\ smi \\ \text{propagates thread token into correct output} \\ txInfo \vdash \begin{array}{r} sms \\ val \end{array} \xrightarrow[\text{SMPEC}]{smi} \begin{array}{c} \boldsymbol{sms'} \\ \boldsymbol{val'} \end{array} \end{array}}{txInfo \vdash \begin{array}{r} sms \\ val \end{array} \xrightarrow[\text{NFTCE}]{smi} \begin{array}{c} \boldsymbol{sms'} \\ \boldsymbol{val'} \end{array}} \tag{21}$$

Fig. 27: Single-UTxO state machine implementation via NFTs