Native Custom Tokens in the Extended UTXO Model

Manuel M.T. Chakravarty[1], James Chapman[1], Kenneth MacKenzie[1], Orestis Melkonian[1,2], Jann Müller[1], Michael Peyton Jones[1], Polina Vinogradova[1], and Philip Wadler[1,2]

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

Abstract. *User-defined tokens* —both fungible ERC-20 and non-fungible ERC-721 tokens— are central to the majority of contracts deployed on Ethereum. User-defined tokens are *non-native* on Ethereum; i.e., they are not directly supported by the ledger, but require custom code. This makes them unnecessarily inefficient, expensive, and complex.

The *Extended UTXO Model* (*EUTXO*) [5] has been introduced as a generalisation of Bitcoin-style UTXO ledgers, allowing support of more expressive smart contracts, approaching the functionality available to contracts on Ethereum. Specifically, a bisimulation argument established a formal relationship between the EUTXO ledger and a general form of state machines. Nevertheless, *transaction outputs* in the EUTXO model lock integral quantities of a single native cryptocurrency only, just like Bitcoin.

In this paper, we study a generalisation of the EUTXO ledger model with *native* user-defined tokens. Following the approach proposed in a companion paper [4] for the simpler case of plain Bitcoin-style UTXO ledgers, we generalise transaction outputs to lock not merely coins of a single cryptocurrency, but entire *token bundles,* including custom tokens whose forging is controlled by *forging policy scripts.* We show that this leads to a rich ledger model that supports a broad range of interesting use cases.

Our main technical contribution is a formalisation of the multi-asset EUTXO ledger in Agda, which we use to establish that the ledger with custom tokens is strictly more expressive than the original EUTXO ledger. In particular, we state and prove a transfer result for inductive and temporal properties from state machines to the multi-asset EUTXO ledger, which was out of scope for the single-currency EUTXO ledger. In practical terms, the resulting system is the basis for the smart contract system of the Cardano blockchain.

Keywords: blockchain · UTXO · tokens · functional programming · state machines · bisimulation

1 Introduction

If we look at contracts on Ethereum in terms of their use and the monetary values that they process, then it becomes apparent that so-called *user-defined* (or *custom*) *tokens* play a central role in that ecosystem. The two most common token types are *fungible tokens*, following the ERC-20 standard [17], and *non-fungible* tokens, following the ERC-721 standard [7].

On Ethereum, ERC-20 and ERC-721 tokens are fundamentally different from the native cryptocurrency, Ether, in that their creation and use always involves user-defined custom code — they are not directly supported by the underlying ledger, and hence are *non-native*. This makes them unnecessarily inefficient, expensive, and complex. Although the ledger already includes facilities to manage and maintain a currency, this functionality is replicated in interpreted user-level code, which is inherently less efficient. Moreover, the execution of user-code needs to be paid for (in gas),

which leads to significant costs. Finally, the ERC-20 and ERC-721 token code gets replicated and adapted, instead of being part of the system, which complicates the creation and use of tokens and leaves room for human error.

The alternative to user-level token code is a ledger that supports *native tokens*. In other words, a ledger that directly supports (1) the creation of new user-defined token or asset types, (2) the forging of those tokens, and (3) the transfer of custom token ownership between multiple participants. In a companion paper [4], we propose a generalisation of Bitcoin-style UTXO ledgers, which we call UTXO_{ma}("ma" for "multi-asset"), adding native tokens by way of so-called *token bundles* and domain-specific *forging policy scripts*, without the need for a general-purpose scripting system.

Independently, we previously introduced the *Extended UTXO Model* (EUTXO) [5] as an orthogonal generalisation of Bitcoin-style UTXO ledgers, enabling support of more expressive smart contracts, with functionality similar to contracts on Ethereum. To support user-defined tokens or currencies on EUTXO, we could follow Ethereum's path and define standards corresponding to ERC-20 and ERC-721 for fungible and non-fungible tokens, but then we would be subject to the same disadvantages that non-native tokens have on Ethereum.

In this paper, to avoid the disadvantages of non-native tokens, we investigate the combination of the two previously mentioned extensions of the plain UTXO model: we add UTXO_{ma}-style token bundles and asset policy scripts to the EUTXO model, resulting in a new EUTXO_{ma} ledger model.

We will show that the resulting EUTXO_{ma} model is strictly more expressive than both EUTXO and UTXO_{ma} by itself. In particular, the constraint-emitting state machines in [5] can not ensure that they are initialised correctly. In EUTXO_{ma}, we are able to use non-fungible tokens to trace *threads* of state machines. Extending the mechanised Agda model from [5] allows us to then prove inductive and temporal properties of state machines by induction over their traces, covering a wide variety of state machine correctness properties.

Moreover, the more expressive scripting functionality and state threading of EUTXO enables us to define more sophisticated asset policies than in UTXO_{ma}. Additionally, we argue that the combined system allows for sophisticated access-control schemes by representing roles and capabilities in the form of user-defined tokens.

In summary, this paper makes the following contributions:

- We introduce the multi-asset EUTXO_{ma} ledger model (§2).
- We outline a range of application scenarios that are arguably better supported by the new model, and also applications that plain UTXO_{ma} does not support at all (§3).
- We formally prove a transfer result for inductive and temporal properties from constraint emitting machines to the EUTXO_{ma} ledger, an important property that we were not able to establish for plain EUTXO (§4).

We discuss related work in §5. Due to space constraints, the formal ledger rules for EUTXO_{ma} are in Appendix A. A mechanised version of the ledger rules and the various formal results from §4 is available as Agda source code.[3]

On top of the conceptual and theoretical contributions made in this paper, we would like to emphasise that the proposed system is highly practical. In fact, EUTXO_{ma} underlies our implementation of *Plutus Platform,* the smart contract system of the Cardano blockchain.[4]

[3] https://github.com/omelkonian/formal-utxo/tree/2d32
[4] https://github.com/input-output-hk/plutus

2 Extending Extended UTXO

Before discussing applications and the formal model of EUTXO_{ma}, we briefly summarise the structure of EUTXO, and then informally introduce the multi-asset extension that is the subject of this paper. Finally, we will discuss a shortcoming in the state machine mapping of EUTXO as introduced in [5] and illustrate how the multi-asset extension fixes that shortcoming.

2.1 The starting point: Extended UTXO

In Bitcoin's UTXO ledger model, the ledger is formed by a list of transactions grouped into blocks. As the block structure is secondary to the discussion in this paper, we will disregard it in the following. A transaction tx is a quadruple (I, O, r, S) comprising a set of *inputs* I, a list of *outputs* O, a *validity interval* r, and a set of signatures S, where inputs and outputs represent cryptocurrency value flowing into and out of the transaction, respectively. The sum of the inputs must be equal to the sum of the outputs; in other words, transactions preserve value. Transactions are identified by a collision-resistant cryptographic hash h computed from the transaction.[5]

An input $i \in I$ is represented as a pair (out_{ref}, ρ) of an *output reference* out_{ref} and a *redeemer value* ρ. The output reference out_{ref} uniquely identifies an output in a preceding transaction by way of the transaction's hash and the output's position in the transaction's list of outputs.

In plain UTXO, an output $o \in O$ is a pair of a *validator script* ν and cryptocurrency value *value*. In the Extended UTXO model (EUTXO) [5], outputs become triples $(\nu, value, \delta)$, where the added *datum* δ enables passing additional information to the validator.

The purpose of the validator is to assess whether an input i of a subsequent transaction trying to spend (i.e., consume) an output o should be allowed to do so. To this end, we execute the validator script to check whether $\nu(\rho, \delta, \sigma) = \mathsf{true}$ holds. Here σ comprises additional information about the *validation context* of the transaction. In the plain UTXO model that contextual information is fairly limited: it mainly consists of the validated transaction's hash, the signatures S, and information about the length of the blockchain. In the EUTXO model, we extend σ to include the entirety of the validated transaction tx as well as all the outputs spent by the inputs of tx.

2.2 Token bundles

In UTXO and EUTXO, the *value* carried by an output is represented as an integral value denoting a specific quantity of the ledger's native cryptocurrency. As discussed in more detail in the companion paper [4], we can generalise *value* to carry a two-level structure of *finitely-supported functions.* The technicalities are in Appendix A; for now, we can regard them as nested finite maps to quantities of tokens. For example, the value $\{\mathsf{Coin} \mapsto \{\mathsf{Coin} \mapsto 3\}, g \mapsto \{t_1 \mapsto 1, t_2 \mapsto 1\}\}$ contains 3 Coin coins (there is only one (fungible) token Coin for a payment currency also called Coin), as well as (non-fungible) tokens t_1 and t_2, both in asset group g. Values can be added naturally, e.g.,

$$\begin{aligned} &\{\mathsf{Coin} \mapsto \{\mathsf{Coin} \mapsto 3\}, g \mapsto \{t_1 \mapsto 1, t_2 \mapsto 1\}\} \\ +\ &\{\mathsf{Coin} \mapsto \{\mathsf{Coin} \mapsto 1\}, g \mapsto \{t_3 \mapsto 1\}\} \\ =\ &\{\mathsf{Coin} \mapsto \{\mathsf{Coin} \mapsto 4\}, g \mapsto \{t_1 \mapsto 1, t_2 \mapsto 1, t_3 \mapsto 1\}\}\ . \end{aligned}$$

In a bundle, such as $g \mapsto \{t_1 \mapsto 1, t_2 \mapsto 1, t_3 \mapsto 1\}$, we call g an *asset group* comprising a set of tokens t_1, t_2, and t_3. In the case of a fungible asset group, such as Coin, we may call it a *currency.* In contrast to fungible tokens, only a single instance of a non-fungible token may be minted.

[5] We denote the hash of some data d as $d^{\#}$.

2.3 Forging custom tokens

To enable the introduction of new quantities of new tokens on the ledger (*minting*) or the removal of existing tokens (*burning*), we add a *forge field* to each transaction. The use of the forge field needs to be tightly controlled, so that the minting and burning of tokens is guaranteed to proceed according to the token's *forging policy*. We implement forging policies by means of scripts that are much like the validator scripts used to lock outputs in EUTXO.

Overall, a transaction in EUTXO_{ma} is thus a sextuple $(I, O, r, \mathit{forge}, \mathit{fpss}, S)$, where *forge*, just like *value* in an output, is a token bundle and *fpss* is a set of *forging policy scripts*. Unlike the value attached to transaction outputs, *forge* is allowed to contain negative quantities of tokens. Positive quantities represent minted tokens and negative quantities represent burned tokens. In either case, any asset group ϕ that occurs in *forge* (i.e., $\mathit{forge} = \{\ldots, \phi \mapsto \mathit{toks}, \ldots\}$) must also have its forging policy script in the transaction's *fpss* field. Each script π in *fpss* is executed to check whether $\pi(\sigma) = \mathsf{true}$, that is whether the transaction, including its *forge* field, is admissible. (In fact we have a slightly different σ here, which we elaborate on in Appendix A.)

2.4 Constraint emitting machines

In the EUTXO paper [5], we explained how we can map *Constraint Emitting Machines* (CEMs) —a variation on Mealy machines— onto an EUTXO ledger. A CEM consists of its type of states S and inputs I, predicates $\mathsf{initial}, \mathsf{final} : \mathsf{S} \to \mathbb{B}$ indicating which states are initial and final, respectively, and a valid set of transitions, given as a partial function $\mathsf{step} : \mathsf{S} \to \mathsf{I} \to \mathsf{Maybe}\ (\mathsf{S} \times \mathsf{TxConstraints})$ from source state and input symbol to target state and constraints. (The result may be $\mathsf{Nothing}$, in case no valid transitions exist from a given state/input.) One could present CEMs using the traditional five-tuple notation $(Q, \Sigma, \delta, q_0, F)$, but we opt for functional notation to avoid confusion with standard *finite state machines* (see [5] on how CEMs differ from FSMs), as well as being consistent with our other definitions.

A sequence of CEM state transitions, each of the form $s \xrightarrow{i} (s', tx^{\equiv})$, is mapped to a sequence of transactions, where each machine state s is represented by one transaction tx_s. Each such transaction contains a *state machine output* o_s whose validator ν_s implements the CEM transition relation and whose datum δ_s encodes the CEM state s.

The transition $tx_{s'}$, representing the successor state, spends o_s with an input that provides the CEM input i as its redeemer ρ_i. Finally, the constraints $tx^{\equiv}$ generated by the state transition need to be met by the successor transition $tx_{s'}$. (We will define the correspondence precisely in §4.)

A simple example for such a state machine is an on-chain n–of–m multi-signature contract. Specifically, we have a given amount $\mathit{value}_{\text{msc}}$ of some cryptocurrency and we require the approval of at least n out of an *a priori* fixed set of $m \geq n$ owners to spend $\mathit{value}_{\text{msc}}$. With plain UTXO (e.g., on Bitcoin), a multi-signature scheme requires out-of-band (off-chain) communication to collect all n signatures to spend $\mathit{value}_{\text{msc}}$. On Ethereum, and also in the EUTXO model, we can collect the signatures on-chain, without any out-of-band communication. To do so, we use a state machine operating according to the transition diagram in Figure 1, where we assume that the threshold n and authorised signatures $\mathit{sigs}_{\text{auth}}$ with $|\mathit{sigs}_{\text{auth}}| = m$ are baked into the contract code.

In the multi-signature state machine's implementation in the EUTXO model, we use a validator function ν_{msc} accompanied by the datum δ_{msc} to lock $\mathit{value}_{\text{msc}}$. The datum δ_{msc} stores the machine state, which is of the form $\mathsf{Holding}$ when only holding the locked value or $\mathsf{Collecting}(\mathit{value}, \kappa, d, \mathit{sigs})$ when collecting signatures *sigs* for a payment of *value* to κ by the deadline d. The initial output for the contract is $(\nu_{\text{msc}}, \mathit{value}_{\text{msc}}, \mathsf{Holding})$.

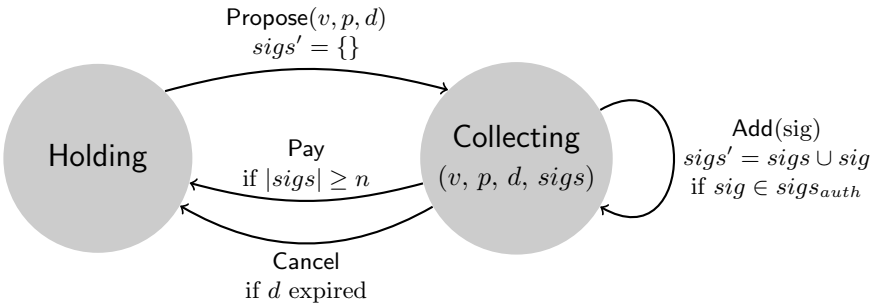

Fig. 1: Transition diagram for the multi-signature state machine; edges labelled with input from redeemer and transition constraints.

The validator ν_{msc} implements the state transition diagram from Figure 1 by using the redeemer of the spending input to determine the transition that needs to be taken. That redeemer (state machine input) can take four forms: (1) $\mathsf{Propose}(value, \kappa, d)$ to propose a payment of $value$ to κ by the deadline d, (2) $\mathsf{Add}(sig)$ to add a signature sig to a payment, (3) Cancel to cancel a proposal after its deadline expired, and (4) Pay to make a payment once all required signatures have been collected. It then validates that the spending transaction tx is a valid representation of the newly reached machine state. This implies that tx needs to keep $value_{\text{msc}}$ locked by ν_{msc} and that the state in the datum δ'_{msc} needs to be the successor state of δ_{msc} according to the transition diagram.

While the state machine in Figure 1 is fine, its mapping to EUTXO transactions comes with a subtle caveat: what if, for a 2–of–3 contract, somebody posts a transition tx_{bad} corresponding to the state $\mathsf{Collecting}(value, \kappa, d, \{sig_1, sig_2\})$ onto the chain *without* going through any of the previous states, including without starting in $\mathsf{Holding}$? Given that Pay merely checks $|sigs| \geq 2$, the payment would be processed when requested on tx_{bad}, even if $\{sig_1, sig_2\}$ are invalid signatures. We would not have been allowed to add the invalid signatures sig_1 and sig_2 by way of $\mathsf{Add}(sig)$ since its state transition checks signature validity, but by initialising the state machine in an intermediate state s with $\mathsf{initial}(s) = \mathsf{false}$, we were able to circumvent that check.

In the plain EUTXO model, there is no simple way for the validator implementing a state machine to assert that the state it is operating on arose out of a succession of predecessor states rooted in an initial state. As a consequence of this limitation, the formal result presented in the EUTXO paper [5] is not as strong as one might hope. More precisely, this previous work did establish soundness and completeness for a weak bisimulation between CEMs and transactions on a EUTXO ledger; however, it fell short in that it did not show that an inductive property met by the states of a CEM may also be asserted for the corresponding states on the ledger. The reason for this omission is precisely the problem we just discussed: for a ledger-representation of a CEM state, we can never be sure that it arose out of a transaction sequence rooted in an initial CEM state.

2.5 Token provenance

In the present work, we solve the problem described above. We will show that non-fungible tokens can be used to identify a unique trace of transactions through an EUTXO_{ma} ledger that corresponds to all the states of the corresponding CEM. Moreover, we introduce a notion of *token provenance* that enables us to identify the first CEM state in such a trace and to ensure that it is a state

accepted by $\mathsf{initial} : \mathsf{S} \to \mathbb{B}$. Together, these allow a state machine validator to ensure that a current CEM state indeed arose out of a trace of transactions rooted in a transaction corresponding to an initial CEM state merely by checking that the non-fungible token associated with this state machine instance is contained in the *value* of the state machine output that it locks: this would be $value_{\text{msc}}$ for the multi-signature contract of §2.4.

3 Applications

The functionality and applications discussed in the companion paper [4] on UTXO_{ma} are all still possible in the extended EUTXO_{ma} model discussed here. In fact, some of the applications can even be realised more easily by using the added EUTXO functionality, for example by using a state machine to control the behaviour of a forging policy over time. On top of that, we can realise new applications and move some of the functionality that needed to be implemented by an off-chain trusted party in UTXO_{ma} into on-chain script code in EUTXO_{ma}.

3.1 State thread tokens

As discussed in §2, many interesting smart contracts can be implemented by way of state machines. However, without custom tokens, the implementation of state machines on an EUTXO ledger suffers from two problems:

- Multiple instances of a single state machine (using the same validator code) generate state machine outputs that look alike. Hence, for a given output locked by the state machine's validator, we cannot tell whether it belongs to one or another run of that state machine.
- The issue raised in §2.4: a validator cannot tell whether the state machine was started in one of its initial states or whether somebody produced out of thin air a state machine in the middle of its execution.

We can work around the first problem by requiring all transactions of a particular state machine instance to be signed by a particular key and checking that as part of state machine execution. This is awkward, as it requires off-chain communication of the key for a multi-party state machine. In any case these two problems open contracts up to abuse if they are not carefully implemented.

We can solve both of these problems with the help of a unique non-fungible token, called a *state thread token,* which is minted in an initial CEM state. From the initial state on, it uniquely identifies the transaction sequence implementing the progression of one specific instance of the token's state machine. The state thread token is passed from one transition to the next by way of the state machine output of each transaction, where the presence of the state thread token is asserted by the state machine validator. More precisely, assuming a validator ν_s implementing the state machine and a matching forging policy ϕ_s, the state thread token of the ϕ_s policy is used as follows:

- The forging policy ϕ_s checks that
  - the transaction mints a unique, non-fungible token tok_s and
  - the transaction's state machine output is locked by the state machine validator ν_s and contains an admissible initial state of the state machine in its datum field.
- The state machine validator ν_s, in turn, asserts that
  - the standard state machine constraints hold,
  - the value locked by ν_s contains the state thread token tok_s, and

- if the spending transaction represents a final state, it burns tok_s. (We ignore burning in the formalisation in §4. It is for cleaning up and not relevant for correctness.)

This solves both problems. The uniqueness of the non-fungible token ensures that there is a single path that represents the progression of the state machine. Moreover, as the token can only be minted in an initial state, it is guaranteed that the state machine must indeed have begun in an initial state. In §4 we formalise this approach and we show that it is sufficient to give us the properties that we want for state machines.

3.2 Tokenised roles and contracts

Custom tokens are convenient to control ownership and access to smart contracts. In particular, we can turn *roles* in a contract into tokens by forging a set of non-fungible tokens, one for each role in the contract. In order to take an action (for example, to make a specific state transition in a state machine) as that role, a user must present the corresponding *role token*.

For example, in a typical financial futures contract we have two roles: the buyer (long), and the seller (short). In a tokenised future contract, we forge tokens which provide the right to act as the long or short position; e.g., in order to settle the future and take delivery of the underlying asset, an agent needs to provide the long token as part of that transaction.

Trading role tokens. Tokenised roles are themselves *resources* on the ledger, and as such are tradeable and cannot be split or double spent, which is useful in practice. For example, it is fairly typical to trade in futures contracts, buying or selling the right to act as the long or short position in the trade. With tokenised roles, this simply amounts to trading the corresponding role token.

The alternative approach is to identify roles with public keys, where users *authenticate* as a role by signing the transaction with the corresponding key. However, this makes trading roles much more cumbersome than simply trading a token; an association between roles and keys could be stored in the contract datum, but this requires interacting with the contract to change the association. Moreover, when using public keys instead of role tokens, we cannot easily implement more advanced use cases that require treating the role as a resource, such as the derivatives discussed below.

The lightweight nature of our tokens is key here. To ensure that roles for all instances of all contracts are distinct, every instance requires a unique asset group with its own forging policy. If we had a global register with all asset groups and token names, adding a new asset group and token for every set of role tokens we create would add significant overhead.

Derivatives and securitisation. Since tokenised roles are just tokens, they are themselves assets that can be managed by smart contracts. This enables a number of derivative (higher-order) financial contracts to be written in a generic way, without requiring a hard-coded list of tokens as the underlying assets.

For example, consider interest, a contract from which payments can be extracted at regular intervals, based on some interest rate. We can tokenise this by issuing a token for the *creditor* role that represents the claim to those payments: if someone wishes to claim the payment then they must present that token.

If we have several instances of interest (perhaps based on different interest rates), we can lock their creditor tokens in a new contract that bundles the cash flows of all the underlying contracts. The payments of this new contract are the sum of the payments of the interest contracts that it

collects. The tokens of the interest contracts cannot be traded separately as long as they are locked by the new contract.

This kind of bundling of cash flows is called *securitisation*. Securitisation is commonly used to even out variations in the payment streams of the underlying contracts and to distribute their default risk across different risk groups (tranches). Derivative contracts, including the example above, make it possible to package and trade financial risks, ultimately resulting in lower expenditure and higher liquidity for all market participants. Our system makes the cost of creating derivatives very low, indeed no higher than making a contract that operates on any other asset.

3.3 Fairness

Tokens can be used to ensure that *all* participants in some agreement have been involved. For example, consider a typical ICO setup in which a number of participants pay for the right to buy the token during an initial issuance phase. A naive implementation is as follows:

- Contributors lock their contributions with a validator ν and a datum δ, where δ contains their public key and ν requires that an appropriate part of the forged tranche be sent to the public key address corresponding to δ.
- The forging policy ϕ requires that the sum of the inputs exceeds a pre-determined threshold T, and allows forging of n units of the token, which must be allocated to the input payers in proportion to their contribution.

Unfortunately, this is not *fair*, in that participants can be omitted by the party who actually creates the forging transaction, as long as the other participants between them reach the threshold.

We can fix this by issuing (fungible) *participation tokens*, representing the right to participate in the ICO. First, we forge l participation tokens, and then we distribute them to the participants (in return for whatever form of payment we require). Finally, in order to issue the main tranche of tokens, we require (in addition to the previous conditions) that some appropriate fraction of the issued participation tokens are spent in that transaction. That means we cannot omit (too many of) the holders of the participation tokens — and the forging policy ensures that all participants are compensated appropriately. As a bonus, this makes the right to participate in the ICO itself tradeable as a tokenised role. In other words, participation tokens make roles into *first-class* assets.

A similar scheme is being used in the Hydra head protocol [6], which implements a layer-2 scalability solution. *Hydra heads* enable groups of participants to advance an a priori locked portion of the mainchain UTXO set in a fast off-chain protocol with fast settlement in a manner that provides the same level of security as the mainchain. The mainchain portion of the protocol, which is based on EUTXO_{ma}, uses custom tokens to ensure that it is impossible for a subgroup of the participants to collude to exclude one or more other participants.

3.4 Algorithmic stablecoins

In [4], we described how to implement a simple, centralised stablecoin in the UTXO_{ma}. However, more sophisticated stablecoin designs exist, such as the Dai of MakerDAO [9].

Stablecoins where more of the critical functionality is validated on-chain can be realised within the EUTXO_{ma} model. For example, we can use a state machine that acts as a *market maker* for a stablecoin by forging stablecoins in exchange for other assets. Mechanisms to audit updates to the current market price, or to suspend trading if the fund's liabilities become too great, can also be implemented programmatically on-chain.

4 Meta-theoretical Properties of EUTXO$_{\mathbf{ma}}$

In [5], we characterised the expressiveness of EUTXO ledgers by showing that we can encode a class of state machines — *Constraint Emitting Machines* (CEMs) — into the ledger via a series of transactions. Formally, we showed that there is a weak bisimulation between CEM transitions and their encoded transitions in the ledger.

However, as we have seen from the example in §2.4 above, this result is not sufficient to allow us to reason about the properties of the state machine. Even if we know that each individual step is valid, there may be some properties that are only guaranteed to hold if we started from a proper initial state. Specifically, we will not be able to establish any inductive or temporal property unless we cover the base case corresponding to initial states. Since our earlier model did not prevent us from starting the ledger in an arbitrary (and perhaps non-valid) state, such properties could not be carried over from the state machines to the ledger.

In this section we extend our formalisation to map CEMs into threaded ledgers. By formalising some of the properties of EUTXO$_{\text{ma}}$, we can guarantee that a ledger was started from a valid initial state, and thus carry over inductive and temporal properties from state machines. All the results in this section have been mechanised in Agda, extending the formalisation from [5].

4.1 Token provenance

Given a token in a EUTXO$_{\text{ma}}$ ledger, we can ask "where did this token *come from*?" Since tokens are always created in specific forging operations, we can always trace them back through their transaction graph to their *origin*.

We call such a path through the transaction graph a *trace*, and define the *provenance* of an asset at some particular output to be a trace leading from the current transaction to an origin which forges a value containing the traced tokens.

In the case of fungible tokens, there may be multiple possible traces: since such tokens are interchangeable, if two tokens of some asset are forged separately, and then later mingled in the same output, then we cannot say which one came from which forging.

Let $\blacklozenge = (p, a)$ denote a particular Asset a controlled by a Policy p, and $v^{\blacklozenge} = v(p)(a)$ the quantity of $\blacklozenge$ tokens present in value v. $\mathsf{Trace}(l, o, \blacklozenge, n)$ is the type of sequences of transactions $t_0, \ldots, t_i, t_{i+1}, \ldots, t_k$ drawn from a ledger l such that:

- the origin transaction t_0 forges at least n $\blacklozenge$ tokens: $t_0.\mathit{forge}^{\blacklozenge} \geq n$
- every pair (t_i, t_{i+1}) has an input/output connection in the ledger that transfers at least n $\blacklozenge$ tokens: $\exists o' \in t_{i+1}.\mathit{inputs}.\mathit{outputRef}.(t_i.\mathit{outputs}[o'.\mathit{index}].\mathit{value}^{\blacklozenge} \geq n)$
- the last transaction t_k carries the traced tokens in output o

We define $\mathsf{Provenance}(l, o, \blacklozenge)$ to be a set of traces $\ldots \mathsf{Trace}(l, o, \blacklozenge, n_i) \ldots$, such that $\sum n_i \geq o.\mathit{value}^{\blacklozenge}$.

To construct an output's provenance, with respect to a particular policy and asset, we aggregate all possible traces containing a token of this asset from the transaction's inputs as well as from the value that is currently being forged. Thus our tracing procedure will construct a sound over-approximation of the actual provenance.

$$\frac{o \in \{t.\mathit{outputs} \mid t \in l\}}{\mathsf{provenance}(l, o, \blacklozenge) : \mathsf{Provenance}(l, o, \blacklozenge)}\ \textsc{Provenance}$$

4.2 Forging policies are enforced

One particular meta-theoretical property we want to check is that each token is created properly: there is an origin transaction which forges it and is checked against the currency's forging policy.

Proposition 1 (Non-empty Provenance). *Given any token-carrying output of a valid ledger, its computed provenance will be inhabited by at least one trace.*

$$\frac{o \in \{t.outputs \mid t \in l\} \quad o.value^{\blacklozenge} > 0}{|\mathit{provenance}(l, o, \blacklozenge)| > 0}\ \textsc{Provenance}^{+}$$

However, this is not enough to establish what we want: due to fungibility, we could identify a single origin as the provenance for two tokens, even if the origin only forges a single token!

To remedy this, we complement (non-empty) token provenance with a proof of *global preservation*; a generalisation of the local validity condition of *value preservation* from Rule 4 in §A.

Proposition 2 (Global Preservation). *Given a valid ledger l:*

$$\sum_{t \in l} t.forge = \sum_{o \in \mathsf{unspentOutputs}(l)} o.value$$

The combination of non-empty token provenance and global preservation assures us that each token is created properly and there is globally no more currency than is forged.

To prove these properties we require transactions to be unique throughout the ledger, which is not implied by the validity rules of Figure 5 in §A. In practice though, one could derive uniqueness from any ledger that does not have more than one transaction with no inputs, since a transaction would always have to consume some previously unspent output to pay its fees. For our purposes, it suffices to assume there is always a single *genesis* transaction without any inputs:

$$\frac{}{|\{t \in l \mid t.inputs = \emptyset\}| = 1}\ \textsc{Genesis}$$

Then we can derive the uniqueness of transactions and unspent outputs.

4.3 Provenance for non-fungible tokens

However, our approach to threading crucially relies on the idea that a non-fungible token can pick out a *unique* path through the transaction graph.

First, we define a token $\blacklozenge$ to be *non-fungible* whenever it has been forged at most once across all transactions of an existing valid ledger l. (This really is a property of a particular ledger: a "non-fungible" token can become fungible in a longer ledger if other tokens of its asset are forged. Whether or not this is possible will depend on external reasoning about the forging policy.) Then we can prove that non-fungible tokens have a singular provenance; they really do pick out a unique trace through the transaction graph.

Proposition 3 (Provenance for Non-fungible Tokens). *Given a valid ledger l and an unspent output carrying a non-fungible token $\blacklozenge$:*

$$\frac{o \in \{t.outputs \mid t \in l\} \quad o.value^{\blacklozenge} > 0 \quad \sum_{t \in l} t.forge^{\blacklozenge} \leq 1}{|\mathit{provenance}(l, o, \blacklozenge)| = 1}\ \text{NF-}\textsc{Provenance}$$

4.4 Threaded state machines

Armed with the provenance mechanism just described, we are now ready to extend the previously established bisimulation between CEMs and state machines in ledgers, so that it relates CEMs with support for initial states to *threaded* state machines in ledgers. That is, in addition to creating an appropriate validator as in [5], we also create a forging policy that forges a unique *thread token* which is then required for the machine to progress (and is destroyed when it terminates). Crucially, the forging policy also checks that the machine is starting in an initial state.

For the sake of brevity, we refrain from repeating the entirety of the CEM model definitions from [5]. Instead, we focus only on the relevant modifications and refer the reader to the mechanised model for more details.

First, we omit the notion of final states altogether, i.e., there is no final predicate in the definition of a CEM. (Note that this also simplifies the bisimulation propositions proven in [5], since we no longer need to consider the special case of final states. Other than that, the statements stay exactly the same.) We follow Robin Milner who observed, "What matters about a sequence of actions is not whether it drives the automaton into an accepting state but whether the automaton is able to perform the sequence interactively." [12] Formally, this corresponds to prefix closure: if an automaton accepts a string s then it accepts any initial part of s (Definition 2.6 [12]). Our state machines are not classical state machines which accept or reject a string — rather they are *interactive processes* which respond to user input. However, it might be useful to re-introduce final states in the future if we support burning of the thread token.

On the other hand, we now include the notion of initial state in the predicate function $\mathsf{initial} : \mathsf{S} \to \mathbb{B}$, which characterises the states in which a machine can start from. This enables us to ensure that multiple copies of the machine with the same thread token cannot run at once and the machine cannot be started in an intermediate state. State machines whose execution must start from an initial state are also referred to as *rooted* state transition systems [8].

To enforce non-fungibility of the forged token, we require that the forging transaction spends a specific output, fixed for a particular CEM instance $\mathcal{C}$ and supplied along with its definition as a field origin. Therefore, since no output can be spent twice, we guarantee that the token cannot be forged again in the future.

The CEM's forging policy checks that we only forge a single thread token, spend the supplied origin, and that the state propagated in the outputs is initial:

$$\mathsf{policy}_{\mathcal{C}}(\mathit{txInfo}, c) = \begin{cases} \mathsf{true} & \mathit{if}\ \ \mathit{txInfo.forge}^{\blacklozenge} = 1 \\ & \quad \mathit{and}\ \ \mathsf{origin} \in \mathit{txInfo.outputRefs} \\ & \quad \mathit{and}\ \ \mathsf{initial}(\mathit{txInfo.outputs}^{\blacklozenge}) \\ \mathsf{false} & \mathit{otherwise} \end{cases}$$

where $\blacklozenge = \{\mathsf{validator}^{\#}_{\mathcal{C}} \mapsto \{\mathsf{policy}^{\#}_{\mathcal{C}} \mapsto 1\}\}$ is the thread token and $\mathit{txInfo.outputs}^{\blacklozenge}$ looks up the output which carries the newly-forged thread token and is locked by the same machine's validator, returning the datum decoded as a value of the state type S.

We extend the previous definition of a CEM's validator (displayed in grey) to also check that the thread token is attached to the source state s and propagated to the target state s':

$$\mathsf{validator}_{\mathcal{C}}(s, i, \mathit{txInfo}) = \begin{cases} \mathsf{true} \ \mathit{if} \ s \xrightarrow{i} (s', tx^{\equiv}) \\ \qquad \mathit{and} \ \mathsf{satisfies}(\mathit{txInfo}, tx^{\equiv}) \\ \qquad \mathit{and} \ \mathsf{checkOutputs}(s', \mathit{txInfo}) \\ \qquad \mathit{and} \ \mathsf{propagates}(\mathit{txInfo}, \blacklozenge, s, s') \\ \mathsf{false} \ \mathit{otherwise} \end{cases}$$

This is sufficient to let us prove a key result: given a threaded ledger state (i.e., one that includes an appropriate thread token), that state *must* have evolved from a valid initial state. Since we know that the thread token must have a singular provenance, we know that there is a unique ledger trace that begins by forging that token — which is guaranteed to be an initial CEM state.

Proposition 4 (Initiality). *Given a valid ledger l and an unspent output carrying the thread token $\blacklozenge$, we can always trace it back to a single origin, which forges the token and abides by the forging policy:*

$$\frac{o \in \{t.\mathit{outputs} \mid t \in l\} \quad o.\mathit{value}^{\blacklozenge} > 0}{\exists tr.\ \mathsf{provenance}(l, o, \blacklozenge) = \{tr\} \ \wedge \ \mathsf{policy}_{\mathcal{C}}(\mathsf{mkPolicyContext}(tr_0, \mathsf{validator}^{\#}_{\mathcal{C}}, l^{tr_0})) = \mathsf{true}} \ \textsc{Initiality}$$

where tr_0 denotes the origin of trace tr, and l^t is the prefix of the ledger up to transaction t. The proof essentially relies on the fact that the thread token is non-fungible, as its forging requires spending a unique output. By NF-Provenance, we then get the unique trace back to a *valid* forging transaction, which is validated against the machine's forging policy.

4.5 Property preservation

Establishing correct initialisation is the final piece we need to be able to show that properties of abstract CEMs carry over to their ledger equivalents. It is no longer possible to reach any state that cannot be reached in the abstract state machine, and so any properties that hold over traces of the state machine also hold over traces of the ledger.

Intuitively, looking at the ledger trace we immediately get from Initiality, we can discern an underlying CEM trace that throws away all irrelevant ledger information and only keeps the corresponding CEM steps. After having defined the proper notions of property preservation for CEM traces, we will be able to transfer those for on-chain traces by virtue of this extraction procedure.

A simple example of a property of a machine is an invariant. We consider predicates on states of the machine that we call state-predicates.

Definition 1. *A state-predicate P is an* invariant *if it holds for all reachable states of the machine. i.e., if it holds for the initial state, and for any s, i, s' and $tx^{\equiv}$ such that $s \xrightarrow{i} (s', tx^{\equiv})$, if it holds for s then it holds for s'.*

Traces of state machine execution. We have traced the path of a token through the ledger. We can also trace the execution of a state machine. We consider rooted traces that start in the initial state, and refer to them as just *traces*. A trace records the history of the states of the machine over time and also the inputs that drive the machine from state to state.

Definition 2. *A trace is an inductively defined relation on states:*

$$\frac{\mathsf{initial}(s) = \mathsf{true}}{s \rightsquigarrow^* s}\ \mathsf{root} \qquad \frac{s \rightsquigarrow^* s' \quad s' \xrightarrow{i} (s'', tx^{\equiv})}{s \rightsquigarrow^* s''}\ \mathsf{snoc}$$

This gives us a convenient notion to reason about temporal properties of the machine, such as those which hold at all times, at some time, and properties that hold until another one does. In this paper we restrict our attention to properties that always hold.

Properties of execution traces. Consider a predicate P on a single step of execution. A *step-predicate* $P(s, i, s', tx^{\equiv})$ refers to the incoming state s, the input i, the outgoing state s' and the constraints $tx^{\equiv}$. A particularly important lifting of predicates on steps to predicates on traces is the transformer that ensures the predicate holds for every step in the trace.

Definition 3. *Given a step-predicate P, a trace tr satisfies $\mathsf{All}(P, tr)$ whenever every step $s \xrightarrow{i} (s', tx^{\equiv})$ satisfies $P(s, i, s', tx^{\equiv})$.*

A predicate transformer lifting state-predicates to a predicate that holds everywhere in a trace can be defined as a special case: $\mathsf{All}^{\mathsf{S}}(P, tr) = \mathsf{All}(\lambda s\ i\ s'\ tx^{\equiv}.\ P(s) \wedge P(s'),\ tr)$.

We are not just interested in properties of CEM traces in isolation, but more crucially in whether these properties hold when a state machine is compiled to a contract to be executed on-chain.

We observe that the on-chain execution traces precisely follow the progress of the thread token. We use the same notion of token provenance to characterise on-chain execution traces. To facilitate reasoning about these traces, we can *extract* the corresponding state machine execution traces and reason about those. This allows us to confine our reasoning to the simpler setting of state machines.

Proposition 5 (Extraction). *Given a valid ledger l and a singular provenance of the (non-fungible) thread token ⧫, we can extract a rooted state machine trace.*

$$\frac{\mathsf{provenance}(l, o, ⧫) = \{tr\}}{tr.source \rightsquigarrow^* tr.destination}\ \textsc{Extraction}$$

where $tr.source, tr.destination$ are the states associated with the endpoints of the trace tr.

Proof. It is straightforward to show this holds, as a corollary of Initiality. For the base case, knowing that the origin of the trace abides by the forging policy ensures that it forges the thread token and outputs an initial state (root). In the inductive step, knowing that the validator runs successfully guarantees that there is a corresponding CEM step (snoc). □

Corollary 1. *Any predicate that always holds for a CEM trace also holds for the one extracted from the on-chain trace.*

We will write $\mathsf{extract}(tr)$ whenever we want to extract the CEM trace from the on-chain trace tr.

Example 1. Consider a CEM representing a simple counter that counts up from zero:

$$(\mathbb{Z}, \{\mathsf{inc}\}, \mathsf{step}, \mathsf{initial}) \quad \textbf{where} \quad \mathsf{step}(i, \mathsf{inc}) = \mathsf{just}(i + 1); \quad \mathsf{initial}(0) = \mathsf{true}$$

A simple property that we want to hold is that the state of the counter is never negative.

Property 1. The counter state c is non-negative, i.e., $c \geq 0$.

Lemma 1. *Property 1 is an invariant, i.e., it holds for all reachable states:*

1. $\forall c.\ \mathsf{initial}(c) \to c \geq 0$
2. $\forall c\ c'.\ c \xrightarrow{i} (c', tx^{\equiv}) \to c \geq 0 \to c' \geq 0$

Proposition 6. *In all reachable states, both off-chain and on-chain, the counter is non-negative.*

1. $\forall c\ c'\ (tr : c \rightsquigarrow^* c').\ \mathsf{All}^{\mathsf{S}}(\lambda x.\ x \geq 0,\ tr)$
2. $\forall l\ o\ tr.\ \mathsf{provenance}(l, o, \blacklozenge) = \{tr\} \to \mathsf{All}^{\mathsf{S}}(\lambda x.\ x \geq 0, \mathsf{extract}(tr))$

Proof. (1) follows from Lemma 1. (2) follows from (1) and Corollary 1. □

Example 2. We return to the n–of–m multi-signature contract of §2. We pass as parameters to the machine a threshold number n of signatures required and a list of m owner public keys *signatories*. The states of the machine are given by $\{\mathsf{Holding}, \mathsf{Collecting}\}$ and the inputs are given by $\{\mathsf{Pay}, \mathsf{Cancel}, \mathsf{Add}, \mathsf{Propose}\}$, along with their respective arguments. The only initial state is Holding and we omit the definition of step which can be read off from the picture in Figure 1.

First and foremost, the previous limitation of starting in non-initial states has now been overcome, as proven for all state machines in Proposition 5. Specifically, this holds at any output in the ledger carrying the Collecting state, therefore it is no longer possible to circumvent the checks performed by the Add input.

Property 2. It is only possible to cancel after the deadline.

We define a suitable step predicate Q on inputs and constraints for a step. If the input is Cancel then the constraints must determine that the transaction can appear on the chain only after the deadline. If the input is not Cancel then the predicate is trivially satisfied.

$$Q(s, i, \_, tx^{\equiv}) = \begin{cases} \mathsf{false}\ \textit{if}\ i = \mathsf{Cancel}\ \textit{and}\ s = \mathsf{Propose}(\_, \_, d)\ \textit{and}\ tx^{\equiv}.\mathsf{range} \neq d \ldots +\infty \\ \mathsf{true}\ \ \textit{otherwise} \end{cases}$$

Note that we could extend Q to include cases for correctness properties of other inputs such as ensuring that only valid signatures are added and that payments are sent to the correct recipient.

Lemma 2. *Q holds everywhere in any trace. i.e.* $\forall s\ s'\ (tr : s \rightsquigarrow^* s').\ \mathsf{All}(Q,\ tr)$.

Proof. By induction on the trace tr. □

Proposition 7. *For any trace, all cancellations occur after the deadline.*

Proof. Follows from Lemma 2 and the fact that the validator ensures all constraints are satisfied. □

Beyond safety properties. All the properties presented here are *safety* properties, i.e., they hold in every state and we express them as state predicates. However, a large class of properties we would like to prove are *temporal*, as in relating different states (also known as *liveness* or *progress* properties [10,2]).

Scilla, for instance, provides the `since_as_long` temporal operator for such purposes [15], which we have also encoded in our framework using a straightforward inductive definition. However, one may need to move to infinitary semantics to encode the entirety of *Linear Temporal Logic* (LTL) or *Computational Tree Logic* (CTL), in contrast to reasoning only about finite traces. In our setting, we would need to provide a *coinductive* definition of traces to encode the temporal operators of the aforementioned logics, as done in [13] in the constructive setting of the Coq proof assistant. We leave this exploration for future work.

5 Related work

We discuss other efforts to use state machines to model smart contracts in our previous paper [5] and we compare our approach to a multi-asset ledger with other systems (including Waves [18], Stellar [16], and Zilliqa [14]) in the companion paper [4]. Here we focus on approaches that reason formally about the properties of such contracts as state machines, which is the essence of this paper's main contribution. We do not know of any other approaches that use tokens as state threads.

Scilla. Scilla [15] is a intermediate-level language for writing smart contracts as state machines. The Scilla authors have used Coq to reason about contracts written in Scilla, proving a variety of temporal properties such as safety, liveness, and others; hence their goals are similar to ours. Since our meta-theory enjoys property preservation over any trace predicate, we can also formally prove these temporal properties.

However, we are targeting a very different ledger model. This means that we need to do additional work: the major contribution of this paper is using tokens to provide state machine instances with an "identity", which comes for free on Ethereum. Another Ethereum feature that widens the gap between our approaches is support for asynchronous message passing, which renders Scilla unsuitable as a source language for a UTXO-based ledger, and explains the different choice of *communicating automata* [14] as the backbone of its model. Nonetheless, it would be interesting to develop a Scilla-like language that was suitable for our ledger model.

BitML. The *Bitcoin Modelling Language* (BitML) [3] allows the definition of smart contracts running on Bitcoin by means of a restricted class of state machines. The BitML compilation process has been proven to be computationally sound (although this proof has not yet been mechanised), which allows trace-based properties about the BitML contract to be transferred to the implementation, as in our system. This proof is used, for example, in [1] to prove and transfer LTL properties of BitML contracts to their implementations. Most importantly, LTL formulas can be automatically verified using a dedicated *model checker*.

Again, our work is closely related in spirit, although our ledger model is different and we use a more expressive class of state machines. In the future, we plan to add support for LTL formulas in our framework.

VeriSolid. VeriSolid [11] synthesises Solidity smart contracts from a state machine specification, and verifies temporal properties of the state machine using CTL. They use this to prove safety, liveness, deadlock-freedom, and others. Again, we expect to support CTL formulas in the near future.

In contrast, the present work focuses on establishing a formal connection between the state machine model and the real implementation on the ledger — in particular, our proofs are mechanised. We also target a UTXO ledger model, whereas VeriSolid targets the Ethereum ledger. Finally, our approach is agnostic about the logic or checker used to prove the properties that we assert on state machines and, by way of the results in this paper, transfer to an $\mathrm{EUTXO}_{\mathrm{ma}}$ implementation of the same state machine.

Acknowledgments. We thank Gabriele Keller for comments on an earlier version of this paper.

References

1. Atzei, N., Bartoletti, M., Lande, S., Yoshida, N., Zunino, R.: Developing secure Bitcoin contracts with BitML. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1124–1128 (2019)
2. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
3. 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)
4. Chakravarty, M.M.T., Chapman, J., MacKenzie, K., Melkonian, O., Müller, J., Peyton Jones, M., Vinogradova, P., Wadler, P., Zahnentferner, J.: UTXO_{ma}: UTXO with multi-asset support. In: 9th International Symposium on Leveraging Applications of Formal Methods (ISoLA). Springer (2020), to appear; https://omelkonian.github.io/data/publications/utxoma.pdf
5. 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)
6. Chakravarty, M.M.T., Coretti, S., Fitzi, M., Gazi, P., Kant, P., Kiayias, A., Russell, A.: Hydra: Fast isomorphic state channels. Tech. rep., Cryptology ePrint Archive, Report 2020/299 (2020), https://eprint.iacr.org/2020/299
7. Entriken, W., Shirley, D., Evans, J., Sachs, N.: ERC-721 non-fungible token standard. Ethereum Foundation (2018), https://eips.ethereum.org/EIPS/eip-721
8. Kröger, F., Merz, S.: Temporal Logic and State Systems. Springer (2008)
9. Maker Team: The Dai stablecoin system (2017), https://makerdao.com/whitepaper/DaiDec17WP.pdf
10. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems - specification. Springer (1992). https://doi.org/10.1007/978-1-4612-0931-7, https://doi.org/10.1007/978-1-4612-0931-7
11. Mavridou, A., Laszka, A., Stachtiari, E., Dubey, A.: VeriSolid: Correct-by-design smart contracts for Ethereum. In: International Conference on Financial Cryptography and Data Security. pp. 446–465. Springer (2019)
12. Milner, R.: Communicating and mobile systems: the π-calculus. Cambridge University Press (1999)
13. Nakata, K., Uustalu, T., Bezem, M.: A proof pearl with the fan theorem and bar induction - walking through infinite trees with mixed induction and coinduction. In: Yang, H. (ed.) Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5-7, 2011. Proceedings. Lecture Notes in Computer Science, vol. 7078, pp. 353–368. Springer (2011). https://doi.org/10.1007/978-3-642-25318-8_26, https://doi.org/10.1007/978-3-642-25318-8_26
14. Sergey, I., Kumar, A., Hobor, A.: Scilla: a smart contract intermediate-level language. arXiv preprint arXiv:1801.00687 (2018)
15. 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)
16. Stellar Development Foundation: Stellar Development Guides. https://solidity.readthedocs.io/ (2020)
17. Vogelsteller, F., Buterin, V.: ERC-20 token standard. Ethereum Foundation (Stiftung Ethereum), Zug, Switzerland (2015), https://eips.ethereum.org/EIPS/eip-20
18. Waves Team: Waves blockchain documentation. https://docs.wavesprotocol.org/ (2020)

A Complete formal definition of EUTXO$_{\text{ma}}$

A.1 Finitely-supported functions.

If K is any type and M is a monoid with identity element 0, then a function $f : K \to M$ is *finitely supported* if $f(k) \neq 0$ for only finitely many $k \in K$. More precisely, for $f : K \to M$ we define the *support* of f to be $\mathsf{supp}(f) = \{k \in K : f(k) \neq 0\}$ and $\mathsf{FinSup}[K, M] = \{f : K \to M : |\mathsf{supp}(f)| < \infty\}$.

If $(M, +, 0)$ is a monoid then $\mathsf{FinSup}[K, M]$ also becomes a monoid if we define addition pointwise $((f + g)(k) = f(k) + g(k))$, with the identity element being the zero map. Furthermore, if M is an abelian group then $\mathsf{FinSup}[K, M]$ is also an abelian group under this construction, with $(-f)(k) = -f(k)$. Similarly, if M is partially ordered, then so is $\mathsf{FinSup}[K, M]$ with comparison defined pointwise: $f \leq g$ if and only if $f(k) \leq g(k)$ for all $k \in K$.

It follows that if M is a (partially ordered) monoid or abelian group then so is $\mathsf{FinSup}[K, \mathsf{FinSup}[L, M]]$ for any two sets of keys K and L. We will make use of this fact in the validation rules presented later (see Figure 5).

Finitely-supported functions are easily implemented as finite maps, with a failed map lookup corresponding to returning 0.

A.2 Ledger types

The formal definition of the EUTXO$_{\text{ma}}$ model integrates token bundles and forge fields from the plain UTXO$_{\text{ma}}$ model [4] into the single currency EUTXO model definition, while adapting forging policy scripts to enjoy the full expressiveness of validators in EUTXO (rather than the limited domain-specific language of UTXO$_{\text{ma}}$). Figures 2 and 3 define the ledger types for EUTXO$_{\text{ma}}$.

A.3 Transaction validity

Finally, we provide the transaction validity rules, which among other things state how forging policy scripts affect transaction validity. To this end, we replace the notion of an integral Quantity for values by the token bundles discussed in §2.2 and represented by values of the type Quantities.

As indicated in §2.1, validator scripts get a context argument, which includes the validated transaction as well as the outputs that it consumed, in EUTXO. For EUTXO$_{\text{ma}}$, we need two different such context types. We have ValidatorContext for validators and PolicyContext for forging policies. The difference is that in ValidatorContext we indicate the input of the validated transaction that consumes the output locked by the executed validator, whereas for forging policies, we provide the policy script hash. The latter makes it easy for the policy script to look up the component of the transaction's forging field that it is controlling.

The validity rules in Figure 5 define what it means for a transaction t to be valid for a valid ledger l during the tick currentTick. (They make use of the auxiliary functions in Figure 4.) Of these rules, Rules 1, 2, 3, 4, and 5 are common to the two systems (EUTXO and UTXO$_{\text{ma}}$) that we are combing here; Rules 6 and 7 are similar in both systems, but we go with the more expressive ones from EUTXO. The crucial changes are to the construction and passing of the context types mentioned above, which appear in Rules 6 and 10. The later is the main point as it is responsible for execution of forging policy scripts.

A ledger l is *valid* if either l is empty or l is of the form $t :: l'$ with l' valid and t valid for l'.

Basic types

$\mathbb{B}, \mathbb{N}, \mathbb{Z}$	the type of Booleans, natural numbers, and integers
$\mathbb{H}$	the type of bytestrings: $\bigcup_{n=0}^{\infty}\{0,1\}^{8n}$
$g(\phi_1 : T_1, \ldots, \phi_n : T_n)$	a record type with fields $\phi_1, \ldots, \phi_n$ of types $T_1, \ldots, T_n$
$t.\phi$	the value of ϕ for t, where t has type T and ϕ is a field of T
$\mathsf{Set}[T]$	the type of (finite) sets over T
$\mathsf{List}[T]$	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

Quantity	an amount of an assets
Asset	a type consisting of identifiers for individual asset classes
Tick	a tick
Address	an "address" in the blockchain
Data	a type of structured data
DataHash	the hash of a value of type Data
dataHash : Data → DataHash	computes the hash of an value of typeData
TxId	the identifier of a transaction
txId : Tx → TxId	computes the identifier of a transaction
lookupTx : Ledger × TxId → Tx	retrieves the unique transaction with a given identifier
Script	the (opaque) type of scripts
$[\![\_]\!]$: Script → Data × ⋯ × Data → $\mathbb{B}$	applies a script to its arguments
scriptAddr : Script → Address	the address of a script

Defined types

$$
\begin{aligned}
\mathsf{Policy} &= \mathsf{Address} \\
\mathsf{Signature} &= \mathbb{H} \\
\\
\mathsf{Quantities} &= \mathsf{FinSup}[\mathsf{Policy}, \mathsf{FinSup}[\mathsf{Asset}, \mathsf{Quantity}]] \\
\\
\mathsf{Output} &= (\mathit{addr} : \mathsf{Address}, \mathit{value} : \mathsf{Quantities}, \mathit{datumHash} : \mathsf{DataHash}) \\
\\
\mathsf{OutputRef} &= (\mathit{id} : \mathsf{TxId}, \mathit{index} : \mathsf{Int}) \\
\\
\mathsf{Input} &= (\mathit{outputRef} : \mathsf{OutputRef}, \\
&\qquad \mathit{validator} : \mathsf{Script}, \\
&\qquad \mathit{datum} : \mathsf{Data}, \\
&\qquad \mathit{redeemer} : \mathsf{Data}) \\
\\
\mathsf{Tx} &= (\mathit{inputs} : \mathsf{Set}[\mathsf{Input}], \\
&\qquad \mathit{outputs} : \mathsf{List}[\mathsf{Output}], \\
&\qquad \mathit{validityInterval} : \mathsf{Interval}[\mathsf{Tick}], \\
&\qquad \mathit{forge} : \mathsf{Quantities}, \\
&\qquad \mathit{forgeScripts} : \mathsf{Set}[\mathsf{Script}], \\
&\qquad \mathit{sigs} : \mathsf{Set}[\mathsf{Signature}]) \\
\\
\mathsf{Ledger} &= \mathsf{List}[\mathsf{Tx}]
\end{aligned}
$$

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

$$
\begin{aligned}
\mathsf{OutputInfo} &= (\mathit{value} : \mathsf{Quantities},\\
&\quad \mathit{validatorHash} : \mathsf{Address},\\
&\quad \mathit{datumHash} : \mathsf{DataHash})\\
\\
\mathsf{InputInfo} &= (\mathit{outputRef} : \mathsf{OutputRef},\\
&\quad \mathit{validatorHash} : \mathsf{Address},\\
&\quad \mathit{datumHash} : \mathsf{DataHash},\\
&\quad \mathit{redeemerHash} : \mathsf{DataHash},\\
&\quad \mathit{value} : \mathsf{Quantities})\\
\\
\mathsf{TxInfo} &= (\mathit{inputInfo} : \mathsf{List}[\mathsf{InputInfo}],\\
&\quad \mathit{outputInfo} : \mathsf{List}[\mathsf{OutputInfo}],\\
&\quad \mathit{validityInterval} : \mathsf{Interval}[\mathsf{Tick}],\\
&\quad \mathit{forge} : \mathsf{Quantities},\\
&\quad \mathit{forgeScripts} : \mathsf{Set}[\mathsf{Script}],\\
&\quad \mathit{sigs} : \mathsf{FinSet}[\mathsf{Signature}])\\
\\
\mathsf{ValidatorContext} &= (\mathsf{TxInfo}, \mathbb{N})\\
\mathsf{PolicyContext} &= (\mathsf{TxInfo}, \mathsf{Policy})
\end{aligned}
$$

$\mathsf{mkValidatorContext} : \mathsf{Tx} \times \mathsf{Input} \times \mathsf{Ledger} \to \mathsf{ValidatorContext}$
summarises a transaction for a validator script in the context of an input and a ledger state

$\mathsf{mkPolicyContext} : \mathsf{Tx} \times \mathsf{Policy} \times \mathsf{Ledger} \to \mathsf{PolicyContext}$
summarises a transaction for a forging policy script in the context of an currency and a ledger state

Fig. 3: The Context types for the $\mathrm{EUTXO}_{\mathrm{ma}}$ model

$$
\begin{aligned}
&\mathsf{unspentTxOutputs} : \mathsf{Tx} \to \mathsf{Set}[\mathsf{OutputRef}]\\
&\mathsf{unspentTxOutputs}(t) = \{(\mathsf{txId}(t), 1), \ldots, (\mathsf{txId}(id), |t.\mathit{outputs}|)\}\\
\\
&\mathsf{unspentOutputs} : \mathsf{Ledger} \to \mathsf{Set}[\mathsf{OutputRef}]\\
&\mathsf{unspentOutputs}([]) = \{\}\\
&\mathsf{unspentOutputs}(t :: l) = (\mathsf{unspentOutputs}(l) \setminus t.\mathit{inputs}) \cup \mathsf{unspentTxOutputs}(t)\\
\\
&\mathsf{getSpentOutput} : \mathsf{Input} \times \mathsf{Ledger} \to \mathsf{Output}\\
&\mathsf{getSpentOutput}(i, l) = \mathsf{lookupTx}(l, i.\mathit{outputRef}.\mathit{id}).\mathit{outputs}[i.\mathit{outputRef}.\mathit{index}]
\end{aligned}
$$

Fig. 4: Auxiliary functions for $\mathrm{EUTXO}_{\mathrm{ma}}$ validation

1. **The current tick is within the validity interval**

$$\mathsf{currentTick} \in t.validityInterval$$

2. **All outputs have non-negative values**

$$\text{For all } o \in t.outputs,\ o.value \geq 0$$

3. **All inputs refer to unspent outputs**

$$\{i.outputRef \mid i \in t.inputs\} \subseteq \mathsf{unspentOutputs}(l).$$

4. **Value is preserved**

$$t.forge + \sum_{i \in t.inputs} \mathsf{getSpentOutput}(i, l) = \sum_{o \in t.outputs} o.value$$

5. **No output is double spent**

$$\text{If } i_1, i \in t.inputs \text{ and } i_1.outputRef = i.outputRef \text{ then } i_1 = i.$$

6. **All inputs validate**

$$\text{For all } i \in t.inputs,\ [\![i.validator]\!](i.datum,\ i.redeemer,\ \mathsf{toData}(\mathsf{mkValidatorContext}(t, i, l))) = \mathsf{true}$$

7. **Validator scripts match output addresses**

$$\text{For all } i \in t.inputs,\ \mathsf{scriptAddr}(i.validator) = \mathsf{getSpentOutput}(i, l).addr$$

8. **Datum objects match output hashes**

$$\text{For all } i \in t.inputs,\ \mathsf{dataHash}(i.datum) = \mathsf{getSpentOutput}(i, l).datumHash$$

9. **Forging**
   A transaction with a non-zero *forge* field is only valid if either:
   (a) the ledger l is empty (that is, if the transaction is the initial transaction).
   (b) for every key $h \in \mathsf{supp}(t.forge)$, there exists $s \in t.forgeScripts$ with $\mathsf{scriptAddr}(s) = h$.

10. **All forging policy scripts validate**

$$\text{For all } s \in t.forgeScripts,\ [\![s]\!](\mathsf{toData}(\mathsf{mkPolicyContext}(t, \mathsf{scriptAddr}(s), l))) = \mathsf{true}$$

Fig. 5: Validity of a transaction t in the EUTXO_{ma} model