UTXO_{ma}: UTXO with Multi-Asset Support

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], Philip Wadler[1,2], and Joachim Zahnentferner[3]

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

Abstract. A prominent use case of Ethereum smart contracts is the creation of a wide range of *user-defined tokens* or *assets* by way of smart contracts. User-defined assets are *non-native* on Ethereum; i.e., they are not directly supported by the ledger, but require repetitive custom code. This makes them unnecessarily inefficient, expensive, and complex. It also makes them insecure as numerous incidents on Ethereum have demonstrated. Even without stateful smart contracts, the lack of perfect fungibility of Bitcoin assets allows for implementing user-defined tokens as layer-two solutions, which also adds an additional layer of complexity.
In this paper, we explore an alternative design based on Bitcoin-style UTXO ledgers. Instead of introducing general scripting capabilities together with the associated security risks, we propose an extension of the UTXO model, where we replace the accounting structure of a single cryptocurrency with a new structure that manages an unbounded number of user-defined, native tokens, which we call *token bundles*. Token creation is controlled by *forging policy scripts* that, just like Bitcoin validator scripts, use a small domain-specific language with bounded computational expressiveness, thus favouring Bitcoin's security and computational austerity. The resulting approach is lightweight, i.e., custom asset creation and transfer is cheap, and it avoids use of any global state in the form of an asset registry or similar.
The proposed UTXO_{ma} model and the semantics of the scripting language have been formalised in the Agda proof assistant.

Keywords: blockchain · UTXO · native tokens · functional programming.

1 Introduction

Distributed ledgers began by tracking just a single asset — money. The goal was to compete with existing currencies, and so they naturally started by focusing on their own currencies — Bitcoin and its eponymous currency, Ethereum and Ether, and so on. This focus was so clear that the systems tended to be identified with their primary currency.

More recently, it has become clear that it is possible and very useful to track other kinds of asset on distributed ledger systems. Ethereum has led the innovation in this space, with ERC-20 [15] implementing new currencies and ERC-721 [8] implementing unique non-fungible tokens.

These have been wildly popular — variants of ERC-20 are the most used smart contracts on Ethereum by some margin. However, they have major shortcomings. Notably, custom tokens on Ethereum are not native. This means that tokens do not live in a user's account, and in order to send another user ERC-20 tokens, the sender must interact with the governing smart contract for the currency. That is, despite the fact that Ethereum's main purpose is to track ownership of assets

and perform transactions, users have been forced to build their own *internal ledger* inside a smart contract.

Other systems have learned from this and have made custom tokens native, such as Stellar, Waves, Zilliqa, and more. However, these typically rely on some kind of global state, such as a global currency registry, or special global accounts that must be created. This is slow, and restricts creative use of custom tokens because of the high overhead in terms of time and money. There have also been efforts to introduce native multi-assets into UTXO ledgers [19], a precursor to our work.

We can do better than this through a combination of two ideas. Firstly, we generalise the value type that the ledger works with to include *token bundles* that freely and uniformly mix tokens from different custom assets, both fungible and non-fungible. Secondly, we avoid any global state by "eternally" linking a currency to a governing forging policy via a hash. Between them this creates a multi-asset ledger system (which we call $\text{UTXO}_{\mathsf{ma}}$) with native, lightweight custom tokens.

Specifically, this paper makes the following contributions:

- We introduce token bundles, represented as finitely-supported functions, as a uniform mechanism to generalise the existing UTXO accounting rules to custom assets including fungible, non-fungible, and mixed tokens.
- We avoid the need for global state in the form of a currency registry by linking custom forging policy scripts by way of their script hash to the name of asset groups.
- We support a wide range of standard applications for custom assets without the need for general-purpose smart contracts by defining a simple domain-specific language for forging policy scripts.
- We provide a formal definition of the $\text{UTXO}_{\mathsf{ma}}$ ledger rules as a basis to formally reason about the resulting system, along with a mechanised version in Agda.[4]

Creating and transferring new assets in the resulting system is lightweight and cheap. It is lightweight as we avoid special setup transactions or registration procedures, and it is cheap as only standard transaction fees are required — this is unlike the Ethereum gas model, where a script must be run each time a custom asset is transferred, which incurs gas costs.

The proposed multi-asset system is not merely a pen and paper exercise. It forms the basis of the multi-asset support for the Cardano blockchain. In a related work [4], we further modify the native multi-asset ledger presented in this paper to an extended UTxO ledger model, which additionally supports the use of Plutus (Turing complete smart contract language) to define forging policies. This ledger model extension allows the use of stateful smart contracts to define state machines for output locking.

2 Multi-Asset Support

In Bitcoin's ledger model [10,2,18], transactions spend as yet *unspent transaction outputs* (*UTXOs*), while supplying new unspent outputs to be consumed by subsequent transactions. Each individual UTXO locks a specific *quantity* of cryptocurrency by imposing specific conditions that need to be met to spend that quantity, such as for example signing the spending transaction with a specific secret cryptographic key, or passing some more sophisticated conditions enforced by a *validator script*. Quantities of cryptocurrency in a transaction output are represented as an integral number of the smallest unit of that particular cryptocurrency — in Bitcoin, these are Satoshis. To natively support multiple currencies in transaction outputs, we generalise those integral quantities to natively

[4] https://github.com/omelkonian/formal-utxo/tree/ed72

support the dynamic creation of new user-defined *assets* or *tokens*. Moreover, we require a means to forge tokens in a manner controlled by an asset's *forging policy*.

We achieve all this by the following three extensions to the basic UTXO ledger model that are further detailed in the remainder of this section.

1. Transaction outputs lock a *heterogeneous token bundle* instead of only an integral value of one cryptocurrency.
2. We extend transactions with a *forge* field. This is a token bundle of tokens that are created (minted) or destroyed (burned) by that transaction.
3. We introduce *forging policy scripts* (*FPS*) that govern the creation and destruction of assets in forge fields. These scripts are not unlike the validators locking outputs in UTXO.

2.1 Token bundles

We can regard transaction outputs in an UTXO ledger as pairs $(value, \nu)$ consisting of a locked value *value* and a validator script ν that encodes the spending condition. The latter may be proof of ownership by way of signing the spending transaction with a specific secret cryptography key or a temporal condition that allows an output to be spent only when the blockchain has reached a certain height (i.e. a certain number of blocks have been produced).

To conveniently use multiple currencies in transaction outputs, we want each output to be able to lock varying quantities of multiple different currencies at once in its *value* field. This suggests using finite maps from some kind of *asset identifier* to an integral quantity as a concrete representation, e.g. $\texttt{Coin} \mapsto 21$. Looking at the standard UTXO ledger rules [18], it becomes apparent that cryptocurrency quantities need to be monoids. It is a little tricky to make finite maps into a monoid, but the solution is to think of them as *finitely supported functions* (see Section 3 for details).

If want to use *finitely supported functions* to achieve a uniform representation that can handle groups of related, but *non-fungible* tokens, we need to go a step further. In order to not lose the grouping of related non-fungible tokens (all house tokens issued by a specific entity, for example) though, we need to move to a two-level structure — i.e., finitely-supported functions of finitely-supported functions. Let's consider an example. Trading of rare in-game items is popular in modern, multi-player computer games. How about representing ownership of such items and trading of that ownership on our multi-asset UTXO ledger? We might need tokens for "hats" and "swords", which form two non-fungible assets with possibly multiple tokens of each asset — a hat is interchangeable with any other hat, but not with a sword, and also not with the currency used to purchase these items. Here our two-level structure pays off in its full generality, and we can represent currency to purchase items together with sets of items, where some can be multiples, e.g.,

$$\begin{aligned} &\{\mathsf{Coin} \mapsto \{\mathsf{Coin} \mapsto 2\}, \mathsf{Game} \mapsto \{\mathsf{Hat} \mapsto 1, \mathsf{Sword} \mapsto 4\}\} \\ +\ &\{\mathsf{Coin} \mapsto \{\mathsf{Coin} \mapsto 1\}, \mathsf{Game} \mapsto \{\mathsf{Sword} \mapsto 1, \mathsf{Owl} \mapsto 1\}\} \\ =\ &\{\mathsf{Coin} \mapsto \{\mathsf{Coin} \mapsto 3\}, \mathsf{Game} \mapsto \{\mathsf{Hat} \mapsto 1, \mathsf{Sword} \mapsto 5, \mathsf{Owl} \mapsto 1\}\}\ . \end{aligned}$$

2.2 Forge fields

If new tokens are frequently generated (such as issuing new hats whenever an in-game achievement has been reached) and destroyed (a player may lose a hat forever if the wind picks up), these operations need to be lightweight and cheap. We achieve this by adding a forge field to every

$\mathbb{B}$	the type of Booleans
$\mathbb{N}$	the type of natural numbers
$\mathbb{Z}$	the type of integers
$\mathbb{H}$	the type of bytestrings: $\bigcup_{n=0}^{\infty}\{0,1\}^{8n}$
$(\phi_1 : T_1, \ldots, \phi_n : T_n)$	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
$x \mapsto f(x)$	an anonymous function
$c^{\#}$	a cryptographic collision-resistant hash of c
$\mathsf{Interval}[A]$	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

Fig. 1: Basic types and notation

transaction. It is a token bundle (just like the *value* in an output), but admits positive quantities (for minting new tokens) and negative quantities (for burning existing tokens). Of course, minting and burning needs to be strictly controlled.

2.3 Forging policy scripts

The script validation mechanism for locking UTXO outputs is as follows : in order to for a transaction to spend an output $(value, \nu)$, the validator script ν needs to be executed and approve of the spending transaction. Similarly, the forging policy scripts associated with the tokens being minted or burned by a transaction are run in order to validate those actions. In the spirit of the Bitcoin Miniscript approach, we chose to include a simple scripting language supporting forging policies for several common usecases, such as single issuer, non-fungible, or one-time issue tokens, etc. (see Section 4 for all the usecases).

In order to establish a permanent association between the forging policy and the assets controlled by it, we propose a hashing approach, as opposed to a global registry lookup. Such a registry requires a specialized access control scheme, as well as a scheme for cleaning up unused entries. In the representation of custom assets we propose, each token is associated with the hash of the forging policy script required to validate at the time of forging the token, eg. in order to forge the value $\{\mathsf{HASHVALUE} \mapsto \{\mathsf{Owl} \mapsto 1\}\}$, a script whose hash is $\mathsf{HASHVALUE}$ will be run.

Relying on permanent hash associations to identify asset forging policies and their assets also has its disadvantages. For example, policy hashes are long strings that, in our model, will have multiple copies stored on the ledger. Such strings are not human-readable, take up valuable ledger real estate, and increase transaction-size-based fees.

3 Formal ledger rules

Our formal ledger model follows the style of the UTXO-with-scripts model from [18] adopting the notation from [5] with basic types defined as in Figure 1.

Finitely-supported functions. We model token bundles as 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 (i.e., $(f + g)(k) = f(k) + g(k)$), 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 in the paper (see Figure 4). Finitely-supported functions are easily implemented as finite maps, with a failed map lookup corresponding to returning 0.

3.1 Ledger types

Figure 2 defines the ledger primitives and types that we need to define the UTXO$_{\mathsf{ma}}$ model. All outputs use a pay-to-script-hash scheme, where an output is locked with the hash of a script. We use a single scripting language for forging policies and to define output locking scripts. Just as in Bitcoin, this is a restricted domain-specific language (and not a general-purpose language); the details follow in Section 4. We assume that each transaction has a unique identifier derived from its value by a hash function. This is the basis of the lookupTx function to look up a transaction, given its unique identifier.

Token bundles. We generalise per-output transferred quantities from a plain Quantity to a bundle of Quantities. A Quantities represents a token bundle: it is a mapping from a policy and an *asset*, which defines the asset class, to a Quantity of that asset.[5] Since a Quantities is indexed in this way, it can represent any combination of tokens from any assets (hence why we call it a token *bundle*).

Asset groups and forging policy scripts. A key concept is the *asset group*. An asset group is identified by the hash of special script that controls the creation and destruction of asset tokens of that asset group. We call this script the *forging policy script*.

Forging. Each transaction gets a *forge* field, which simply modifies the required balance of the transaction by the Quantities inside it: thus a positive *forge* field indicates the creation of new tokens. In contrast to outputs, Quantities in forge fields can also be negative, which effectively burns existing tokens.[6]

Additionally, transactions get a *scripts* field holding a set of forging policy scripts: Set[Script]. This provides the forging policy scripts that are required as part of validation when tokens are minted or destroyed (see Rule 8 in Figure 4). The forging scripts of the assets being forged are

[5] We have chosen to represent Quantities as a finitely-supported function whose values are themselves finitely-supported functions (in an implementation, this would be a nested map). We did this to make the definition of the rules simpler (in particular Rule 8). However, it could equally well be defined as a finitely-supported function from tuples of PolicyIDs and Assets to Quantitys.

[6] The restriction on outputs is enforced by Rule 2. We simply do not impose such a restriction on the *forge* field: this lets us define rules in a simpler way, with cleaner notation.

Ledger primitives

$\mathsf{Quantity}$	an amount of currency, forming an abelian group (typically $\mathbb{Z}$)
Asset	a type consisting of identifiers for individual asset classes
Tick	a tick
$\mathsf{Address}$	an "address" in the blockchain
TxId	the identifier of a transaction
$\mathsf{txId} : \mathsf{Tx} \to \mathsf{TxId}$	a function computing the identifier of a transaction
$\mathsf{lookupTx} : \mathsf{Ledger} \times \mathsf{TxId} \to \mathsf{Tx}$	retrieve the unique transaction with a given identifier
$\mathsf{verify} : \mathsf{PubKey} \times \mathbb{H} \times \mathbb{H} \to \mathbb{B}$	signature verification
Script	forging policy scripts
$\mathsf{scriptAddr} : \mathsf{Script} \to \mathsf{Address}$	the address of a script
$[\![\_]\!] : \mathsf{Script} \to (\mathsf{Address} \times \mathsf{Tx} \times \mathsf{Set}[\mathsf{Output}]) \to \mathbb{B}$	apply script inside brackets to its arguments

Ledger types

$$
\begin{aligned}
\mathsf{PolicyID} &= \mathsf{Address} && \text{(an identifier for a custom currency)}\\
\mathsf{Signature} &= \mathbb{H}\\
\\
\mathsf{Quantities} &= \mathsf{FinSup}[\mathsf{PolicyID}, \mathsf{FinSup}[\mathsf{Asset}, \mathsf{Quantity}]]\\
\\
\mathsf{Output} &= (\mathit{addr} : \mathsf{Address}, \mathit{value} : \mathsf{Quantities})\\
\\
\mathsf{OutputRef} &= (\mathit{id} : \mathsf{TxId}, \mathit{index} : \mathsf{Int})\\
\\
\mathsf{Input} &= (\mathit{outputRef} : \mathsf{OutputRef}\\
&\qquad \mathit{validator} : \mathsf{Script})\\
\\
\mathsf{Tx} &= (\mathit{inputs} : \mathsf{Set}[\mathsf{Input}],\\
&\qquad \mathit{outputs} : \mathsf{List}[\mathsf{Output}],\\
&\qquad \mathit{validityInterval} : \mathsf{Interval}[\mathsf{Tick}],\\
&\qquad \mathit{forge} : \mathsf{Quantities}\\
&\qquad \mathit{scripts} : \mathsf{Set}[\mathsf{Script}],\\
&\qquad \mathit{sigs} : \mathsf{Set}[\mathsf{Signature}])\\
\\
\mathsf{Ledger} &= \mathsf{List}[\mathsf{Tx}]
\end{aligned}
$$

Fig. 2: Ledger primitives and basic types

executed and the transaction is only considered valid if the execution of the script returns true. A forging policy script is executed in a context that provides access to the main components of the forging transaction, the UTXOs it spends, and the policy ID. The passing of the context provides a crucial piece of the puzzle regarding self-identification: it includes the script's own $\mathsf{PolicyID}$, which avoids the problem of trying to include the hash of a script inside itself.

Validity intervals. A transaction's *validity interval* field contains an interval of ticks (monotonically increasing units of "time", from [5]). The validity interval states that the transaction must only be validated if the current tick is within the interval. The validity interval, rather than the actual current chain tick value, must be used for script validation. In an otherwise valid transaction,

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

Fig. 3: Auxiliary validation functions

passing the current tick to the evaluator could result in different script validation outcomes at different ticks, which would be problematic.

Language clauses. In our choice of the set of predicates `p1, ..., pn` to include in the scripting language definition, we adhere to the following heuristic: we only admit predicates with quantification over finite structures passed to the evaluator in the transaction-specific data, i.e. sets, maps, and lists. The computations we allow in the predicates themselves are well-known computable functions, such as hashing, signature checking, arithmetic operations, comparisons, etc.

The gamut of policies expressible in the model we propose here is fully determined by the collection of predicates, assembled into a single script by logical connectives `&&`, `||`, and `Not`. Despite being made up of only hard-coded predicates and connectives, the resulting policies can be quite expressive, as we will demonstrate in the upcoming applications section. When specifying forging predicates, we use `tx._` notation to access the fields of a transaction.

3.2 Transaction validity

Figure 4 defines what it means for a transaction t to be valid for a valid ledger l during the tick currentTick, using some auxiliary functions from Figure 3. 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'.

The rules follow the usual structure for an UTXO ledger, with a number of modifications and additions. The new **Forging** rule (Rule 8) implements the support for forging policies by requiring that the currency's forging policy is included in the transaction — along with Rule 9 which ensures that they are actually run! The arguments that a script is applied to are the ones discussed earlier.

When forging policy scripts are run, they are provided with the appropriate transaction data, which allows them to enforce conditions on it. In particular, they can inspect the *forge* field on the transaction, and so a forging policy script can identify how much of its own currency was forged, which is typically a key consideration in whether to allow the transaction.

We also need to be careful to ensure that transactions in our new system preserve value correctly. There are two aspects to consider:

- We generalise the type of value to Quantities. However, since Quantities is a monoid (see Section 3), Rule 4 is (almost) identical to the one in the original UTXO model, simply with a different monoid. Concretely, this amounts to preserving the quantities of each of the individual token classes in the transaction.
- We allow forging of new tokens by including the forge field into the balance in Rule 4.

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**

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

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**

$$\text{For all } i \in t.inputs,\ [\![i.validator]\!](\mathsf{scriptAddr}(i.validator), t, \{\mathsf{getSpentOutput}(i, l) \mid i \in t.inputs\}) = \mathsf{true}$$

7. **Validator scripts match output addresses**

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

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

9. **All scripts validate**

$$\text{For all } s \in t.scripts,\ [\![s]\!](\mathsf{scriptAddr}(s), t, \{\mathsf{getSpentOutput}(i, l) \mid i \in t.inputs\}) = \mathsf{true}$$

Fig. 4: Validity of a transaction t in a ledger l

4 A stateless forging policy language

The domain-specific language for forging policies strikes a balance between expressiveness and simplicity. It particular, it is stateless and of bounded computational complexity. Nevertheless, it is sufficient to support the applications described in Section 5.

Semantically meaningful token names. The policy ID is associated with a policy script (it is the hash of it), so it has a semantic meaning that is identified with that of the script. In the clauses of our language, we give semantic meaning to the names of the tokens as well. This allows us to make some judgements about them in a programmatic way, beyond confirming that the preservation of value holds, or which ones are fungible with each other. For example, the `FreshTokens` constructor

```
⟦JustMSig(msig)⟧(h, tx, utxo) = checkMultiSig(msig, tx)

⟦SpendsOutput(o)⟧(h, tx, utxo) = o ∈ { i.outputRef : i ∈ tx.inputs }

⟦TickAfter(tick1)⟧(h, tx, utxo) = tick1 ≤ min(tx.validityInterval)

⟦Forges(tkns)⟧(h, tx, utxo) = (h ↦ tkns ∈ tx.forge) && (h ↦ tkns ≥ 0)

⟦Burns(tkns)⟧(h, tx, utxo) = (h ↦ tkns ∈ tx.forge) && (h ↦ tkns ≤ 0)

⟦FreshTokens⟧(h, tx, utxo) =
   ∀ pid ↦ tkns ∈ tx.forge, pid == h ⇒
     ∀ t ↦ q ∈ tkns,
       t == hash(indexof(t, tkns), tx.inputs) && q == 1

⟦AssetToAddress(addr)⟧(h, tx, utxo) =
   ∀ pid ↦ tkns ∈ utxo.balance, pid == h ⇒
      addr == _ ⇒ (h, pid ↦ tkns) ∈ utxo
    ∧ addr ≠ _ ⇒ (addr, pid ↦ tkns) ∈ utxo

⟦DoForge⟧(h, tx, utxo) = h ∈ supp(tx.forge)

⟦SignedByPIDToken⟧(h, tx, utxo) =
   ∀ pid ↦ tkns ∈ utxo.balance, pid == h ⇒
     ∀ s ∈ tx.sigs, ∃ t ∈ supp(tkns),
       isSignedBy(tx, s, t)

⟦SpendsCur(pid)⟧(h, tx, utxo) =
     pid == _ ⇒ h ∈ supp(utxo.balance)
   ∧ pid ≠ _ ⇒ pid ∈ supp(utxo.balance)
```

Fig. 5: Forging Policy Language

gives us a way to programmatically generate token names which, by construction, mean that these tokens are unique, without ever checking the global ledger state.

Forging policy scripts as output-locking scripts. As with currency in the non-digital world, it is a harder problem to control the transfer of assets once they have come into circulation (see also Section 7). We can, however, specify directly in the forging policy that the assets being forged must be locked by an output script of our choosing. Moreover, since both output addresses and policies are hashes of scripts, we can use the asset policy ID and the address interchangeably. The `AssetToAddress` clause is used for this purpose.

Language clauses. The various clauses of the validator and forging policy language are as described below, with their formal semantics as in Figure 5. In this figure, we use the notation `x ↦ y` to represent a single key-value pair of a finite map. Recall form Rule 9 that the arguments passed to

the validation function ⟦`s`⟧ `h` are: the hash of the forging (or output locking) script being validated, the transaction `tx` being validated, and the ledger outputs which the transaction `tx` is spending (we denote these `utxo` here).

- `JustMSig(msig)` verifies that the m-out-of-n signatures required by s are in the set of signatures provided by the transaction. We do not give the multi-signature script evaluator details as this is a common concept, and assume a procedure `checkMultiSig` exists.
- `SpendsOutput(o)` checks that the transaction spends the output referenced by `o` in the UTXO.
- `TickAfter(tick1)` checks that the validity interval of the current transaction starts after time `tick1`.
- `Forges(tkns)` checks that the transaction forges exactly `tkns` of the asset with the policy ID that is being validated.
- `Burns(tkns)` checks that the transaction burns exactly `tkns` of the asset with the policy ID that is being validated.
- `FreshTokens` checks that all tokens of the asset being forged are non-fungible.
  This script must check that the names of the tokens in this token bundle are generated by hashing some unique data. This data must be unique to both the transaction itself and the token within the asset being forged. In particular, we can hash a pair of
  1. some *output* in the UTXO that the transaction consumes, and
  2. the *index* of the token name in (the list representation of) the map of tokens being forged (under the specific policy, by this transaction). We denote the function that gets the index of a key in a key-value map by `indexof`.
- `AssetToAddress(addr)` checks that all the tokens associated with the policy ID that is equal to the hash of the script being run are output to an UTXO with the address `addr`. In the case that no `addr` value is provided (represented by _), we use the `addr` value passed to the evaluator as the hash of the policy of the asset being forged.
- `DoForge` checks that this transaction forges tokens in the bundle controlled by the policy ID that is passed to the FPS script evaluator (here, again, we make use of the separate passing of the FPS script and the policy ID).
- `SignedByPIDToken(pid)` verifies the hash of every key that has signed the transaction.
- `SpendsCur(pid)` verifies that the transaction is spending assets in the token bundle with policy ID `pid` (which is specified as part of the *constructor*, and may be different than the policy ID passed to the evaluator).

5 Applications

UTXO_{ma} is able to support a large number of standard use cases for multi-asset ledgers, as well as some novel ones. In this section we give a selection of examples. There are some common themes: (1) Tokens as resources can be used to reify many non-obvious things, which makes them first-class tradeable items; (2) cheap tokens allow us to solve many small problems with *more tokens*; and (3) the power of the scripting language affects what examples can be implemented.

5.1 Simple single token issuance

To create a simple currency SimpleCoin with a fixed supply of `s = 1000 SimpleCoins` tokens, we might try to use the simple policy script `Forges(s)` with a single forging transaction. Unfortunately, this is not sufficient as somebody else could submit another transaction forging another `1000 SimpleCoins`.

In other words, we need to ensure that there can only ever be a single transaction on the ledger that successfully forges SimpleCoin. We can achieve that by requiring that the forging transaction consumes a specific UTXO. As UTXOs are guaranteed to be (1) unique and (2) only be spent once, we are being guaranteed that the forging policy can only be used once to forge tokens. We can use the script:

```
simple_policy(o, v) = SpendsOutput(o) && Forges(v)
```

where `o` is an output that we create specifically for this purpose in a preceding setup transaction, and `v = s`.

5.2 Reflections of off-ledger assets

Many tokens are used to represent (be backed by) off-ledger assets on the ledger. An important example of this is *backed stablecoins*. Other noteworthy examples of such assets include video game tokens, as well as service tokens (which represent service provider obligations).

A typical design for such a system is that a trusted party (the "issuer") is responsible for creation and destruction of the asset tokens on the ledger. The issuer is trusted to hold one of the backing off-ledger assets for every token that exists on the ledger, so the only role that the on-chain policy can play is to verify that the forging of the token is signed by the trusted issuer. This can be implemented with a forging policy that enforces an m-out-of-n multi-signature scheme, and no additional clauses:

```
trusted_issuer(msig) = JustMSig(msig)
```

5.3 Vesting

A common desire is to release a supply of some asset on some schedule. Examples include vesting schemes for shares, and staged releases of newly minted tokens. This seems tricky in our simple model: how is the forging policy supposed to know which tranches have already been released without some kind of global state which tracks them? However, this is a problem that we can solve with more tokens. We start building this policy by following the single issuer scheme, but we need to express more.

Given a specific output `o`, and two tranches of tokens `tr1` and `tr2` which should be released after `tick1` and `tick2`, we can write a forging policy such as:

```
vesting = SpendsOutput(o) && Forges({"tr1" ↦ 1, "tr2" ↦ 1})
       || TickAfter(tick1) && Forges(tr1bundle) && Burns({"tr1" ↦ 1})
       || TickAfter(tick2) && Forges(tr2bundle) && Burns({"tr2" ↦ 1})
```

This disjunction has three clauses:

- Once only, you may forge two unique tokens `tranche1` and `tranche2`.
- If you spend and burn `tr1` and it is after `tick1`, then you may forge all the tokens in `tr1bundle`.
- If you spend and burn `tr2` and it is after `tick2`, then you may forge all the tokens in `tr2bundle`.

By reifying the tranches as tokens, we ensure that they are unique and can be used precisely once. As a bonus, the tranche tokens are themselves tradeable.

5.4 Inventory tracker: tokens as state

We can use tokens to carry some data for us, or to represent state. A simple example is inventory tracking, where the inventory listing can only be modified by a set of trusted parties. To track inventory on-chain, we want to have a single output containing all of the tokens of an "inventory tracking" asset. If the trusted keys are represented by the multi-signature `msig`, the inventory tracker tokens should always be kept in a UTXO entry with the following output:

```
(hash(msig) , {hash(msig) ↦ {hats ↦ 3, swords ↦ 1, owls ↦ 2}})
```

The inventory tracker is an example of an asset that should indefinitely be controlled by a specific script (which ensures only authorized users can update the inventory), and we enforce this condition in the forging script itself:

```
inventory_tracker(msig) = JustMSig(msig) && AssetToAddress(_)
```

In this case, `inventory_tracker(msig)` is both the forging script and the output-locking script. The blank value supplied as the argument means that the policy ID (and also the address) are both assumed to be the hash of the `inventory_tracker(msig)` script. Defined this way, our script is run at initial forge time, and any time the inventory is updated. Each time it only validates if all the inventory tracker tokens in the transaction's outputs are always locked by this exact output script.

5.5 Non-fungible tokens

A common case is to want an asset group where *all* the tokens are non-fungible. A simple way to do this is to simply have a different asset policy for each token, each of which can only be run once by requiring a specific UTXO to be spent. However, this is clumsy, and typically we want to have a set of non-fungible tokens all controlled by the same policy. We can do this with the `FreshTokens` clause. If the policy always asserts that the token names are hashes of data unique to the transaction and token, then the tokens will always be distinct.

5.6 Revocable permission

An example where we employ this dual-purpose nature of scripts is revocable permission. We will express permissions as a *credential token*.

The list of users (as a list of hashes of their public keys) in a credential token is composed by some central accreditation authority. Users usually trust that this authority has verified some real-life data, e.g. that a KYC accreditation authority has checked off-chain that those it accredits meet some standard.[7] Note here that we significantly simplify the function of KYC credentials for brevity of our example.

For example, suppose that exchanges are only willing to transfer funds to those that have proved that they are KYC-accredited.

In this case, the accreditation authority could issue an asset that looks like

```
{KYC_accr_authority ↦ {accr_key_1 ↦ 1, accr_key_2 ↦ 1, accr_key_3 ↦ 1}}
```

[7] KYC stands for "know your customer", which is the process of verifying a customer's identity before allowing the customer to use a company's service.

where the token names are the public keys of the accredited users. We would like to make sure that

- only the authority has the power to ever forge or burn tokens controlled by this policy, and it can do so at any time,
- all the users with listed keys are able to spend this asset as on-chain proof that they are KYC-accredited, and
- once a user is able to prove they have the credentials, they should be allowed to receive funds from an exchange.

We achieve this with a script of the following form:

```
credential_token(msig) = JustMSig(msig) && DoForge
                       || AssetToAddress(_) && Not DoForge && SignedByPIDToken(_)
```

Here, forges (i.e. updates to credential tokens) can only be done by the `msig` authority, but every user whose key hash is included in the token names can spend from this script, provided they return the asset to the same script. To make a script that only allows spending from it if the user doing so is on the list of key hashes in the credential token made by `msig`, we write

```
must_be_on_list(msig) = SpendsCur(credential_token(msig))
```

In our definition of the credential token, we have used all the strategies we discussed above to extend the expressivity of an FPS language. We are not yet using the UTXO model to its full potential, as we are just using the UTXO to store some information that cannot be traded. However, we could consider updating our credential token use policy to associate spending it with another action, such as adding a pay-per-use clause. Such a change really relies on the UTXO model.

6 Related work

Ethereum. Ethereum's ERC token standards [15,8] are one of the better known multi-asset implementations. They are a non-native implementation and so come with a set of drawbacks, such as having to implement ledger functionality (such as asset transfers) using smart contracts, rather than using the underlying ledger.

Augmenting the Ethereum ledger with functionality similar to that of the model we present here would likely be possible. Additionally, access to global contract state would make it easier to define forging policies that care about global information, such as the total supply of an asset.

Waves. Waves [16] is an account-based multi-asset ledger, supporting its own smart contract language. In Waves, both accounts and assets themselves can be associated with contracts. In both cases, the association is made by adding the associated script to the account state (or the state of the account containing the asset). A script associated with an asset imposes conditions on the use of this asset, including minting and burning restrictions, as well as transfer restrictions.

Stellar. Stellar [13] is an account-based native multi-asset system geared towards tracking real-world item ownership via associated blockchain tokens. Stellar is optimised to allow a token issuer to maintain a level of control over the use of their token even once it changes hands. The Stellar ledger also features a distributed exchange listing, which is used to facilitate matching (by price) exchanges between different tokens.

Zilliqa. Zilliqa is an account-based platform with an approach to smart contract implementation similar to that of Ethereum [12]. The Zilliqa fungible and non-fungible tokens are designed in a way that mimics the ERC-20 and ERC-721 tokens, respectively. While this system is designed to be statically analysable, it does not offer new solutions to the problem of dependency on the global state.

DAML. DAML [7] is a smart contract language designed to be used on the DAML ledger model. The DAML ledger model does not support keeping records of asset ownership, but instead, only stores current contract states in the following way: a valid transaction interacting with a contract results in the creation of new contracts that are the next steps of the original contract, and removal of the original contract from the ledger.

Only the contracts with which a transaction interacts are relevant to validating it, which is similar to our approach of validation without global context. Although this system does not have built-in multi-asset support (or any ledger-level accounting), the transfer of any type of asset can be represented on the ledger via contracts. Due to the design of the system to operate entirely by listing contracts on the ledger, each action, including accepting funds transferred by a contract, requires consent. This is another significant way in which this model is different from ours.

Bitcoin. Bitcoin popularised UTXO ledgers, but has neither native nor non-native multi-asset support on the main chain. The Bitcoin ledger model does not appear to have the accounting infrastructure or sufficiently expressive smart contracts for implementing multi-asset support in a generic way. There have been several layer-two approaches to implementing custom Bitcoin assets. Because each mined Bitcoin is unique, a particular Bitcoin can represent a specific custom asset, as is done in [3]. A more sophisticated accounting strategy is implemented in another layer-two custom asset approach [11]. There have also been attempts to implement custom tokens using Lightning network channels [1].

Tezos. Tezos [9] is an account-based platform with its own smart contract language. It has been used to implement an ERC-20-like fungible token standard (FA1.2), with a unified token standard in the works (see [14]). The custom tokens for both multi-asset standards are non-native, and thus have shortcomings similar to those of Ethereum token standards.

Nervos CKB. Nervos CKB [17] is a UTXO-inspired platform that operates on a broader notion of a Cell, rather than the usual output balance amount and address, as the value stored in an entry. A Cell entry can contain any type of data, including a native currency balance, or any type of code. This platform comes with a Turing-complete scripting language that can be used to define custom native tokens. There is, however, no dedicated accounting infrastructure to handle trading custom assets in a similar way as the base currency type.

7 Discussion

7.1 General observations

Asset registries and distributed exchanges. The most obvious way to manage custom assets might be to add some kind of global *asset registry*, which associates a new asset group with its policy. Once we have an asset registry, this becomes a natural place to put other kinds of infrastructure that rely on global state associated with assets, such as decentralised exchanges.

However, our system provides us with a way to associate forging policies and the assets controlled by them *without* any global state. This simplifies the ledger implementation in the concurrent and distributed environment of a blockchain. Introducing global state into our model would result in disrupted synchronisation (on which [6] relies to great effect to implement fast, optimistic settlement), as well as slow and costly state updates at the time of asset registration. Hence, on balance we think it is better to have a stateless system, even if it relegates features like decentralised exchanges to be Layer 2 solutions.

Spending policies. Some platforms we discussed provide ways to express restrictions on the *transfer* of tokens, not just on their forging and burning, which we refer to as *spending policies*. Unlike forging policies, spending polices are not a native part of our system. We have considered a number of approaches to adding spending policies, but we have not found a solution that does not put an undue burden on the *users* of such tokens, both humans and programmatic users such as layer-two protocols (e.g. Lightning). For example, it would be necessary to ensure that spending policies are not in conflict with forging or output-locking scripts (any time the asset is spent).

Forging tokens requires a specific action by the user (providing and satisfying a forging policy script), but this action is always taken knowingly by a user who is specifically trying to forge those tokens. *Spending* tokens is, however, a completely generic operation that works over arbitrary bundles of tokens. Indeed, a virtue of our system is that custom tokens all look and behave uniformly. In contrast, spending policies make custom tokens extremely difficult to handle in a generic way, in particular for automated systems.

These arguments do not invalidate the usefulness of spending policies, but instead highlight that they are not obviously compatible with trading of native assets in a generic way, and an approach that addresses these issues in an ergonomic way is much needed. This problem is not ours alone: spending policies in other systems we have looked at here (such as Waves), do not provide universal solutions to the issues we face with spending policies either.

Viral scripts. One way to emulate spending policies in our system is to lock all the tokens with a particular script that ensures that they *remain* locked by the same script when transferred. We call such a script a "viral" script (since it spreads to any new outputs that are "infected" with the token).

This allows the conditions of the script to be enforced on every transaction that uses the tokens, but at significant costs. In particular such tokens can never be locked by a *different* script, which prevents such tokens from being used in smart contracts, as well as preventing an output from containing tokens from two such viral asset groups (since *both* would require that their validator be applied to the output!). In some cases, however, this approach is exactly what we want. For example, in the case of credential tokens, we want the script locking the credential to permanently allow the issuer access to the credential (in order to maintain their ability to revoke it).

Global State. There are limitations of our model due to the fact that global information about the ledger is not available to the forging policy script. Many global state constraints can be accommodated with workarounds (such as in the case of provably unique fresh tokens), but some cannot: for example, a forging policy that allows a variable amount to be forged in every block depending on the current total supply of assets of another policy. This is an odd policy to have, but nevertheless, not one that can be defined in our model.