The Extended UTXO Model

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

[1] IOHK, {manuel.chakravarty, james.chapman, kenneth.mackenzie, orestis.melkonian, michael.peyton-jones}@iohk.io
[2] University of Edinburgh, orestis.melkonian@ed.ac.uk, wadler@inf.ed.ac.uk

Abstract. Bitcoin and Ethereum, hosting the two currently most valuable and popular cryptocurrencies, use two rather different ledger models, known as the *UTXO model* and the *account model*, respectively. At the same time, these two public blockchains differ strongly in the expressiveness of the smart contracts that they support. This is no coincidence. Ethereum chose the account model explicitly to facilitate more expressive smart contracts. On the other hand, Bitcoin chose UTXO also for good reasons, including that its semantic model stays simple in a complex concurrent and distributed computing environment. This raises the question of whether it is possible to have expressive smart contracts, while keeping the semantic simplicity of the UTXO model.

In this paper, we answer this question affirmatively. We present *Extended UTXO* (*EUTXO*), an extension to Bitcoin's UTXO model that supports a substantially more expressive form of *validation scripts*, including scripts that implement general state machines and enforce invariants across entire transaction chains.

To demonstrate the power of this model, we also introduce a form of state machines suitable for execution on a ledger, based on Mealy machines and called Constraint Emitting Machines (CEM). We formalise CEMs, show how to compile them to EUTXO, and show a *weak bisimulation* between the two systems. All of our work is formalised using the Agda proof assistant.

Keywords: blockchain · UTXO · functional programming · state machines.

1 Introduction

Bitcoin, the most widely known and most valuable cryptocurrency, uses a graph-based ledger model built on the concept of *UTXOs* (*unspent transaction outputs*) [2,17]. Individual *transactions* consist of a list of *inputs* and a list of *outputs*, where outputs represent a specific *value* (of a cryptocurrency) that is available to be spent by inputs of subsequent transactions. Each output can be spent by (i.e., connect to) exactly one input. Moreover, we don't admit cycles in these connections, and hence we can regard a collection of transactions spending from each other as a directed acyclic graph, where a transaction with m inputs and n

outputs is represented by a node in the graph with m edges in and n edges out. The sum of the values consumed by a transaction's inputs must equal the sum of the values provided by its outputs, thus value is conserved.

Whether an output can be consumed by an input is determined by a function ν attached to the output, which we call the output's *validator*. A transaction input proves its eligibility to spent an output by providing a *redeemer* object ρ, such that $\nu(\rho) = \mathsf{true}$; redeemers are often called *witnesses* in Bitcoin. In the simplest case, the redeemer is a cryptographic hash of the spending transaction signed by an authorised spender's private key, which is checked by the validator, which embeds the corresponding public key. More sophisticated protocols are possible by using more complex validator functions and redeemers — see [3] for a high-level model of what is possible with the functionality provided by Bitcoin.

The benefit of this graph-based approach to a cryptocurrency ledger is that it plays well with the concurrent and distributed nature of blockchains. In particular, it forgoes any notion of shared mutable state, which is known to lead to highly complex semantics in the face of concurrent and distributed computations involving that shared mutable state.

Nevertheless, the UTXO model, generally, and Bitcoin, specifically, has been criticised for the limited expressiveness of programmability achieved by the validator concept. In particular, Ethereum's *account-based ledger* and the associated notion of *contract accounts* has been motivated by the desire to overcome those limitations. Unfortunately, it does so by introducing a notion of shared mutable state, which significantly complicates the semantics of contract code. In particular, contract authors need to understand the subtleties of this semantics or risk introducing security issues (such as the vulnerability to recursive contract invocations that led to the infamous DAO attack [5]).

Contributions. The contribution of the present paper is to propose an extension to the basic UTXO ledger model, which (a) provably increases expressiveness, while simultaneously (b) preserving the dataflow properties of the UTXO graph; in particular, it forgoes introducing any notion of shared mutable state. More specifically, we make the following contributions:

- We propose the *EUTXO model*, informally in Section 2 and formally in Section 3.
- We demonstrate that the EUTXO model supports the implementation of a specific form of state machines (*Constraint Emitting Machines*, or CEMs), which the basic UTXO model does not support, in Section 4.
- We provide formalisations of both the EUTXO model and Constraint Emitting Machines. We prove a weak bisimulation between the two using the Agda proof assistant[3], building on previous work by Melkonian et al. [11].

Section 5 summarises related work.

The EUTXO model will be used in the ledger of Cardano, a major blockchain system currently being developed by IOHK. It also provides the foundation of

[3] https://github.com/omelkonian/formal-utxo/tree/a1574e6

Cardano's smart contract platform *Plutus*[4], which includes a small functional programming language *Plutus Core* which is used to implement Scripts. Although a technical description of Cardano itself is beyond the scope of this paper, one can try out the Plutus Platform in an online playground.[5]

Other future work includes a formal comparison of EUTXO with Ethereum's account-based model.

2 Extending UTXO

Various forms of state machines have been proposed to characterise smart contract functionality that goes beyond what is possible with the basic UTXO model — see, for example, [8,16] using Ethereum's account-based model. However, we might wonder whether we can extend the basic UTXO model in such a way as to support more expressive state machines without switching to an account-based model.

Given that we can regard the individual transactions in a continuous chain of transactions as individual steps in the evolution of a state machine, we require two pieces of additional functionality from the UTXO model: (a) we need to be able to maintain the machine state, and (b) we need to be able to enforce that the same contract code is used along the entire sequence of transactions — we call this *contract continuity*.

To maintain the machine state, we extend UTXO outputs from being a pair of a validator ν and a cryptocurrency value *value* to being a triple $(\nu, \mathit{value}, \delta)$ of validator, value, and a *datum* δ, where δ contains arbitrary contract-specific data. Furthermore, to enable validators to enforce contract continuity, we pass the entirety of the transaction that attempts to spend the output locked by a validator to the validator invocation. Thus a validator can inspect the transaction that attempts to spend its output and, in particular, it can ensure that the contract output of that transaction uses validator code belonging to the same contract — often, this will be the same validator. Overall, to check that an input with redeemer ρ that is part of the transaction tx is entitled to spend an output $(\nu, \mathit{value}, \delta)$, we check that $\nu(\mathit{value}, \delta, \rho, tx) = \mathsf{true}$.

As we are allowing arbitrary data in δ and we enable the validator ν to impose arbitrary validity constraints on the consuming transaction tx, the resulting Extended UTXO (EUTXO) model goes beyond enabling state machines. However, in this paper we restrict ourselves to the implementation of state machines and leave the investigation of further-reaching computational patterns to future work.

As a simple example of a state machine contract consider an n–of–m multi-signature contract. Specifically, we have a given amount $\mathit{value}_{\mathrm{msc}}$ 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 $\mathit{value}_{\mathrm{msc}}$. With plain UTXO (e.g., on Bitcoin), a multi-signature scheme requires out-of-band (off-chain) communication to collect all

[4] https://github.com/input-output-hk/plutus
[5] https://prod.playground.plutus.iohkdev.io/

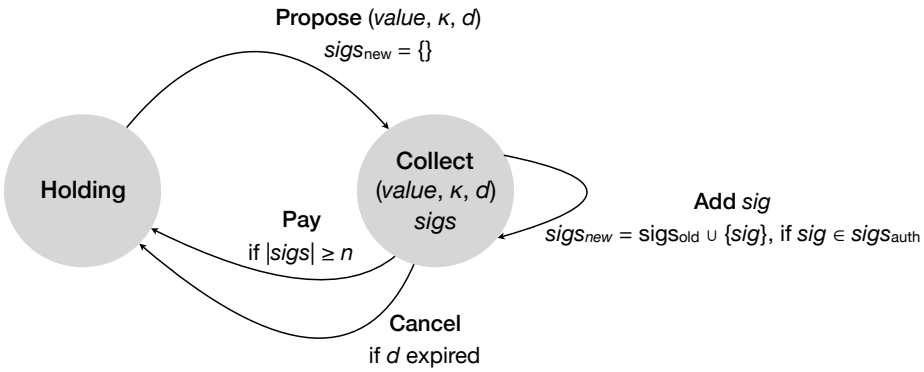

Fig. 1. Transition diagram for the multi-signature state machine; edges labelled with input from redeemer and transition constraints.

n signatures to spend $value_{\text{msc}}$. 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 $sigs_{\text{auth}}$ with $|sigs_{\text{auth}}| = m$ are baked into the contract code.

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

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) Propose$(value, \kappa, d)$ to propose a payment of $value$ to κ by the deadline d, (2) 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.

The increased expressiveness of the EUTXO model goes far beyond simple contracts such as this on-chain multi-signature contract. For example, the complete functionality of the Marlowe domain-specific language for financial contracts [15] has been successfully implemented as a state machine on the EUTXO model.

3 Formal model

3.1 Basic types and notation

Figure 2 defines some basic types and notation used in the rest of the paper; we have generally followed the notation established by Zahnentferner in [17].

$\mathbb{B}$	the type of booleans
$\mathbb{N}$	the type of natural numbers
$\mathbb{Z}$	the type of integers
$(\phi_1 : T_1, \ldots, \phi_n : T_n)$	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

Fig. 2. Basic types and notation

The Data type. We will make particular use of a primitive type Data which can be used to pass information into scripts. This is intended to be any relatively standard structured data format, for example JSON or CBOR [6].

The specific choice of type does not matter for this paper, so we have left it abstract. The intention is that this should be well supported by all the programming languages we are interested in, including whatever language is used for scripts, and whatever languages are used for off-chain code that interacts with the chain.

We assume that for every (non-function) type T in the scripting language we have corresponding toData and fromData functions.

3.2 EUTXO: Enhanced scripting

Our first change to the standard UTXO model is that as well as the validator we allow transaction outputs to carry a piece of data called the *datum* (or *datum object*), which is passed in as an additional argument during validation. This allows a contract to carry some state (the datum) without changing its "code" (the validator). We will use this to carry the state of our state machines (see Section 2).

The second change is that the validator receives some information about the transaction that is being validated. This information, which we call the *context*,

is passed in as an additional argument of type Context. The information supplied in the context enables the validator to enforce much stronger conditions than is possible with a bare UTXO model — in particular, it can inspect the *outputs* of the current transaction, which is essential for ensuring contract continuity (see Section 2).

The third change is that we provide some access to time by adding a *validity interval* to transactions. This is an interval of ticks (see Subsection 3.3) during which a transaction can be processed (a generalisation of a "time-to-live"). Thus, any scripts which run during validation can assume that the current tick is within that interval, but do not know the precise value of the current tick.

Finally, we represent all the arguments to the validator (redeemer, datum, Context) as values of type Data. Clients are therefore responsible for encoding whatever types they would like to use into Data (and decoding them inside the validator script).

3.3 A Formal Description of the EUTXO Model

In this section we give a formal description of the EUTXO model. The description is given in a straightforward set-theoretic form, which (1) admits an almost direct translation into languages like Haskell for implementation, and (2) is easily amenable to mechanical formalisation. We will make use of this in Section 4.

The definitions in this section are essentially the definitions of UTXO-based cryptocurrencies with scripts from Zahnentferner [17], except that we have made the changes described above.

Figure 3 lists the types and operations used in the the basic EUTXO model. Some of these are defined here, the others must be provided by the ledger ("ledger primitives").

Addresses. We follow Bitcoin in referring to the targets of transaction outputs as "addresses". In this system, they refer only to *script* addresses (likely a hash of the script), but in a full system they would likely include public-key addresses, and so on.

Ticks. A tick is a monotonically increasing unit of progress in the ledger system. This corresponds to the "block number" or "block height" in most blockchain systems. We assume that there is some notion of a "current tick" for a given ledger.

Inputs and outputs. Transactions have a Set of inputs but a List of outputs. There are two reasons that we do not also have a Set of outputs although they are conceptually symmetrical:

– We need a way to uniquely identify a transaction output, so that it can be referred to by a transaction input that spends it. The pair of a transaction id and an output index is sufficient for this, but other schemes are conceivable.

Ledger primitives

$\mathsf{Quantity}$	an amount of currency
Tick	a tick
$\mathsf{Address}$	an "address" in the blockchain
Data	a type of structured data
$\mathsf{DataHash}$	the hash of a value of type Data
TxId	the identifier of a transaction
$\mathsf{txId} : \mathsf{Tx} \to \mathsf{TxId}$	a function computing the identifier of a transaction
Script	the (opaque) type of scripts
$\mathsf{scriptAddr} : \mathsf{Script} \to \mathsf{Address}$	the address of a script
$\mathsf{dataHash} : \mathsf{Data} \to \mathsf{DataHash}$	the hash of an object of typeData
$[\![\_]\!] : \mathsf{Script} \to \mathsf{Data} \times \mathsf{Data} \times \mathsf{Data} \to \mathbb{B}$	applying a script to its arguments

Defined types

$$
\begin{aligned}
\mathsf{Output} &= (\mathit{value} : \mathsf{Quantity},\\
&\quad\ \mathit{addr} : \mathsf{Address},\\
&\quad\ \mathit{datumHash} : \mathsf{DataHash})\\[1ex]
\mathsf{OutputRef} &= (\mathit{id} : \mathsf{TxId}, \mathit{index} : \mathbb{N})\\[1ex]
\mathsf{Input} &= (\mathit{outputRef} : \mathsf{OutputRef},\\
&\quad\ \mathit{validator} : \mathsf{Script},\\
&\quad\ \mathit{datum} : \mathsf{Data},\\
&\quad\ \mathit{redeemer} : \mathsf{Data})\\[1ex]
\mathsf{Tx} &= (\mathit{inputs} : \mathsf{Set}[\mathsf{Input}],\\
&\quad\ \mathit{outputs} : \mathsf{List}[\mathsf{Output}],\\
&\quad\ \mathit{validityInterval} : \mathsf{Interval}[\mathsf{Tick}])\\[1ex]
\mathsf{Ledger} &= \mathsf{List}[\mathsf{Tx}]
\end{aligned}
$$

Fig. 3. Primitives and types for the EUTXO model

– A Set requires a notion of equality. If we use the obvious structural equality on outputs, then if we had two outputs paying X to address A, they would be equal. We need to distinguish these — outputs must have an identity beyond just their address and value.

The location of validators and datum objects. Validator scripts and full datum objects are provided as parts of transaction *inputs*, even though they are conceptually part of the output being spent. The output instead specifies them by providing the corresponding address or hash.[6]

This strategy reduces memory requirements, since the UTXO set must be kept in memory for rapid access while validating transactions. Hence it is desir-

[6] That these match up is enforced by Rules 7 and 8 in Figure 6.

able to keep outputs small — in our system they are constant size. Providing the much larger validator script only at the point where it is needed is thus a helpful saving. The same considerations apply to datum objects.

An important question is how the person who spends an output *knows* which validator and datum to provide in order to match the hashes on the output. This can always be accomplished via some off-chain mechanism, but we may want to include some on-chain way of accomplishing this.[7] However, this is not directly relevant to this paper, so we have omitted it.

Fees, forge, and additional metadata. Transactions will typically have additional metadata, such as transaction fees or a "forge" field that allows value to be created or destroyed. These are irrelevant to this paper, so have been omitted.[8]

Ledger structure. We model a ledger as a simple list of transactions: a real blockchain ledger will be more complex than this, but the only property that we really require is that transactions in the ledger have some kind of address which allows them to be uniquely identified and retrieved.

3.4 The Context type

Recall from the introduction to Section 3.2 that when a transaction input is being validated, the validator script is supplied with an object of type Context (encoded as Data) which contains information about the current transaction. The Context type is defined in Figure 4, along with some related types.

$$
\begin{aligned}
\mathsf{OutputInfo} &= (\mathit{value} : \mathsf{Quantity},\\
&\quad\ \mathit{validatorHash} : \mathsf{Address},\\
&\quad\ \mathit{datumHash} : \mathsf{DataHash})\\[1em]
\mathsf{InputInfo} &= (\mathit{outputRef} : \mathsf{OutputRef},\\
&\quad\ \mathit{validatorHash} : \mathsf{Address},\\
&\quad\ \mathit{datum} : \mathsf{Data},\\
&\quad\ \mathit{redeemer} : \mathsf{Data},\\
&\quad\ \mathit{value} : \mathsf{Quantity})\\[1em]
\mathsf{Context} &= (\mathit{inputInfo} : \mathsf{Set}[\mathsf{InputInfo}],\\
&\quad\ \mathit{outputInfo} : \mathsf{List}[\mathsf{OutputInfo}],\\
&\quad\ \mathit{validityInterval} : \mathsf{Interval}[\mathsf{Tick}],\\
&\quad\ \mathit{thisInput} : \mathbb{N})
\end{aligned}
$$

Fig. 4. The Context type for the EUTXO model

[7] Cardano will provide a mechanism in this vein.
[8] Adding such fields might require amending Rule 4 to ensure value preservation.

The contents of Context. The Context type is a summary of the information contained in the Tx type in Figure 3, situated in the context of a validating transaction, and made suitable for consumption in a script. That results in the following changes:

1. The InputInfo type is augmented with information that comes from the output being spent, specifically the value attached to that output.
2. The Context type includes an index that indicates the input currently being validated. This allows scripts to identify their own address, for example.
3. Validators are included as their addresses, rather than as scripts. This allows easy equality comparisons without requiring script languages to be able to represent their own programs.

We assume that there is a function $\mathsf{toContext} : \mathsf{Tx} \times \mathsf{Input} \times \mathsf{Ledger} \to \mathsf{Context}$ which summarises a transaction in the context of an input and a ledger state.

Determinism. The information provided by Context is entirely determined by the transaction itself. This means that script execution during validation is entirely deterministic, and can be simulated accurately by the user *before* submitting a transaction: thus both the outcome of script execution and the amount of resources consumed can be determined ahead of time. This is helpful for systems that charge for script execution, since users can reliably compute how much they will need to pay ahead of time.

A common way for systems to violate this property is by providing access to some piece of mutable information, such as the current time (in our system, the current tick has this role). Scripts can then branch on this information, leading to non-deterministic behaviour. We sidestep this issue with the validation interval mechanism (see the introduction to Section 3.2).

$\mathsf{lookupTx} : \mathsf{Ledger} \times \mathsf{TxId} \to \mathsf{Tx}$
$\mathsf{lookupTx}(l, id)$ = the unique transaction in l whose id is id

$\mathsf{unspentTxOutputs} : \mathsf{Tx} \to \mathsf{Set}[\mathsf{OutputRef}]$
$\mathsf{unspentTxOutputs}(t) = \{(\mathsf{txId}(t), 1), \ldots, (\mathsf{txId}(id), |t.outputs|)\}$

$\mathsf{unspentOutputs} : \mathsf{Ledger} \to \mathsf{Set}[\mathsf{OutputRef}]$
$\mathsf{unspentOutputs}([]) = \{\}$
$\mathsf{unspentOutputs}(t :: l) = (\mathsf{unspentOutputs}(l) \setminus t.inputs) \cup \mathsf{unspentTxOutputs}(t)$

$\mathsf{getSpentOutput} : \mathsf{Input} \times \mathsf{Ledger} \to \mathsf{Output}$
$\mathsf{getSpentOutput}(i, l) = \mathsf{lookupTx}(l, i.outputRef.id).outputs[i.outputRef.index]$

Fig. 5. Auxiliary functions for EUTXO validation

3.5 Validity of EUTXO transactions

Figure 6 defines what it means for a transaction t to be valid for a valid ledger l during the tick $\mathsf{currentTick}$, using some auxiliary functions from Figure 5. Our definition combines Definitions 6 and 14 from Zahnentferner [17], differing from the latter in Rule 6. 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'.

---

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

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

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

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

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

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

4. **Value is preserved**

$$\text{Unless } l \text{ is empty, } \sum_{i \in t.\mathit{inputs}} \mathsf{getSpentOutput}(i, l).\mathit{value} = \sum_{o \in t.\mathit{outputs}} o.\mathit{value}$$

5. **No output is double spent**

$$\text{If } i_1, i_2 \in t.\mathit{inputs} \text{ and } i_1.\mathit{outputRef} = i_2.\mathit{outputRef} \text{ then } i_1 = i_2.$$

6. **All inputs validate**

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

7. **Validator scripts match output addresses**

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

8. **Each datum matches its output hash**

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

Fig. 6. Validity of a transaction t in the EUTXO model

Creating value. Most blockchain systems have special rules for creating or destroying value. These are usually fairly idiosyncratic, and are not relevant to this paper, so we have provided a simple genesis condition in Rule 4 which allows the initial transaction in the ledger to create value.

Lookup failures. The function $\mathsf{getSpentOutput}$ calls $\mathsf{lookupTx}$, which looks up the unique transaction in the ledger with a particular id and can of course fail.

However Rule 3 ensures that during validation all of the transaction inputs refer to existing unspent outputs, and in these circumstances lookupTx will always succeed for the transactions of interest.

4 Expressiveness of EUTXO

In this section, we introduce a class of state machines that can admit a straightforward modelling of smart contracts running on an EUTXO ledger. The class we choose corresponds closely to Mealy machines [9] (deterministic state transducers). The transition function in a Mealy machine produces a value as well as a new state. We use this value to model the emission of constraints which apply to the current transaction in the ledger. We do not claim that this class captures the full power of the ledger: instead we choose it for its simplicity, which is sufficient to capture a wide variety of use cases.

We demonstrate how one can represent a smart contracts using Mealy machines and formalise a *weak bisimulation* between the machine model and the ledger model. Furthermore, we have mechanised our results in Agda[9], based on an executable specification of the model described in Section 3.

4.1 Constraint Emitting Machines

We introduce Constraint Emitting Machines (CEM) which are based on Mealy machines. A CEM consists of its type of states S and inputs I, a predicate function $\mathsf{final} : \mathsf{S} \to \mathsf{Bool}$ indicating which states are final and a valid set of transitions, given as a function $\mathsf{step} : \mathsf{S} \to \mathsf{I} \to \mathsf{Maybe}\ (\mathsf{S} \times \mathsf{TxConstraints})$[10] from source state and input symbol to target state and constraints and denoted $s \xrightarrow{i} (s', tx^{\equiv})$.

The class of state machines we are concerned with here diverge from the typical textbook description of Mealy Machines in the following aspects:

- The set of states can be infinite.
- There is no notion of **initial state**, since we would not be able to enforce it on the blockchain level. Therefore, each contract should first establish some initial trust to bootstrap the process. One possible avenue for overcoming this limitation is to build a notion of *trace simulation* on top of the current relation between single states, thus guaranteeing that only valid sequences starting from initial states appear on the ledger. For instance, this could be used to establish inductive properties of a state machine and carry them over to the ledger; we plan to investigate such concerns in future work.
- While **final states** traditionally indicate that the machine *may* halt at a given point, allowing this possibility would cause potentially stale states to clutter the UTXO set in the ledger. Thus, a CEM final state indicates that the machine *must* halt. It will have no continuing transitions from this point onward and the final state will not appear in the UTXO set. This corresponds to the notion of a *stopped* process [14] which cannot make any transitions.

[9] https://github.com/omelkonian/formal-utxo/tree/a1574e6/Bisimulation.agda

[10] The result may be Nothing, in case no valid transitions exist from a given state/input.

– The set of output values is fixed to *constraints* which impose a certain structure on the transaction that will implement the transition. Our current formalisation considers a limited set of first-order constraints, but these can easily be extended without too many changes in the accompanying proofs.

4.2 Transitions-as-transactions

We want to compile a smart contract $\mathcal{C}$ defined as a CEM into a smart contract that runs on the chain. The idea is to derive a validator script from the step function, using the datum to hold the state of the machine, and the redeemer to provide the transition signal. A valid transition in a CEM will correspond to a single valid transaction on the chain. The validator is used to determine whether a transition is valid and the state machine can advance to the next state. More specifically, this validator should ensure that we are transitioning to a valid target state, the corresponding transaction satisfies the emitted constraints and that there are no outputs in case the target state is final:

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

Note that unlike the step function which returns the new state, the validator only returns a boolean. On the chain the next state is provided with the transaction output that "continues" the state machine (if it continues), and the validator simply validates that the correct next state was provided.[11]

4.3 Behavioural Equivalence

We have explained how to compile state machines to smart contracts but how do we convince ourselves that these smart contracts will behave as intended? We would like to show (1) that any valid transition in a CEM corresponds to a valid transaction on the chain, and (2) that any valid transaction on the chain corresponds to a valid transition. We refer to these two properties as soundness and completeness below.

While state machines correspond to automata, the automata theoretic notion of equivalence — trace equivalence — is too coarse when we consider state machines as running processes. Instead we use bisimulation, which was developed in concurrency theory for exactly this purpose, to capture when processes behave the same [14]. We consider both the state machine and the ledger itself to be running processes.

If the state machine was the only user of the ledger then we could consider so-called strong bisimulation where we expect transitions in one process to correspond to transitions in the other and vice-versa. But, as we expect there to be

[11] A user can run the step function locally to determine the correct next state off-chain.

other unrelated transactions occurring on the ledger we instead consider weak bisimulation where the ledger is allowed to make additional so-called *internal* transitions that are unrelated to the behaviour we are interested in observing.

The bisimulation proof relates steps of the CEM to new transaction submissions on the blockchain. Note that we have a *weak* bisimulation, since it may be the case that a ledger step does not correspond to a CEM step.

Definition 1 (Process relation). *A CEM state s corresponds to a ledger l whenever s appears in the current UTXO set, locked by the validator derived from this CEM:*

$$l \sim s$$

Definition 2 (Ledger step). *Concerning the blockchain transactions, we only consider valid ledgers.*[12] *Therefore, a valid step in the ledger consists of submitting a new transaction tx, valid w.r.t. to the current ledger l, resulting in an extended ledger l':*

$$l \xrightarrow{tx} l'$$

Proposition 1 (Soundness). *Given a valid CEM transition $s \xrightarrow{i} (s', tx^{\equiv})$ and a valid ledger l corresponding to source state s, we can construct a valid transaction submission to get a new, extended ledger l' that corresponds to target state s':*

$$\frac{s \xrightarrow{i} (s', tx^{\equiv}) \quad l \sim s}{\exists tx\ l'\ .\ l \xrightarrow{tx} l'\ \wedge l' \sim s'}\ \text{SOUND}$$

Note. We also require that the omitted constraints are satisfiable in the current ledger and the target state is not a final one, since there would be no corresponding output in the ledger to witness $l' \sim s'$. We could instead augment the definition of correspondence to account for final states, but we have refrained from doing so for the sake of simplicity.

Proposition 2 (Completeness). *Given a valid ledger transition $l \xrightarrow{tx} l'$ and a CEM state s that corresponds to l, either tx is irrelevant to the current CEM and we show that the extended ledger l' still corresponds to source state s, or tx is relevant and we exhibit the corresponding CEM transition $s \xrightarrow{i} (s', tx^{\equiv})$*[13]*:*

$$\frac{l \xrightarrow{tx} l' \quad l \sim s}{l' \sim s\ \vee\ \exists i\ s'\ tx^{\equiv}\ .\ s \xrightarrow{i} (s', tx^{\equiv})}\ \text{COMPLETE}$$

Together, soundness and completeness finally give us weak bisimulation. Note, however, that our notion of bisimulation differs from the textbook one (e.g. in Sangiorgi [14]), due to the additional hypotheses that concern our special treatment of constraints and final states.

[12] In our formal development, we enforce validity statically at compile time.

[13] We cannot provide a correspondence proof in case the target state is final, as explained in the previous note.

5 Related work

Bitcoin Covenants [12] allow Bitcoin transactions to restrict how the transferred value can be used in the future, including propagating themselves to ongoing outputs. This provides contract continuity and allows the implementation of simple state machines. Our work is inspired by Covenants, although our addition of a datum is novel and simplifies the state passing.

The Bitcoin Modelling Language (BitML) [3] is an idealistic process calculus that specifically targets smart contracts running on Bitcoin. The semantics of BitML contracts essentially comprise a (labelled) *transition system*, aka a state machine. Nonetheless, due to the constrained nature of the plain UTXO model without any extensions, the construction is far from straightforward and requires quite a bit of off-chain communication to set everything up. Most importantly, the BitML compilation scheme only concerns a restricted form of state machines, while ours deals with a more generic form that admits any user-defined type of states and inputs. BitML builds upon an abstract model of Bitcoin transactions by the same authors [2]; one of our main contributions is an extended version of such an abstract model, which also accounts for the added functionality apparent in Cardano.

Ethereum and its smart contract language, Solidity [4], are powerful enough to implement state machines, due to their native support for global contract instances and state. However, this approach has some major downsides, notably that contract state is global, and must be kept indefinitely by all core nodes. In the EUTXO model, contract state is localised to where it is used, and it is the responsibility of clients to manage it.

Scilla [16] is a intermediate-level language for writing smart contracts as state machines. It compiles to Solidity and is amendable to formal verification. Since Scilla supports the asynchronous messaging capabilities of Ethereum, Scilla contracts correspond to a richer class of automata, called *Communicating State Transition Systems* [13]. In the future, we plan to formally compare this class of state machines with our own class of CEMs, which would also pave the way to a systematic comparison of Ethereum's account-based model against Cardano's UTXO-based one.

Finally, there has been an attempt to model Bitcoin contracts using *timed automata* [1], which enables semi-automatic verification using the UPPAAL model checker [7]. While this provides a pragmatic way to verify temporal properties of concrete smart contracts, there is no formal claim that this class of automata actually corresponds to the semantics of Bitcoin smart contracts. In contrast, our bisimulation proof achieves the bridging of this semantic gap.

References

1. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, Ł.: Modeling bitcoin contracts by timed automata. In: International Conference on Formal Modeling and Analysis of Timed Systems. pp. 7–22. Springer (2014)

2. Atzei, N., Bartoletti, M., Lande, S., Zunino, R.: A formal model of Bitcoin transactions. In: Meiklejohn and Sako [10], pp. 541–560. https://doi.org/10.1007/978-3-662-58387-6_29, https://doi.org/10.1007/978-3-662-58387-6_29
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. Ethereum Foundation: Solidity documentation. https://solidity.readthedocs.io/ (2016–2019)
5. Falkon, S.: The story of the DAO — its history and consequences. https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee (2017), medium.com
6. IETF: RFC 7049 - Concise Binary Object Representation (CBOR). https://tools.ietf.org/html/rfc7049 (Oct 2013), accessed: 2020-01-01
7. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International journal on software tools for technology transfer **1**(1-2), 134–152 (1997)
8. Mavridou, A., Laszka, A.: Designing secure Ethereum smart contracts: A finite state machine based approach. In: Meiklejohn and Sako [10], pp. 523–540. https://doi.org/10.1007/978-3-662-58387-6_28, https://doi.org/10.1007/978-3-662-58387-6_28
9. Mealy, G.H.: A method for synthesizing sequential circuits. The Bell System Technical Journal **34**(5), 1045–1079 (1955)
10. Meiklejohn, S., Sako, K. (eds.): Financial Cryptography and Data Security — 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26–March 2, 2018, Revised Selected Papers, Lecture Notes in Computer Science, vol. 10957. Springer (2018). https://doi.org/10.1007/978-3-662-58387-6, https://doi.org/10.1007/978-3-662-58387-6
11. Melkonian, O., Swierstra, W., Chakravarty, M.M.: Formal investigation of the Extended UTxO model (Extended Abstract). https://omelkonian.github.io/data/publications/formal-utxo.pdf (2019)
12. Möser, M., Eyal, I., Sirer, E.G.: Bitcoin covenants. In: International Conference on Financial Cryptography and Data Security. pp. 126–141. Springer (2016)
13. Nanevski, A., Ley-Wild, R., Sergey, I., Delbianco, G.A.: Communicating state transition systems for fine-grained concurrent resources. In: European Symposium on Programming Languages and Systems. pp. 290–310. Springer (2014)
14. Sangiorgi, D.: Introduction to Bisimulation and Coinduction. Cambridge University Press (2012)
15. Seijas, P.L., Thompson, S.J.: Marlowe: Financial contracts on blockchain. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice — 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5–9, 2018, Proceedings, Part IV. Lecture Notes in Computer Science, vol. 11247, pp. 356–375. Springer (2018). https://doi.org/10.1007/978-3-030-03427-6_27, https://doi.org/10.1007/978-3-030-03427-6_27
16. 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)
17. Zahnentferner, J.: An abstract model of UTxO-based cryptocurrencies with scripts. IACR Cryptology ePrint Archive **2018**, 469 (2018), https://eprint.iacr.org/2018/469