Chimeric Ledgers: Specifying, Translating and Unifying UTxO-based and Account-based Cryptocurrency Accounting

Joachim Zahnentferner
Input Output HK
chimeric.ledgers@protonmail.com

Abstract

Cryptocurrencies are historically divided in two broad groups with respect to the style of transactions that they accept. In the account-based style, each address is seen as an account with a balance, and transactions are transfers of value from one account to another. In the UTxO-based style, transactions inductively spend outputs generated by previous transactions and create new unspent outputs, and there is no intrinsic notion of account associated with an address. Each style has advantages and disadvantages. This paper formally defines: the two styles; translations that allow to simulate one style by the other; new transaction types that allow both styles of transactions to co-exist on the same ledger; and a new transaction type that combines features from both styles.

1 Introduction

Cryptocurrencies offer an alternative to central bank fiat money where new money is generated algorithmically and transparently and people may transact electronically without having to rely on banks or other centralized payment systems. These characteristics were particularly attractive in the context of the global financial crisis, when people's trust on centralized financial institutions was shared, and will continue to be relevant whenever the risk of arbitrarily inflationary policies and privacy violations are present.

However, a major challenge that needs to be addressed for cryptocurrencies themselves to be trusted and massively adopted by people is the assurance of software correctness. There

The author is grateful to Alexander Chepurnoy, Duncan Coutts, Edsko de Vries, Philipp Kant and Philip Wadler and Bruno Woltzenlogel Paleo for fruitful discussions on drafts of this paper.

are already billions of dollars being transacted through cryptocurrency and it would be imprudent to rely such high stakes on potentially flawed software. The decentralized nature of cryptocurrency exacerbates this problem because, when failures occur or flaws are exploited, there is no one to blame.

In order to judge whether the implementation of a cryptocurrency's software components are correct, we must first have specifications of what the implementations should do. Only then we can test the implementations to check whether they really do what the specifications require. Nevertheless, most cryptocurrencies today adopt an approach where the implementation is its own specification. While this may seem convenient, it is unsatisfactory. A production-ready implementation in a concrete programming language often has (and needs to have) optimizations and details that are irrelevant to a particular aspect of the desired behaviour. A high-level programming-language-agnostic mathematical specification, on the other hand, allows the desired behaviour to be defined in a way that is more concise and more widely understandable, focusing on details that matter and abstracting away what is irrelevant in a given context.

Due to their precision and focus on the definition of concepts, specifications may also fulfill an educational purpose, which is valuable for an expanding industry and field of research. Many software engineers joining cryptocurrency companies and researchers bringing their specialties from other fields to the cryptocurrency domain currently need to learn concepts by studying informal, imprecise and incomplete discussions in the Internet or by looking at source codes. Specifications can make their learning curve less steep.

This paper focuses on the accounting aspect of two different types of fund transfer transactions commonly used in cryptocurrency ledgers nowadays. The first one, known as UTxO-*based* (because it tracks *unspent transaction outputs*), was introduced by Bitcoin [1], whereas the second one, called *account-based*, is used by Ethereum [2, 3]. Their semantics is mathematically specified by defining how they should affect balances. Notions such as cryptographic signatures, blocks and monetary policies are orthogonal to the accounting aspect and are intentionally abstracted away. The rationale for this decision is to provide an independent accounting foundation on top of which a wide range of monetary policies and cryptographic authorizations schemes could be defined in future work.

There have been many informal discussions comparing both accounting approaches, often presenting reasons to choose one instead of the other (e.g. [4–8]). This paper takes a neutral stance on this debate. Assuming that each approach has its own advantages and disadvantages, it is likely that they will co-exist in the foreseeable future. This paper aims to facilitate their co-existence, and its main proposal is the notion of *chimeric ledger*, which can contain both UTxO-based and account-based transactions. Translations between the two types of transactions are also defined.

The contributions of this paper are organized as follows:

- specification of UTxO-based transactions (Sec. 3).
- specification of account-based transactions (Sec. 4).
- translations between both approaches (Sec. 5).
- specification of new transaction types that allow users to convert their funds between the UTxO-based and account-based styles of accounting, thereby enabling chimeric ledgers (Sec. 7).
- specification of a new transaction type, combining features of UTxO-based transaction and account-based transaction (Sec. 8).

In order to widen the specification's educational reach and ease its use for assuring software correctness, an explicit effort has been made to write a specification that only requires basic mathematical skills to be understood. The successful achievement of these goals is evidenced by the use of an earlier version of this paper to on-board newly hired software developers and formal methods specialists at Input Output HK and by the many works that took this earlier version as a starting point, such as: a UTxO-wallet specification [1]; the implementation of a wallet back-end and its tests[2]; an approach to input selection that avoids the dust problem [9]; a Coq formalization of the wallet specification[3]; the specification[4] of stake delegation in Cardano [5]; the specification of Cardano's ledger[6] and its prototype implementation[7]; the extended UTxO model[8] needed for Cardano's smart contract language Plutus[9], to which the financial contract DSL Marlowe [10] shall be compiled; A wallet emulator for Plutus contracts[10]; reference implementations of this specification in Scala[11], Haskell[12] and Python[13]; the implementation in IOHK's enterprise blockchain framework[14]; and an extension of this specification with validator and redeemer scripts [11].

[1] github.com/input-output-hk/utxo-wallet-specification/

[2] github.com/input-output-hk/cardano-wallet

[3] github.com/polinavino/cardano-sl/pull/1/commits and github.com/polinavino/cardano-sl/blob/04fd55a4e90ee343110d3a51c5a52600bf8ea44a/wallet-new/coq-formal-spec/coq-wallet-formal-spec.tex

[4] github.com/input-output-hk/fm-ledger-rules/tree/master/docs/delegation_design_spec

[5] github.com/input-output-hk/PoS-Ledger-Delegation

[6] github.com/input-output-hk/fm-ledger-rules

[7] github.com/input-output-hk/cardano-chain

[8] github.com/input-output-hk/plutus/tree/master/docs/extended-utxo

[9] iohk.io/blog/smart-contracts-language-for-cardano-launches-at-plutusfest/

[10] github.com/input-output-hk/plutus/tree/master/wallet-api/src/Wallet/Emulator

[11] github.com/input-output-hk/chimeric-ledgers-spec-scala

[12] github.com/input-output-hk/chimeric-ledgers-spec-haskell

[13] github.com/safeada/python-cardano/blob/master/cardano/chimeric_ledger.py

[14] github.com/input-output-hk/cardano-enterprise

2 Preliminaries

A *blockchain* is, as the name suggests, an immutable chain of blocks of data. Blocks are continuously appended to the chain by mutually distrusting peers. When peers disagree with each other (e.g. on which block to append), a fork occurs, leading to competing versions of the chain that differ from each other on their most recent blocks. Forks are quickly corrected, because peers follow a consensus protocol to decide which competing version they should keep.

Cryptocurrencies are blockchain systems where the block data are transactions transferring values between addresses. In such systems, the blockchain functions as a decentralized ledger, maintained by peers reliably without a central trusted authority. Peers are responsible for validating and grouping available unprocessed transactions into a new block and then trying to append the new block to the chain according to the consensus protocol. Typically, there is an economic incentive for this work: if a peer manages to append a block, it receives all the fees allocated by the transactions in the block and possibly an additional subsidy of newly forged money. The forging of new money depends on the monetary policy of the cryptocurrency.

In the formalism presented here, the technical details of the underlying blockchain are disregarded, because only the ledger of transactions (i.e. the data stored in the blockchain) is of interest. In particular, it does not matter whether the ledger is stored in a single chain or in multiple chains (e.g. a main chain and various sidechains). Furthermore, as the focus is on the accounting aspects only, the formalism abstracts away features of cryptocurrencies such as signatures, scripts and smart contracts. In particular, it does not matter whether the transactions transfer value directly (as in Bitcoin) or call smart contracts that indirectly execute the transfers of value (e.g. as in ERC20 tokens [12] running on Ethereum). For convenience, a *ledger* is assumed to be simply a list of transactions. Nevertheless, the formalism described here is still applicable to cryptocurrencies that store transactions in graphs (e.g. IOTA [13]), as long as the graph nodes can be topologically sorted into a list (as is the case if there are no circular dependencies between transactions). It is possible and perhaps helpful, though not necessary, to associate a *state* to a ledger and think of any *transaction* as a state transition function. The following sections define various subtypes of $\mathsf{Transaction}$.

Definition 1. A *ledger* is a list of valid transactions:

$$\mathsf{Ledger} \stackrel{\mathsf{def}}{=} \mathsf{List}[\mathsf{Transaction}]$$

Notations: A record data type with fields $\varphi_1, \ldots, \varphi_n$ of types $T_1, \ldots, T_n$ is denoted $(\varphi_1 \colon T_1, \ldots, \varphi_n \colon T_n)$. If t is a value of a record data type T and φ is the name of a field of T, then $t.\varphi$ denotes the value of φ for t. A list λ of type $\mathsf{List}[T]$ is either the empty list $[]$ or a

list $e :: \lambda'$ with *head* e of type T and *tail* λ' of type $\mathsf{List}[T]$. $[e_1, \ldots, e_n]$ is an abbreviation for $e_1 :: \ldots :: e_n :: []$. $\lambda(i)$ denotes the i-th element of λ (with the head being the 0-th element, by convention). The concatenation of two lists λ_1 and λ_2 is denoted $\lambda_1 ::: \lambda_2$. The length of a list λ is denoted $|\lambda|$. A list of integers from n to m, including n and m, is denoted $[n..m]$. The standard equality symbol ($=$) is used to state that two values are equal. The definitional equality symbol ($\stackrel{\text{def}}{=}$) is used to define the new constant or function symbol on the left term. The explanatory equality symbol ($\doteq$) is used to explain an introduced value: a sentence such as "the value $v \doteq (n, m)$ is ..." should be read as "the value v, which is of the form (n, m), is ...". To ease readability, integer values representing amounts of money are preceded by \$ and values representing addresses are preceded by @. An anonymous function that takes a tuple as argument may be denoted as $(a_1, \ldots, a_n) \Rightarrow \ldots$. For instance, the k-th projection of a tuple may be denoted $(a_1, \ldots, a_n) \Rightarrow a_k$. The cryptographic collision-resistant hash of an object c is denoted $c^{\#}$. The assignment of a value c to a variable v is denoted $v \leftarrow c$.

3 UTxO-Based Cryptocurrencies

In a time where bank accounts are ubiquitous in financially advanced societies, it is intuitive and natural to imagine transactions as transfers of value between accounts. This, however, is not how Bitcoin's UTxO-based transactions work. To intuitively understand UTxO-based transactions, which are formally described in Definition 2, it is helpful to think of them in terms of cash transactions[15], where a buyer pays with coins and notes of various denominations, with a few key improvements enabled by Bitcoin's digital nature:

- With cash, denominations are fixed (e.g. notes of \$1, \$5, \$10, \$50, \$100 and coins of 50c, 10c, 5c, ...). In Bitcoin, denominations are arbitrary.
- With cash, the coins and notes remain intact as they change hands during a transaction. With bitcoin, the "coins" of arbitrary denominations spent by a transaction are essentially destroyed (i.e. spent as inputs) and new "coins" of equal total value are created (i.e. the new unspent outputs).
- With cash, if the buyer doesn't have coins and notes with the exact value of the item he wants to purchase, he/she must pay with a value greater than the value of the item and then hope that the seller will give back any surplus (i.e. the "change") shortly thereafter in a second transaction, instead of running away with it. With Bitcoin, the buyer can create a "coin" (i.e. an output) with value equal to the surplus and send it back to himself/herself in the same transaction where the payment is made.

[15] Bitcoin's cash-like approach can be appreciated from a historical and ideological point of view, considering that Bitcoin was created with a strong anti-bank sentiment, as evidenced by Satoshi Nakamoto's famous message ("The Times 3 January 2009 Chancellor on brink of second bailout for banks") in Bitcoin's first block.

- A Bitcoin transaction also has a fee indirectly paid to whoever includes it in the ledger, and may (under certain conditions) forge value out of nothing, whereas cash relies on a centralized mint to forge new coins and notes.

Definition 2. The datatype for UTxO-*based transactions* is defined[16] as:

$$\begin{aligned}\mathsf{UtxoTx} \stackrel{\mathsf{def}}{=} (&\mathit{inputs}\colon \mathsf{Set}[\mathsf{Input}],\\ &\mathit{outputs}\colon \mathsf{List}[\mathsf{Output}],\\ &\mathit{forge}\colon \mathsf{Value}, \mathit{fee}\colon \mathsf{Value})\end{aligned}$$

The datatype for *outputs* is:

$$\mathsf{Output} \stackrel{\mathsf{def}}{=} (\mathit{address}\colon \mathsf{Address}, \mathit{value}\colon \mathsf{Value})$$

where value is the value[17] of the output and $\mathit{address}$ is the address that owns it. The datatype for *inputs* is:

$$\mathsf{Input} \stackrel{\mathsf{def}}{=} (\mathit{id}\colon \mathsf{Id}, \mathit{index}\colon \mathsf{Int})$$

where id is the id[18] of a previous transaction to which this input refers, and index indicates which of the referred transaction's outputs should be spent.

The transaction output to which an input refers in a ledger can be retrieved with the functions of Definition 3. Because the input may refer to a transaction (or its output) that cannot be found in the ledger, all these functions have return (monadic) types of the form $\mathsf{Option}[X]$ for some X, whose values may be either none or $\mathrm{some}(x)$ for any x of type X.

Definition 3. The function $\mathrm{tx}\colon \mathsf{Input} \to \mathsf{Ledger} \to \mathsf{Option}[\mathsf{UtxoTx}]$, applied to an input i and a ledger λ, retrieves a transaction t contained in λ such that $t^{\#} = i.\mathit{id}$, if such a t exists. The function $\mathrm{out}\colon \mathsf{Input} \to \mathsf{Ledger} \to \mathsf{Option}[\mathsf{Output}]$ returns $\mathrm{tx}(i).\mathit{get}.\mathit{inputs}(i.\mathit{index})$, if this exists. And finally, the function $\mathrm{value}\colon \mathsf{Input} \to \mathsf{Ledger} \to \mathsf{Option}[\mathsf{Value}]$, returns $\mathrm{out}(i, \lambda).\mathit{get}.\mathit{value}$, if this exists.

UTxO-based cryptocurrencies rely on the notion of unspent transaction outputs. This is usually imagined to be a set of all the outputs of all the ledger's transactions that have not

[16] Normally, a transaction would also include signatures or authorization scripts that determine whether the inputs are allowed to spend the outputs to which they refer. For the reasons discussed in Sec. 2, such authorization details are abstracted away here.

[17] The types Address and Value are regarded here as aliases for unbounded unsigned non-negative integers.

[18] Values of type Id are typically cryptographic collision-resistant hashes of the transactions to which they refer. Whenever the number of potential transactions (which is typically infinite) is larger than the number of possible values of type Id (which is typically, but not necessarily, finite), the pigeonhole principle guarantees that at least two transactions will be mapped to the same id. Nevertheless, a collision-resistant hash function ensures that it is hard to construct a transaction that will have the same id as another.

been spent yet. Unfortunately, this is problematic. As an output is essentially just a pair of an address and a value, there might be several transactions having equal outputs. A set would treat all these equal outputs as the same object. A multi-set would not solve the problem either, because it would not be clear which copy of an output belongs to which transaction. An element of the set of unspent transaction outputs needs to be a reference to a transaction and an output in this transaction, and this is essentially what an input is. Therefore it is helpful to define unspent transaction outputs as a set of "spendable" inputs.

Definition 4. The *unspent outputs* of a transaction can be computed by applying the following function:

$$\begin{aligned}
&\mathrm{unspentOutputs}\colon \mathsf{UtxoTx} \to \mathsf{Set}[\mathsf{Input}] \\
&\mathrm{unspentOutputs}(t) \stackrel{\mathsf{def}}{=} (map \\
&\qquad\qquad ((o,i) \Rightarrow \mathsf{Input}(t^{\#}, i)) \\
&\qquad\qquad t.outputs.zipWithIndex \\
&\qquad\quad).toSet
\end{aligned}$$

where: $zipWithIndex$ augments the outputs with their respective indexes, the anonymous function maps an output to a spendable input consisting of the transaction's hash and the output's index, and $toSet$ converts the list to a set.

The *outputs spent* by a transaction are simply the transaction's inputs, and can be computed by applying the following function:

$$\begin{aligned}
&\mathrm{spentOutputs}\colon \mathsf{UtxoTx} \to \mathsf{Set}[\mathsf{Input}] \\
&\mathrm{spentOutputs}(t) \stackrel{\mathsf{def}}{=} t.inputs
\end{aligned}$$

In a ledger containing only UTxO-based transactions, the relevant information for the ledger's state is its set of unspent outputs, which can be computed by starting with an empty set for the empty ledger, and then updating it for every added transaction by removing the outputs spent by the transaction and adding the unspent outputs generated by the transaction.

Definition 5. The *set of unspent outputs* of a ledger can be computed by applying the following function:

$$\begin{aligned}
&\mathrm{unspentOutputs}\colon \mathsf{Ledger} \to \mathsf{Set}[\mathsf{Input}] \\
&\mathrm{unspentOutputs}([]) \stackrel{\mathsf{def}}{=} \emptyset \\
&\mathrm{unspentOutputs}(t::\lambda) \stackrel{\mathsf{def}}{=} \mathrm{unspentOutputs}(\lambda) \\
&\qquad\qquad\qquad - \mathrm{spentOutputs}(t) \\
&\qquad\qquad\qquad + \mathrm{unspentOutputs}(t)
\end{aligned}$$

For a UTxO-based transaction to be valid, all its inputs must refer to unspent outputs in the ledger and value must be preserved, as described in Definition 6.

Definition 6. A UTxO-based transaction t is *valid* for a ledger λ iff the following two conditions hold:

all inputs refer to unspent outputs:

$$\forall i \in t.inputs, i \in \text{unspentOutputs}(\lambda)$$

value is preserved:

$$t.forge + \sum_{i \in t.inputs} value(i, \lambda).get = t.fee + \sum_{o \in t.outputs} o.value$$

The balance of an address in a ledger can be defined as the sum of the values of outputs that have been paid to the address and that have not been spent yet. To simplify the definition, it is useful to firstly consider the balance of an address in a single transaction.

Definition 7. The UTxO-*balance* of an address a in a valid transaction t w.r.t. a ledger[19] λ is:

$$\mathcal{B}_{\text{UTxO}} \colon \mathsf{Address} \to \mathsf{UtxoTx} \to \mathsf{Ledger} \to \mathsf{Value}$$

$$\mathcal{B}_{\text{UTxO}}(a, t, \lambda) \stackrel{\mathsf{def}}{=} \sum_{\substack{o \in t.outputs \\ o.address = a}} o.value - \sum_{\substack{i \in t.inputs \\ o' = \text{out}(i,\lambda).get \\ o'.address = a}} o'.value$$

Now the balance of an address in a ledger can be defined inductively, treating each transaction's balance for the address as a balance update and overloading the definition of $\mathcal{B}_{\text{UTxO}}$ through ad-hoc polymorphism.

Definition 8. The UTxO-*balance* of an address a in a ledger λ is:

$$\mathcal{B}_{\text{UTxO}} \colon \mathsf{Address} \to \mathsf{Ledger} \to \mathsf{Value}$$

$$\mathcal{B}_{\text{UTxO}}(a, []) \stackrel{\mathsf{def}}{=} 0$$
$$\mathcal{B}_{\text{UTxO}}(a, t::\lambda) \stackrel{\mathsf{def}}{=} \mathcal{B}_{\text{UTxO}}(a, \lambda) + \mathcal{B}_{\text{UTxO}}(a, t, \lambda)$$

The definitions above are illustrated in Example 1.

[19] A ledger needs to be passed as an additional argument, because the transaction does not know the value of its inputs.

Example 1. The following are examples of UTxO-based transactions:

$$t_1 \stackrel{\text{def}}{=} \mathsf{UtxoTx}(\emptyset, [\mathsf{Output}(@1, \$1000)], \$1000, \$0)$$

$$t_2 \stackrel{\text{def}}{=} \mathsf{UtxoTx}(\{\mathsf{Input}(t_1^{\#}, 0)\}, [\mathsf{Output}(@2, \$800), \mathsf{Output}(@1, \$200)], \$0, \$0)$$

$$t_3 \stackrel{\text{def}}{=} \mathsf{UtxoTx}(\{\mathsf{Input}(t_2^{\#}, 1)\}, [\mathsf{Output}(@3, \$199)], \$0, \$1)$$

$$t_4 \stackrel{\text{def}}{=} \mathsf{UtxoTx}(\{\mathsf{Input}(t_3^{\#}, 0)\}, [\mathsf{Output}(@2, \$207)], \$10, \$2)$$

$$t_5 \stackrel{\text{def}}{=} \mathsf{UtxoTx}(\{\mathsf{Input}(t_4^{\#}, 0), \mathsf{Input}(t_2^{\#}, 0)\}, [\mathsf{Output}(@2, 500), \mathsf{Output}(@3, 500)], \$0, \$7)$$

$$t_6 \stackrel{\text{def}}{=} \mathsf{UtxoTx}(\{\mathsf{Input}(t_5^{\#}, 0), \mathsf{Input}(t_5^{\#}, 1)\}, [\mathsf{Output}(@3, 999)], \$0, \$1)$$

Transaction t_1 forges \$1000, creates a \$1000 coin and assigns it to address @1. In transaction t_2, @1 spends it, giving an \$800 coin to @2 and a \$200 coin back to itself. In transaction t_3, besides a coin to @3, a fee or tax of \$1 is paid. In t_4, @3 spends its \$199 coin, forges \$10, pays a \$2 fee and creates a \$207 coin for @2. In t_5, the \$207 and \$800 coins of @2 are spent and two \$500 coins are created, one of them owned by @3 and the other owned by @2. In t_6 these two coins are spent, a new \$999 coin for @3 is created and a \$1 fee is paid. Each of these transactions is valid for a ledger consisting of the previous transactions, in order. In the final ledger $\lambda \doteq [t_6, t_5, t_4, t_3, t_2, t_1]$, the unspent outputs are: $\mathrm{unspentOutputs}(\lambda) \doteq \{\mathsf{Input}(t_6^{\#}, 0)\}$. The balance of @3 is $\mathcal{B}_{\text{UTxO}}(@3, \lambda) \doteq 999$; the balance of any other address is zero. A total of \$1010 has been forged and a total of \$11 has been destroyed in fees.

Remarks about Bitcoin: In Bitcoin's implementation [14], *forge* and *fee* are implicit. The fee is assumed to be the total value of inputs minus the total value of outputs in a transaction with at least one input or zero otherwise, and the forged amount is supposed to be equal to the total value of the outputs in a *coinbase* transaction without inputs or zero otherwise. The framework presented here is slightly more flexible and general, because a transaction may simultaneously have non-zero fee and forge values and may have non-zero forge values even with a non-empty set of inputs (as illustrated in transaction t_4 in Example 1).

4 Account-Based Cryptocurrencies

Whereas Bitcoin, with its UTxO-based approach, aimed to be an account-less electronic cash system, other cryptocurrencies, such as Ethereum and all ERC-20 tokens, opted for cash-less account-based systems. An address is seen as an account, and a transaction transfers a value

from one account to another. Every transaction also contains a unique nonce to protect against replay attacks[20]. To allow transactions that just create money and assign it a receiver or that just take money from a sender and spend it as fee, both the sender and the receiver are optional.

Definition 9. The datatype for *account-based transaction* is defined as:

$$\begin{aligned}\mathsf{AccTx} \stackrel{\text{def}}{=} (&sender\colon \mathsf{Option}[\mathsf{Address}],\\ &receiver\colon \mathsf{Option}[\mathsf{Address}],\\ &value\colon \mathsf{Value}, forge\colon \mathsf{Value}, fee\colon \mathsf{Value},\\ &nonce\colon \mathsf{Int})\end{aligned}$$

with the following requirement[21]:
value is preserved:

$$t.forge + \sum_{s \in s.sender} (t.value + t.fee - t.forge)$$
$$\|$$
$$t.fee + \sum_{r \in t.receiver} t.value$$

The balance of an address in a transaction is what the address receives (if it is the receiver) minus what it spends (if it is the sender).

Definition 10. The *account-balance* of an address a in a transaction t is:

$$\mathcal{B}_{\text{Acc}}\colon \mathsf{Address} \to \mathsf{AccTx} \to \mathsf{Value}$$

$$\begin{aligned}\mathcal{B}_{\text{Acc}}(a,t) &\stackrel{\text{def}}{=} received - spent\\ &\text{where :}\\ received &= \text{if } (a \in t.receiver)\ t.value \text{ else } 0\\ spent &= \text{if } (a \in t.sender)\ t.value + t.fee - t.forge\\ &\quad \text{else } 0\end{aligned}$$

As before, the balance of an address in a ledger can be defined inductively, treating each transaction's balance for the address as a balance update.

[20] Without a unique id, nothing would prevent the receiver of an authorized transaction from including it multiple times in the ledger.

[21] It trivially holds when both sender and receiver are defined.

Definition 11. The *account-balance* of an address a in a ledger λ is:

$$\mathcal{B}_{\text{Acc}} \colon \mathsf{Address} \to \mathsf{Ledger} \to \mathsf{Value}$$

$$\mathcal{B}_{\text{Acc}}(a, []) \stackrel{\mathsf{def}}{=} 0$$
$$\mathcal{B}_{\text{Acc}}(a, t::\lambda) \stackrel{\mathsf{def}}{=} \mathcal{B}_{\text{Acc}}(a, \lambda) + \mathcal{B}_{\text{Acc}}(a, t, \lambda)$$

Definition 12. An account-based transaction t is *valid* for a ledger λ iff the following two conditions hold:

sender has enough money:

$$\forall s \in t.sender, \mathcal{B}_{\text{Acc}}(s, \lambda) \geq t.value + t.fee - t.forge$$

transaction is unique: $\forall t' \in \lambda, t' \neq t$

The definitions above are illustrated in Example 2.

Example 2. The following are examples of account-based transactions:

$$t_1 \stackrel{\mathsf{def}}{=} (\text{none}, \text{some}(@1), \$1000, \$1000, \$0, 0)$$
$$t_2 \stackrel{\mathsf{def}}{=} (\text{some}(@1), \text{some}(@2), \$800, \$0, \$0, 0)$$
$$t_3 \stackrel{\mathsf{def}}{=} (\text{some}(@1), \text{some}(@3), \$199, \$0, \$1, 0)$$
$$t_4 \stackrel{\mathsf{def}}{=} (\text{some}(@3), \text{some}(@2), \$207, \$10, \$2, 0)])$$
$$t_5 \stackrel{\mathsf{def}}{=} (\text{some}(@2), \text{some}(@3), \$500, \$0, \$7, 0)])$$
$$t_6 \stackrel{\mathsf{def}}{=} (\text{some}(@2), \text{some}(@3), \$499, \$0, \$1, 0)])$$

The transactions listed above achieve the same effects as the transactions in Example 1. In the final ledger $\lambda \doteq [t_6, t_5, t_4, t_3, t_2, t_1]$, $\mathcal{B}_{\text{Acc}}(@3, \lambda) \doteq 999$.

Remarks about Ethereum: Note that Ethereum has a more strict condition for the transaction's nonce. Whereas here it is only required that there be no $t' \in \Lambda'$ such that $t' = t$, Ethereum requires that $t.nonce = t^*.nonce + 1$, if t^* is the most recent transaction in Λ' such that $t^*.sender = t.sender$. The weaker condition used here is sufficient to prevent replays and allows transactions to be processed in any order. Ethereum's condition, on the other hand, gives users control over the order in which their transactions should be executed and can be more efficiently checked, as it only requires a comparison with the most recent transaction from the same sender already in the ledger. The weaker condition is adopted here, because it is more general and hence encompasses also potential account-based cryptocurrencies that are less strict than Ethereum. In Ethereum, ethers can be transferred either

directly through an externally created message or through a message created during the execution of a contract. Ethereum uses the word "transaction" only for the former.
Remarks about ERC-20 Tokens: In ERC-20 tokens, value is transferred by sending to the Ethereum ledger a transaction that calls the 'transfer' function of the token's contract. Although the Ethereum transaction that wraps the function call has a fee and a nonce, the transfer function itself has no fee, forge or nonce arguments. Therefore, the notion of ERC-20 transfer is mostly less general than this paper's notion of account-based transaction. However, because an ERC-20 transfer pays fees in a different currency (i.e. ethers), the definitions discussed here would have to be generalized (straightforwardly) to a multi-currency scenario to fully simulate ERC-20.

5 Translations

As illustrated by examples 1 and 2, it is possible to have UTxO-based and account-based transactions that have the same effect. The following two subsections describe how to translate transactions from one style to another in general. The translations produce equivalent transactions in the sense of Definition 13.

Definition 13. A list[22] of transactions $\ell \doteq [t_n, \ldots, t_1]$ on a ledger λ is *equivalent* to a list of transactions $\ell' \doteq [t'_m, \ldots, t'_1]$ on a ledger λ' iff the following hold:

equal forged amounts: **equal fees:**

$$\sum_{t \in \ell} t.forge = \sum_{t' \in \ell'} t'.forge \qquad \sum_{t \in \ell} t.fee = \sum_{t' \in \ell'} t'.fee$$

balance effects are equal:

$$\forall a, \mathcal{B}(a, \ell :: \lambda) - \mathcal{B}(a, \lambda) = \mathcal{B}(a, \ell' :: \lambda') - \mathcal{B}(a, \lambda')$$

where $\mathcal{B}(a, \lambda) \stackrel{\text{def}}{=} \mathcal{B}_{\text{Acc}}(a, \lambda) + \mathcal{B}_{\text{UTxO}}(a, \lambda)$.

Besides a theoretical interest, translations could be useful to approaches that may require interaction between cryptocurrencies that are based on different accounting models. This may include decentralized exchanges [15], cross-chain transactions [16, 17] and side-chains [18, 19].

5.1 From Account-Based to UTxO-Based

If t is an account-based transaction, let $\mathcal{T}^{\text{Acc}}_{\text{UTxO}}(t, \lambda')$ be a UTxO-*translation* of t for a target ledger λ' (assumed to be such that $\mathcal{B}_{\text{UTxO}}(s, \lambda') \geq t.value + t.fee - t.forge$, if t's sender is s) constructed as follows:

[22] Sec. 5.2 clarifies why the definition compares lists of transactions instead of single transactions.

1) Let $spent \stackrel{\text{def}}{=} t.value + t.fee - t.forge$, if $t.sender = \text{some}(s)$, and $spent \stackrel{\text{def}}{=} 0$, otherwise.

2) Construct a subset I of inputs from $\text{unspentOutputs}(\lambda')$ such that

$$\forall i \in I, \text{out}(i, \lambda').get.address = t.sender.get$$

$$\sum_{i \in I} \text{out}(i, \lambda').get.value \geq spent$$

3) If $t.receiver = \text{some}(r)$ for some address r, construct an output $\mathsf{Output}(r, t.value)$.

4) If $t.sender = \text{some}(s)$ for some address s, construct an output $\mathsf{Output}(s, change)$, where $change = spent - t.value$.

5) Let O be a list of outputs containing the outputs constructed in the previous two steps.

6) Let $\mathcal{T}^{\text{Acc}}_{\text{UTxO}}(t, \lambda') \stackrel{\text{def}}{=} \mathsf{UtxoTx}(I, O, t.forge, t.fee)$.

The following proposition can be proven easily by analyzing the fours cases depending on whether $t.receiver$ and $t.sender$ are defined.

Proposition 1. *$[\mathcal{T}^{\text{Acc}}_{\text{UTxO}}(t, \lambda')]$ on λ' is equivalent to $[t]$ on ledger λ, for any account-based transaction t and ledgers λ and λ' such that $\mathcal{B}_{\text{Acc}}(s, \lambda) \geq spent$ and $\mathcal{B}_{\text{UTxO}}(s, \lambda') \geq spent$, where $spent = t.value + t.fee - t.forge$, if $t.sender = \text{some}(s)$.*

In general, many translations are possible, because there may be many choices of I in step 2.

The translation described above assumes that each address in the source ledger λ is mapped to the same single address in the target ledger λ' (and it may even be the case that $\lambda' = \lambda$). It is straightforward to generalize this translation to the case where each address in the source ledger is mapped to a set of addresses in the target ledger. In step 2 it would be necessary to search for inputs that belong to any of the addresses in the set to which the sender of t is mapped. And in steps 3 and 4, it would be necessary to distribute the value and the change among the addresses in the sets to which the receiver and the sender of t are mapped, respectively, thereby creating a list of outputs instead of a single output in each step. The distribution leads to more degrees of freedom and hence to a greater number of possible translations.

5.2 From UTxO-Based to Account-Based

Translating in the other direction is more difficult.

Proposition 2. *For an arbitrary* UTxO*-based transaction t on a source ledger λ, there is generally no account-based transaction t' and target ledger λ' such that $[t]$ on λ is equivalent to $[t']$ on λ'.*

Proof. Consider the ledger λ of Example 1 and a new transaction

$$t_7 \stackrel{\text{def}}{=} \mathsf{UtxoTx}(\{\mathsf{Input}(t_6^{\#}, 0)\}, \\ [\mathsf{Output}(@1, \$499), \mathsf{Output}(@2, \$499)], \$0, \$1)$$

which has a single sender (@3) and two different receivers (@1 and @2). Any account-based transaction t' has at most one receiver. Therefore, t' could transfer money to @1 or @2, but not to both. □

Whenever a UTxO-based transaction has more than one sender or more than one receiver, it cannot be translated into a single equivalent account-based transaction. But it can be translated into a list of account-based transactions. One way to approach this problem is to see it as constraint satisfaction problem: if a UTxO-based transaction t in a ledger λ has n senders and m receivers, an account-based transaction $t'_{(s,r)}$ may be created for each pair of sender and receiver (s, r) in such a way that $\sum_{s,r} t'_{(s,r)}.fee = t.fee$, $\sum_{s,r} t'_{(s,r)}forge = t.forge$, $\sum_s t'_{(s,r)}.value = \sum_{o \in t.outputs, o.address=r} o.value$ and $\sum_r (t'_{(s,r)}.value + t'_{(s,r)}.fee - t'_{(s,r)}.forge) = \sum_{i \in t.inputs, \mathrm{out}(i,\lambda).get.address=s} \mathrm{out}(i, \lambda).get.value$. This approach is very general: all possible equivalent non-redundant[23] account-based transaction sequences can be generated as solutions to this constraint satisfaction problem. However, it can lead to $n \times m$ transactions.

Fortunately, the problem is loosely constrained, and it is in fact always possible to solve the problem with at most $n+m$ transactions and at least $max(n, m)$ transactions. To obtain a solution that is optimal on the number of transactions, the constraint satisfaction problem could be turned into an optimization problem, but solving the optimization problem would be computationally expensive. Instead, the proposal below describes a heuristic translation that is guaranteed to be within these bounds and to run in $\mathcal{O}(n + m)$, although it is not guaranteed to obtain the smallest possible number of transactions. During its execution, the translation optimistically expects that it will generate exactly $max(n^*, m^*)$ account-based transactions (where n^* and m^* are the numbers of senders and receivers that remain to be considered), and tries (but doesn't always perfectly succeed) to distribute the remaining fee and the remaining forged value evenly across the generated transactions.

If t is a valid UTxO-based transaction for a source ledger λ, let $\mathcal{T}^{\mathrm{UTxO}}_{\mathrm{Acc}}(t, \lambda)$ be an *account-translation* of t for the source ledger λ constructed as follows:

[23] A sequence of transactions is *redundant* iff there is more than one transaction from the same sender to the same receiver.

1) Initialize ss as the list of senders of t (i.e. those addresses a such that $\mathcal{B}_{\text{UTxO}}(a, t, \lambda) < 0$).

2) Initialize rs as the list of receivers of t (i.e. those addresses a such that $\mathcal{B}_{\text{UTxO}}(a, t, \lambda) > 0$).

3) Initialize $remForge \leftarrow t.forge$ (the remaining forge) and $remFee \leftarrow t.fee$ (the remaining fee).

4) For every $s \in ss$, initialize $remSpend(s) \leftarrow -\mathcal{B}_{\text{UTxO}}(s, t, \lambda)$, where $remSpend$ is a map storing the remaining amount each sender should spend.

5) For every $r \in rs$, initialize $remReceive(s) \leftarrow \mathcal{B}_{\text{UTxO}}(s, t, \lambda)$, where $remReceive$ is a map storing the remaining amount each receiver should receive.

6) Initialize $result \leftarrow []$

7) While ss is non-empty or rs is non-empty, execute one of the following cases:

A) If both ss and rs are non-empty, let s and r be their heads and st and rt their tails and do:

1. $expectNumTrans \leftarrow max(|ss|, |rs|)$[24]
2. $fee \leftarrow remFee/expectNumTrans$
3. $forge \leftarrow remForge/expectNumTrans$
4. $v \leftarrow remSpend(s) - fee + forge$
5. if $v = remReceive(r)$, do $ss \leftarrow st$ and $rs \leftarrow rt$;
   else if $v < remReceive(r)$, do $fee \leftarrow fee * (v/remReceive(r))$ and $forge \leftarrow forge * (v/remReceive(r))$ (reducing the fee and forge in order to heuristically account for the larger than expected number of transactions) and do $ss \leftarrow st$ (leaving rs unchanged); otherwise (if $v > remReceive$)), do $fee \leftarrow fee * (remReceive(r)/v)$, $forge \leftarrow forge * (remReceive(r)/v)$ (reducing the fee and forge in to heuristically account for the larger than expected number of transactions), $v \leftarrow remReceive(r)$ and $rs \leftarrow rt$ (leaving ss unchanged).
6. Add the following transaction to $result$: $\mathsf{AccTx}(\text{some}(s), \text{some}(r), v, forge, fee, 0)$

B) If ss is empty, let r be the head and rt the tail of rs and do:

1. $v \leftarrow remReceive(r)$; $fee \leftarrow remFee/|rs|$; $forge \leftarrow v + fee$
2. Add the following transaction to $result$: $\mathsf{AccTx}(\text{none}, \text{some}(r), v, forge, fee, 0)$
3. $rs \leftarrow rt$

C) If rs is empty, let s be the head and st the tail of ss and do:

[24] For a pessimistic variant of this optimistic translation, it suffices to modify step 7.A.1 so that $expectNumTrans \leftarrow |ss| + |rs|$.

1. $v \leftarrow 0$; $fee \leftarrow remFee/|ss|$;
   $forge \leftarrow fee - remSpend(s)$
2. Add the following transaction to *result*: $\mathsf{AccTx}(\mathrm{some}(s), \mathrm{none}, \$0, forge, fee, 0)$
3. $ss \leftarrow st$

and then update the state as follows:
$remForge \leftarrow remForge - forge$;
$remFee \leftarrow remFee - fee$;
$remSpend(s) \leftarrow remSpend(s) - (v + fee - forge)$
$remReceive(r) \leftarrow remReceive(s) - v$ (if s and r are defined, respectively).

8) Let $\mathcal{T}^{\text{UTxO}}_{\text{Acc}}(t, \lambda) \stackrel{\text{def}}{=} result$

All account-based transactions generated by the translation have nonces set to zero by default. Before using them on a target ledger, it may be necessary to increment their nonces. Alternatively, the target ledger could be passed as an extra argument to $\mathcal{T}^{\text{UTxO}}_{\text{Acc}}$, which would then inspect the target ledger and create transactions with nonces that guarantee uniqueness. The translation assumes the target ledger has a permissive monetary policy that does not restrict forging, and tries to distribute $t.forge$ evenly across all generated transactions. If this is not the case, the translation should be modified to adhere to the monetary policy. To simplify the presentation, it has been assumed that an address in the source ledger (of the UTxO-based transaction) corresponds to the same address in the target ledger (of the account-based transactions obtained by translation). It would be straightforward to generalize the translation to the case where an address in the source ledger is mapped to another address (or even to a set of addresses) in the target ledger. The following propositions hold.

Proposition 3. *$\mathcal{T}^{\text{UTxO}}_{\text{Acc}}(t, \lambda)$ on λ' is equivalent to $[t]$ on ledger λ, for any* UTxO*-based transaction t and ledgers λ and λ' such that $\mathcal{B}_{\text{Acc}}(s, \lambda') \geq \mathcal{B}_{\text{UTxO}}(s, t, \lambda)$ for every sender address in t.*

Proposition 4. *$\mathcal{T}^{\text{UTxO}}_{\text{Acc}}(t, \lambda)$ can be computed in $\mathcal{O}(n + m)$ and in $\Omega(\max(n, m))$, where n is the number of senders and m is the number of receivers of t, and the number of generated transactions g is such that $max(n, m) \leq g \leq n + m$.*

Proof. Note that, for every iteration of the while loop, one address is removed from either ss or rs, and sometimes (when $v = remReceive(r)$ in step 7.A.5) both ss and rs have an address removed. Therefore, if $n = |ss|$ and $m = |rs|$ at the beginning, there will be at least $\max(n, m)$ and at most $n + m$ iterations of the while loop. Since all steps in each iteration can be done in constant time w.r.t. n and m, the running-time is in $\mathcal{O}(n + m)$ and in $\Omega(\max(n, m))$. As every iteration generates a transaction, the total number of generated transactions is between $max(n, m)$ and $n + m$. □

6 Comparison and Unification

Despite the accounting equivalence implied by the translations, Proposition 2 and the translation of Sec. 5.2 show that it is generally necessary to create several account-based transactions to simulate the effect of a single UTxO-based transaction. This is important: while all the value transfers in a single UTxO-based transaction are processed atomically and simultaneously, the corresponding account-based transactions are not guaranteed to be so. There may be a wide time gap between their executions.

Moreover, whereas a single UTxO-based transaction with multiple inputs and outputs does not need to match senders with receivers, the corresponding sequence of account-based transaction must do so, arbitrarily, even though this information may be irrelevant. A consequence of this fact is that a single UTxO-based transaction with multiple inputs and outputs can be (linearly) more concise than the corresponding sequence of account-based transactions.

On the other hand, account-based transactions are clearly simpler and more concise for transfers between two addresses, because the change's value and destination do not need to be explicitly stated. Furthermore, value preservation is something that depends only on the transaction itself (and is even trivial when both sender and receiver are defined), whereas checking value preservation with UTxO requires searching the values of the unspent outputs referred by the inputs and hence does not depend only on the transaction itself.

Algebraically, the operation of composition of two UTxO-based transactions is easily definable (i.e. as long as the two transactions do not spend the same unspent output, there is another UTxO-based transaction that has the same effect as the two transactions combined), whereas account-based transactions can only be composed into a single new account-based transaction if they involve the same addresses.

There are two trivial solutions for the problem of lack of atomicity in the account-based style. Firstly, a new "wrapper" transaction type could be defined, to contain a list of account-based transactions that ought to be executed atomically. Secondly, the account-based transaction type AccTx could be modified to include an optional pointer to another account-based transaction, and the notion of ledger could be modified to only allow the simultaneous inclusion of such chains of account-based transactions. However, a better solution, that unifies ideas from both styles and combines their above mentioned advantages, is proposed below.

Definition 14. The *hybrid transaction* type is:

$$\begin{aligned}\mathsf{HybridTx} \stackrel{\mathsf{def}}{=} (&\mathit{inputs}\colon \mathsf{Map}[\mathsf{Address}, \mathsf{Value}],\\ &\mathit{outputs}\colon \mathsf{Map}[\mathsf{Address}, \mathsf{Value}],\\ &\mathit{forge}\colon \mathsf{Value}, \mathit{fee}\colon \mathsf{Value}, \mathit{nonce}\colon \mathsf{Int})\end{aligned}$$

requiring value preservation.

The inputs and outputs are simply maps from addresses to values that they spend or receive through the transaction. The transaction updates the account balances of these addresses as expected. To be valid, senders of a hybrid transaction must have a positive balance on the ledger and the transaction must be unique.

7 Chimeric Ledgers

Definition 1 already allows a ledger to contain transactions of various types. However, for UTxO- and account-based transactions to co-exist meaningfully, users should be able to convert their UTxO balances into account balances and vice versa. The conversion can be enabled through new transaction types.

Definition 15. The type for *deposit transactions* is:

$$\mathsf{DepTx} \stackrel{\mathsf{def}}{=} (\mathit{inputs}\colon \mathsf{Set}[\mathsf{Input}], \mathit{depositor}\colon \mathsf{Address}, \\ \mathit{forge}\colon \mathsf{Value}, \mathit{fee}\colon \mathsf{Value})$$

The type for *withdrawal transactions* is:

$$\mathsf{WithTx} \stackrel{\mathsf{def}}{=} (\mathit{withdrawer}\colon \mathsf{Address}, \mathit{outputs}\colon \mathsf{List}[\mathsf{Output}], \\ \mathit{forge}\colon \mathsf{Value}, \mathit{fee}\colon \mathsf{Value}, \mathit{nonce}\colon \mathsf{Int})$$

Through the analogy that sees UTxO-based transactions as cash-like transfers and account based transactions as bank account transfers, it can be said that a deposit transaction takes coins and notes of various denominations (i.e. the inputs) and deposits them in the depositor's account. The inputs are considered spent, and the account balance of the depositor's address is increased by an amount equal to what has been spent minus the fee and plus any forged amount. A withdrawal transaction does the opposite. It reduces the account balance of the withdrawer's address by an amount equal to the total value of the withdrawn coins and notes (i.e. the generated outputs) plus the fee and minus the forged amount.

In the Cardano [20] cryptocurrency, which is planned to have a multi-layered/multi-chain architecture where different chains may be based on different accounting models, it is expected that deposit and withdrawal transactions could ease the communication between the chains. A restricted form of deposit and withdrawal transactions is also being considered as a solution for implementing stake delegation rewards on a UTxO-based ledger in a way that does not create "dust" (i.e. an excessive amount of unspent outputs of insignificant value).

8 Discussion

Related Work: The definitions in Sec. 3 are inspired by similar definitions in [21], [22] and [23]. In contrast to those works, the framework described here has explicit fees and forged values and focuses only on accounting. The definitions in Sec. 4 aim to be a simplification and generalization, again focusing only on accounting, of similar definitions in [3] and [12]. The work in [24] presents a formal graph-based framework that can be instantiated to both UTxO and account-based transactions.

Hybrid transactions combine the best of both styles. They have the atomicity and algebraic elegance of UTxO-based transactions and the simplicity and locality of account-based transactions.

With a chimeric ledger, the dilemma of choosing between UTxO and account-based styles can be avoided. Instead, both transaction styles are offered, and users may choose whatever they prefer. If there is any difference between the two styles (e.g. storage space, processing time, privacy, suitability for smart contracts, ...), the free market would naturally price each transaction style differently.

The new deposit and withdrawal transaction types may be helpful to describe the transfer of assets between a main chain and a sidechain when each chain uses a different accounting style. Furthermore, they could also be useful in scenario where account-based smart contracts needed to be executed in a mostly UTxO-based ledger: a contract could receive funds through a deposit transaction and pay out funds through withdrawal transaction. Alternatively, the translations defined in Sec. 5 could be used to allow smart contracts to operate directly with UTxO under the surface, although with an account-based semantics in the programming language level.

As mentioned in the introduction, the minimalistic abstract and general models presented in this paper have already served as a basis for numerous further works involving specifications, implementations and tests. These models have facilitated on-boarding of new researchers, developers and formal methods specialists at Input Output HK (IOHK) and supported IOHK's high assurance formal software development methodology.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] V. Buterin, "Ethereum: A next generation smart contract & decentralized application platform," *Ethereum Project White Paper*, 2014. [Online]. Available: http://www.the-blockchain.com/docs/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf

[3] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, 2014. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[4] V. Buterin, "Thoughts on utxos," *Medium.com*, 2016. [Online]. Available: https://medium.com/@ConsenSys/thoughts-on-utxo-by-vitalik-buterin-2bb782c67e53

[5] M. Hearn, "Rationale for and tradeoffs in adopting a utxo-style model," *Corda Blog*, 2016. [Online]. Available: https://www.corda.net/2016/12/rationale-tradeoffs-adopting-utxo-style-model/

[6] P. Dai, "Why qtum chose utxo model and the benefits," *8BTC*, 2017. [Online]. Available: http://news.8btc.com/why-qtum-choose-utxo-model-and-the-benefits

[7] "Design rationale: Blockchain-level protocol: Account and not utxos," *Ethereum Wiki*, 2017. [Online]. Available: https://github.com/ethereum/wiki/wiki/Design-Rationale#accounts-and-not-utxos

[8] "Utxo model vs. account/balance model (forum thread)," *StackExchange: Bitcoin*, 2017. [Online]. Available: https://bitcoin.stackexchange.com/questions/49853/utxo-model-vs-account-balance-model

[9] E. de Vries, "Self organisation in coin selection." [Online]. Available: https://iohk.io/blog/self-organisation-in-coin-selection/

[10] P. Lamela Seijas and S. Thompson, "Marlowe: Financial contracts on blockchain," in *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, T. Margaria and B. Steffen, Eds. Cham: Springer International Publishing, 2018, pp. 356–375.

[11] J. Zahnentferner, "An abstract model of utxo-based cryptocurrencies with scripts," Cryptology ePrint Archive, Report 2018/469, 2018, https://eprint.iacr.org/2018/469.

[12] F. Vogesteller and V. Buterin, "ERC-20 token standard," *EIPs*, 2015. [Online]. Available: github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md

[13] S. Popov, "The tangle," *IOTA White Paper*, 2017. [Online]. Available: iota.org/IOTA_Whitepaper.pdf

[14] "Transaction," *Bitcoin Wiki*, July 2017. [Online]. Available: https://en.bitcoin.it/wiki/Transaction

[15] "Bisq: The peer-to-peer bitcoin exchange." [Online]. Available: https://github.com/bisq-network/docs/blob/master/exchange/whitepaper.adoc

[16] "Atomic cross-chain trading," *Bitcoin-Wiki*. [Online]. Available: en.bitcoin.it/wiki/Atomic_cross-chain_trading

[17] M. Herlihy, "Atomic cross-chain swaps," *CoRR*, vol. abs/1801.09515, 2018.

[18] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timoón, and P. Wuille, "Enabling blockchain innovations with pegged sidechains," 2014. [Online]. Available: https://www.blockstream.com/sidechains.pdf

[19] A. Kiayias, N. Lamprou, and A. Stouka, "Proofs of proofs of work with sublinear complexity," in *Financial Cryptography and Data Security - FC 2016 International Workshops*, 2016, pp. 61–78.

[20] "Cardano." [Online]. Available: whycardano.com/

[21] B. White, "A theory for lightweight cryptocurrency ledgers," 2015. [Online]. Available: https://github.com/ceilican/ledgertheory/blob/master/lightcrypto.pdf

[22] N. Atzei, M. Bartoletti, S. Lande, and R. Zunino, "A formal model of bitcoin transactions,"

Cryptology ePrint Archive, 2017. [Online]. Available: https://eprint.iacr.org/2017/1124.pdf

[23] A. Chepurnoy, "The blockchain and the scorex tutorial," Scorex Foundation, Tech. Rep., 2016. [Online]. Available: https://github.com/ScorexFoundation/ScorexTutorial/blob/master/scorex.pdf

[24] C. Cachin, A. D. Caro, P. Moreno-Sanchez, B. Tackmann, and M. Vukolić, "The transaction graph for modeling blockchain semantics," Cryptology ePrint Archive, Report 2017/1070, 2017, https://eprint.iacr.org/2017/1070.

-----BEGIN PGP MESSAGE-----

hQEMA3mYtjIcCbbOAQf/ci71Krwdlkd3ZzsoAkZdMKQYseQxI1YAxqEeshvnqcDJ
aCbKf5YXMdxEjumXW9EvTzjG8PnkLfviN9tpPWxmTujXOJcoiBkZ/CGSeO2msuhd
tC+W9xn5z0+Z7p4HZPRfiZ4ZmlBow77JFRMcqqf/OG/eZ+7kkigECZzs9bBE4b+
kG3EFl268D69JVu6q1eiyybF6U9D8OmDK4kRA3LuPyzqrE8sfdUDW8U0AuSUIsfY
YW8Jy6TdgikEDq6NzSS6X/jcjBq03XM3HJvA78l2XIooNSFKyoXWcJHTe9J+zXHe
krtnsI8Q/5US/C3FK/fX77k2gIKNgO8lYZQrQztBLNLA7AGYfZvDVc7WGliw8UJn
wPlJR6DUuOLvo1laej7OhYHvLqFQFoLjSOKzM77uaftXWpEIc+ETY4eD6mEthBnz
kLraN9ZRhu3u8yunuIpBrSIpX50EXKwOA/aUhTvuR8ZH9GEMVyL3tpqxNbUFbSTp
lXElcCeE7TkcORLpV8E/NzPFk2TlazRl8iHGepLR7vw1DT66H3xN4+qcI2FwtEs8
5rDKq6NDkhPKNC8IsYcdtmNzdy1CK2KbD3lBrEB3io1DvU2sD20a+NEsl8MeMknY
YW5GZ75y8W3YfL5/S0pVB6RqanpyZIHcq2P6wMYojpPhnPc500chOYGdLHeKwlJL
uEC3dEOMg3LBtkBWQTiNXF/NkO2WS4YmTZJROI7oZnlGuoIZYDkj3r/WaJHTZd7K
7A7BhxJSiDbj5uC2F/cpBduHV6mtdpByv87wXWF1JMNFWcVow0x99gg1j/aCCToE
fjO1/sqpsxoo4jNqr9kREqpVYghoayxNy0fQYfew/aaPOp382gfKEbDeEhNfMV+f
Gq3MDJfGuUmfFyDnGLvr/fZu8o83elb446mwtBj+
=cj7k
-----END PGP MESSAGE-----