Multi-Currency Ledgers

Joachim Zahnentferner
chimeric.ledgers@protonmail.com

Input Output HK
Hong Kong

18th of May 2018

Abstract. This paper extends an abstract formal model of UTxO-based and account-based transactions to allow the creation and use of multiple cryptocurrencies on a single ledger. The new model also includes a general framework to establish and enforce monetary policies for created currencies. In contrast to alternative approaches, all currencies in this model exist natively on the ledger and do not necessarily depend on a main currency. In comparison to non-native approaches based on scripts and smart contracts, native currencies allow smaller transactions that can be more efficiently processed and can be moved between chains through a sidechain approach.

1 Introduction

Starting with Bitcoin [1], blokchain technology has found in cryptocurrencies one of its most popular and prominent domain of applications. Today there are more than 1600 alternative cryptocurrencies listed in `worldcoinindex.com`.

To create an alternative cryptocurrency, one option is to start a new independent blockchain network based either on possibly modified pre-existing blockchain software (e.g. as done by Litecoin [2]) or on new blockchain software (e.g. as done by Ethereum [3,4]). Another option is to fork an existing blockchain (e.g. as done by BitcoinCash [5]). Although conceptually simple, these options have disadvantages. From the point of view of a prospective cryptocurrency issuer, they require expertise in the development and deployment of blockchain software. And, from the point of view of a user of the new cryptocurrency, the modified or new blockchain software and the network of nodes maintaining the independent blockchain may not be as robust and secure as those of pre-existing cryptocurrencies that have been running for a longer time and by a larger community of people. To tackle these disadvantages, there have been many approaches that aim to enable users to easily create new cryptocurrencies or, more generally, new asset types on top of an existing blockchain.

For instance, in the case of the Bitcoin blockchain, the original *colored coin* approach proposed to exploit Bitcoin's lack of perfect fungibility, in order to assign additional meaning to bitcoins. The idea can be almost equally well explained with physical dollar notes: a group of people could agree to paint some

dollar notes (with known serial numbers[1]) with a certain color and establish that whoever owns a painted dollar note also owns a certain number of units (a.k.a. tokens) of another asset type. As decided by the group, ownership of a token may represent ownership of a real world asset such as shares of a company, euros in a joint bank account, or participation in a joint fund. In contrast to physical dollar notes, in the case of bitcoin it is a set of transaction outputs that is assumed to be "colored"; and, whenever, these outputs are spent by a transaction, the new outputs generated by the transaction are assumed to be colored as well. Interestingly, the assigned meaning exists only by convention, implicitly, in the minds or private contracts of the group members. Neither the central bank (in the case of colored dollar notes) nor the blockchain network (in the case of colored bitcoin outputs) guarantees that owning a colored coin really entitles the owner to the assets that the coloring is supposed to represent. The owner must trust those who colored the coins. Also, with this approach one must transfer the main colored asset (e.g. a colored dollar note or a colored bitcoin output) in order to transfer the asset that it implicitly represents. Therefore, the implicit asset type is strongly dependent on the main currency (e.g. respectively, dollar or bitcoin). This could become a problem if the main currency's value increases so much that it becomes prohibitively expensive to transfer the implicit asset. Fortunately, however, the current Colored Coins project [6] has departed from the original colored coin idea and adopted a different approach that reduces the dependency of the new asset type on the main currency. This new approach, variations of which can also be seen in other projects such as Open Assets [7], Mastercoin/OmniProtocol [8] and Counterparty [9], actually does not color bitcoins, but instead uses bitcoin scripts to encode, as metadata, the quantities of tokens that are being transferred. It is possible, for instance, to have an output of 0 bitcoins that is marked, by a script in the transaction, as having a value of 10 tokens. Nevertheless, new cryptocurrencies created using the encoding are still not completely independent of bitcoin, because transfers must (and may only) pay fees in bitcoin. Furthermore, the encoding makes transactions bigger and more time-consuming to process. For example, comparing a "colored" transaction with an output of 10 tokens and a transaction with an output of 10 bitcoins, the latter stores the value natively on the output, whereas the former must additionally contain a script that (non-natively) encodes and stores the value outside the output.

In Ethereum, the creation of new *tokens* is supported by its smart contract language. To create a new token, it suffices to create a smart contract that keeps track of balances for every user and that provides a function to be called for transfers from one user to another. The ERC-20 Token Standard [10] establishes a standard interface for such token contracts. As in the case of colored coins, the created tokens are dependent on Ethereum's main currency, because fees for transactions that execute the smart contract's transfer function must be paid in

[1] In fact, it is unnecessary to actually paint the notes. It is sufficient to keep track of their serial numbers. Coloring coins is similar to the technique of marking bills, used by the police to trace money used by illegal activities.

ether. Although the ERC-20 Token Standard determines a common interface, it does not impose any standard implementation. Consequently, the implementation of an ERC-20 token's smart contract may contain bugs, which may even be maliciously intentional. Given that most tokens ought to behave in a similar way, having several similar smart contracts deployed to the block-chain leads to code duplication and is inefficient in terms of storage. Moreover, all these contracts mimic what Ethereum already does for its main currency: update account balances. But they do it more slowly and more expensively, because token transfers require the execution of a smart contract, whereas a native ether transfer transaction does not.

Recognizing the demand for the creation, transfer and exchange of new currencies or asset types in general, blockchain platforms such as NXT [11], Waves [12] and Ripple [13] decided to support this feature through special native transaction types, which are conceptually simpler, less bug-prone, more compact and more efficient to process, although less general, than the scripting and smart contract approaches adopted by Bitcoin and Ethereum. However, in all these blockchain systems, the created asset types are still dependent on the system's main currency, which is the only currency accepted in fee payments.

Another motivation for the use of native currencies instead of tokens based on smart contracts, is that native currencies could be moved from one chain to another in a blockchain system having multiple chains that can communicate with each other through a *sidechains* mechanism [**?**,14,15]. Tokens, on the other hand, must stay in the chain where they are created, because an instance of a smart contract in one chain cannot communicate with another instance in another chain through the sidechains mechanism. Native transactions are needed by sidechains mechanisms [15], and Ethereum-style smart contracts cannot create native transactions.

The abstract *multi-currency ledger* model described in this paper is inspired by the native transaction approaches taken by NXT, Waves and Ripple. However, in contrast to NXT, Waves and Ripple, the proposed model allows the creation of mutually independent cryptocurrencies. All currencies may have equal status and there is no need for a main currency. The proposed model is also simpler and, in some ways, more general than NXT, Waves and Ripple (as discussed in more detail in Section 7). The model presented here is an intentionally minimalistic extension of the *chimeric ledger* model [16] to the multi-currency case. As in [16] and [17], the abstract multi-currency ledger model presented here omits technical details of the underlying blockchain, because only the ledger of transactions (i.e. the data stored in the blockchain) is of interest. A *ledger* is assumed to be a list of transactions. A reference implementation in Scala is available in `https://github.com/input-output-hk/chimeric-ledgers-spec-scala`.

2 Assets and Currencies

An *asset* is anything that can be owned and that is deemed valuable by its owners. A *currency* is a particular type of asset that may function as store of

value, means of exchange and unit of account. For the purposes of this paper, the "may" is important: whether the asset type will actually have these functions is a sociological phenomenon and depends on people's choices. Blockchain technology cannot coerce people to use assets in a particular way, but it can provide ways to create types of assets that may be used as currencies. For an asset type to possibly function as a currency, the following properties seem necessary:

1. the assets must be sufficiently fungible (otherwise they cannot serve as unit of account);
2. the assets must be fully under the control of the owner, and not of the issuer or some other centralized party (otherwise they cannot serve as a reliable means of exchange);
3. the asset type must have a reasonably predictable monetary policy (otherwise they cannot serve as a trusted store of value).

Note that assets such as cars or real state do not satisfy the first property. Various financial assets such as bonds, shares and even money just sitting in a bank account do not necessarily satisfy the second property. Moreover, although any external asset can in principle be represented on a ledger, the question of whether ownership of the representation really entails ownership of the asset that it ought to represent is a matter that depends on trust on the issuer of the representation. All such types of assets are beyond this paper's scope, because this paper focuses solely on currencies.

3 Notations

A record data type with fields φ_1, ..., φ_n of types T_1, ..., 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 length of a list λ is denoted $|\lambda|$. A map μ of type $\mathsf{Map}[A, B]$ is a collection of key-value pairs where each key occurs at most once, keys are of type A and values are of type B. The value corresponding to a key k in a map μ is denoted $\mu(k)$. A set s of type $\mathsf{Set}[T]$ is a collection of elements of type T such that every element may occur at most once in s. An option o of type $\mathsf{Option}[T]$ is a collection that is either empty (denoted by none) or contains a single element e (denoted by some(e)). The standard equality symbol ($=$) is used to state that two values are equal. The definitional equality symbol ($\stackrel{\mathsf{def}}{=}$) 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 ...".

4 Multi-Currency Values

In the chimeric ledger model of [16] and [17], the type Value was assumed to be the type of integers, and values of this type were used to denote the (often non-negative) quantity of an implicitly known single currency that was transferred, forged, paid as fee or generally considered as the balance of an address in a transaction or in a ledger. In a multi-currency setting, the least that is needed is a generalization of the notion of Value so that it contains not only a quantity but also the name of the currency. This could be achieved by turning the type Value into, for instance, a pair of an integer quantity and a string for the currency's name. However, such a new Value would not have the same algebraic properties that it used to have. Integers form a commutative group with respect to the operation of addition. On the other hand, integers paired with currency names do not. For instance, "adding" $(10, \text{"euro"})$ and $(5, \text{"dollar"})$ does not result in something that could be meaningfully represented as a single pair of an integer quantity and a single currency name.

A quick inspection of the definitions of balance and the validity conditions in [16] and [17] reveals that it is important to be able to add, subtract and compare values. Therefore, it is more sensible to generalize Value in a such way that Value is, as it used to be before, a commutative group with a (partial) order. This can be done by turning Value into a map from currency names (strings) to quantities (integers) and defining addition and inverse operators and the order relation appropriately.

Definition 1. *The type for* multi-currency values *is defined as:*

$$\mathsf{Value} \stackrel{\mathsf{def}}{=} \mathsf{Map}[\mathsf{Currency}, \mathsf{Quantity}]$$

where:

- Currency *is assumed to be a type alias for* String *or any other reasonable currency identifier.*
- Quantity *is assumed to be a type alias for integers.*
- *for any* $v : \mathsf{Value}$ *and any* $c : \mathsf{Currency}$*, the quantity of* c *in* v *is denoted* $v(c)$ *with default* 0 *when the map* v *is not defined for* c*.*

and for any $v, v' : \mathsf{Value}$ *and* $c : \mathsf{Currency}$*:*

$$(v + v')(c) \stackrel{\mathsf{def}}{=} v(c) + v'(c)$$

$$\emptyset(c) \stackrel{\mathsf{def}}{=} 0$$

$$(-v)(c) \stackrel{\mathsf{def}}{=} -v(c)$$

$$v - v' \stackrel{\mathsf{def}}{=} v + (-v')$$

$$v \geq v' \stackrel{\mathsf{def}}{=} \forall c, v(c) \geq v'(c)$$

The following two propositions are easy to show.

Proposition 1. Value *is a commutative group with* $+$ *its distinguished group operation,* $\emptyset$ *its identity element, and unary* $-$ *the operation for inverse elements.*

Proposition 2. $\geq$ *is a partial order relation on* Value*.*

It is also easy to see that $\geq$ is not total.

Example 1. The values $v_1 \doteq \mathsf{Value}(\text{“}euro'' \to 10)$ and $v_2 \doteq \mathsf{Value}(\text{“}dollar'' \to 5)$ are incomparable. Neither $v_1 \geq v_2$ nor $v_2 \geq v_1$. Therefore, $\geq$ is not total.

The order's non-totality is not a problem. The order is only used in validity conditions that check whether a value is greater than or equal to $\emptyset$ (i.e. the *empty value*) or a balance b. In such cases, when the value is incomparable to $\emptyset$ or b, it is not greater than or equal to $\emptyset$ or b, and the condition fails as it should.

Given that values are often required to be non-negative (i.e. greater than or equal to $\emptyset$), it may be tempting to define Value not as a commutative group (with inverses and subtraction) but rather as a commutative monoid with a monus operation and without negative values. However, this would be inconvenient for various reasons. Firstly, negative values actually do occur, as balances of addresses in transactions, and it would be unnatural to cope with them in a monoid that does not have negative values. Secondly, defining subtraction is simpler than defining monus, since monus requires special treatment of the case when the subtrahend is greater than the minuend. Thirdly, the use of monus may suggest that a transaction that tries to transfer more than what the sender has should silently succeed by transferring only the value that the sender has, but this is probably not desirable. And fourthly, while negative ledger balances are a problem in a public ledger with pseudonymous identities because of sybil attacks, they could be useful to record debt in a possibly private ledger where identities are known. An abstract model where values are a group could easily support this use case just by changing the validity conditions to allow a negative balance greater than or equal to a debt limit, as credit cards allow, for example. With a monoid, on the other hand, handling debt would not be as easy.

The data structure for Value described above is very similar to multisets or bags of currencies, essentially differing only in that it allows negative quantities.

The main benefit of generalizing Value to the multi-currency case in a way that retains its algebraic operations and properties is that all the transaction types defined in [16] and [17] (e.g. UTxO-based transfers, account-based transfers, hybrid transfers, deposits and withdrawals) generalize trivially to the multi-currency case. The only change needed is the replacement of 0 by $\emptyset$ in the validity conditions. For this reason, these transaction types are not redefined here.

5 Monetary Policies

In the chimeric ledger model [16], the monetary policy that constrains the forging of the single currency was intentionally left unspecified. It was assumed that such a policy would be implemented as additional validity conditions on the *forge*

values inside transactions. However, in the multi-currency case, each currency may have a different monetary policy and it is desirable to allow the creator of a currency to specify the policy. This can be done with the following new transaction type.

Definition 2. *The datatype for* currency creation transactions *is defined as:*

$$\begin{aligned}\mathsf{CurrencyTx} \stackrel{\mathsf{def}}{=} (&\mathit{currency}\colon \mathsf{Currency},\\ &\mathit{policy}\colon \mathsf{Policy},\\ &\mathit{inputs}\colon \mathsf{Set}[\mathsf{Input}],\\ &\mathit{creator}\colon \mathsf{Option}[\mathsf{Address}],\\ &\mathit{fee}\colon \mathsf{Value})\end{aligned}$$

where currency is the name of the new currency, fee is the currency creation fee, inputs are a (possibly empty) set of inputs to pay the fee, creator is an optional account address[2] *for fee payment as well and policy is a monetary policy (as discussed below).*

For the sake of generality, a monetary policy p of type Policy is assumed to be a script written in a *monetary policy specification language*. The script denotes a function $[\![p]\!]\colon \mathsf{State} \to \mathsf{Option}[\mathsf{Certificate}] \to \mathbb{B}$, where $\mathbb{B}$ is the type of booleans, State is the type for relevant state information about the ledger and the current transaction and Certificate is the type for a certificate (e.g. a signature) to be provided by whoever wants to forge a currency. The function denoted by the script should return true if and only if the forging is authorized. As such, the monetary policy and the certificate are analogous to, respectively, validator and redeemer scripts, but they authorize the forging of a value instead of the expenditure of an output.

The state allows the specification of monetary policies where the authorization of forging depends on the available supply, the current block number, the shape or content of the transaction, or the position of the transaction in a block. Bitcoin's deflationary policy, which allows forging of a fixed amount only at the first transaction of a block, is an example of monetary policy that could be implemented by making use of the state.

The optional certificate allows the specification of monetary policies that allow someone (e.g. the currency creator or a central bank) to forge more of the currency. In order to allow the forger to provide a certificate, it is necessary to change the type of the *forge* field of all transaction types that can forge currency. Whereas in [16] and [17] the field *forge* was simply of type Value, now it must be of a new datatype Forge, as defined below.

Definition 3. *The datatype for* forging *is defined as:*

$$\mathsf{Forge} \stackrel{\mathsf{def}}{=} (\mathit{value}\colon \mathsf{Value}, \mathit{certificates}\colon \mathsf{Map}[\mathsf{Currency}, \mathsf{Certificate}])$$

[2] Here the ledger is assumed to be chimeric, containing both UTxO-based and account-based transactions. In a pure account-based ledger, the *inputs* field is unnecessary. In a pure UTxO-based ledger, the *creator* field is unnecessary.

where value is the value being forged and certificates maps each currency being forged to a forging authorization certificate.

It is now possible to define a new validity condition for any transaction that forges currencies.

Definition 4. *Any transaction t that has a forge field must satisfy the following condition, in addition to its own validity conditions, to be considered* valid *for a ledger λ:*

forging is authorized by the policies:

$$\forall (c \to v) \in t.forge.value,$$
$$[\![t'.policy]\!](s, t.forge.certificates(c)) = true$$

where s is the current state, which contains relevant information about λ and t, and t' is the currency creation transaction in λ such that $t'.currency = c$.

Finally, in order to be valid, currency creation transactions must satisfy conditions related to the payment of the fee and expenditure of outputs as well as a condition ensuring that the created currency is new in the ledger.

Definition 5. *A currency creation transaction t is* valid *for a ledger λ iff the following conditions hold:*

creator has enough money:

$$\forall a \in t.creator, \mathcal{B}_{\text{Acc}}(a, \lambda) \geq t.fee - \sum_{i \in t.inputs} value(i, \lambda).get$$

fee is non-negative:

$$t.fee \geq \emptyset$$

no surplus from inputs:

$$t.fee \geq \sum_{i \in t.inputs} value(i, \lambda).get$$

all inputs refer to unspent outputs:

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

no output is double spent:

$$|t.inputs| = |t.ins.map(i \Rightarrow i.outputRef)|$$

where $i \Rightarrow i.outputRef$ is the anonymous function that takes an input i and returns its output reference.

all inputs validate:

$$\forall i \in t.inputs, [\![i.validator]\!](s, [\![i.redeemer]\!](s)) = true$$

where s is the current state, which may depend on, and contain information about, λ and t.

validator scripts hash to their output addresses:

$$\forall i \in t.inputs, i.validator^{\#} = \text{out}(i, \lambda).get.address$$

created currency is new:

$$\neg \exists t' \in \lambda, t' : \mathsf{CurrencyTx} \wedge t'.currency = t.currency$$

6 Smart Contracts interacting with Currencies

The *native* approach adopted here allows users to create and use currencies in a way that is simpler, computationally cheaper and less bug-prone than through an approach based on smart contracts. However, not all desirable asset types are currencies; and smart contracts remain essential for the implementation of non-currency asset types. For instance, *shares* that pay dividends to their owners are an interesting example of non-currency asset type that could be implemented as an ERC-20 token contract that interacts with a currency. The tokens would represent the shares, and the interaction with a currency would be needed for paying dividends in the currency.

The following functions, written in Solidity [18], illustrate how an ERC-20 token contract[3] may interact with Ethereum's single currency (known as *ether*).

```
function () public payable {
   _supply = _supply.add(msg.value);
   balances[msg.sender] = balances[msg.sender].add(msg.value);
}

function withdraw(uint amount) public returns (bool success) {
   balances[msg.sender] = balances[msg.sender].sub(amount);
   _supply = _supply.sub(amount);
   msg.sender.transfer(amount);
   return true;
}
```

The first function allows a user to deposit an amount of ethers and get an equal amount of freshly created tokens added to the supply. Because the function is unnamed, it is the function that is called by default in the smart contract and it is known as a callback function. As it is declared as *payable*, a user calling this function can include a non-zero *value* of ethers in the transaction where the function is called. This value can be accessed by the function through the non-local built-in variable `msg.value`. The address of the user who called the function can be accessed through the non-local built-in variable `msg.sender`.

The second function allows a user to withdraw an amount of ethers, destroying an equal amount of tokens that he or she owns. The contract transfers its ethers to the caller (`msg.sender`) by using its built-in `transfer` function.

Note that a one-to-one correspondence between the currency and the token is not necessary. It is used here just for the sake of simplicity, as it suffices to illustrate the *payable* and *transfer* mechanisms through which a Solidity contract may interact with a native currency.

In the multi-currency case, as described in Section 4, the value becomes a map from currencies to quantities. Therefore, in Solidity, it would be natural to generalize `msg.value` to a `mapping(string => uint)` and let the argument of built-in `msg.sender.transfer` function take an argument of type

[3] Appendix A shows the full Solidity code for tokens pegged to and backed by ethers.

`mapping(string => uint)` as well, instead of `uint`. The functions shown below do the same as the deposit and withdrawal functions shown above, but using such a hypothethical generalized multi-currency Solidity language.

```
function () public payable {
    _supply = _supply.add(msg.value["ETH"]);
    balances[msg.sender] = balances[msg.sender].add(msg.value["ETH"]);
}

mapping(string => uint) toBeWithdrawn;
function withdraw(uint amount) public returns (bool success) {
   balances[msg.sender] = balances[msg.sender].sub(amount);
   _supply = _supply.sub(amount);
   toBeWithdrawn["ETH"] = amount;
   msg.sender.transfer(toBeWithdrawn);
   return true;
}
```

Note that `toBeWithdrawn` has to be defined outside the `withdraw` function because Solidity does not allow the declaration of local variables of type `mapping` inside functions. Despite the stateful non-local mapping, this `withdraw` function is immune to the reentrancy bug because it follows the *Checks-Effects-Interactions Pattern* [19].

7 Discussion

The *multi-currency ledger* model defined here allows the creation of mutually independent native currencies with pre-established monetary policies. The design goals of simplicity and minimalism have been achieved through:

1. the generalization of the notion of Value from a simple integer amount to a map from currencies to integer amounts. The preservation of the algebraic structure of commutative group meant that the pre-existing notions of account-based and UTxO-based transactions defined in the *chimeric ledger* model [16] did not need to be modified.
2. a transaction type for creating currency and establishing its monetary policy.

Three prominent examples of blockchain systems enabling native asset type creation are Waves, Ripple and NXT. It is worth exploring in more detail how they relate to the *multi-currency ledger* model proposed here.

Waves's white paper [12] states that "Waves will allow payment of network transaction fees in custom tokens (assets). Along with the transaction in question, an order to exchange the asset into the main network token is sent to the decentralized exchange, and the transaction can be included in the next block only after that order has been executed". Indeed, some (but not all) transaction types in Waves's code allow for fees to be paid in a different asset. However, the currency must be previously whitelisted manually in the settings file of every

node. Currently, only Waves's main currency is whitelisted (cf. the fees section of [20]) and fees must be paid in the main currency when using online wallets such as `waveswallet.io` and `beta.wavesplatform.com`. Furthermore, by inspecting the code, it seems that the idea that an exchange order will be automatically created and sent to the decentralized exchange remains a non-implemented feature at the moment. In any case, since fees paid in created assets would first have to be exchanged for the main currency, created assets would still not enjoy the same status as the main currency. The secondary status of created assets also reflects itself in the fact that Waves's code has different transaction types for transfers of the main currency and for transfers of created assets, even though they do essentially the same operation but with different asset types. When creating an asset in Waves, there are only two possible monetary policies: either fixed supply or a non-fixed supply where the original issuer has the authority to re-issue at any moment. Both of these policies are expressible within the general monetary policy framework presented here.

In Ripple, a user can create a currency by making a payment transaction from his own account to some account and specifying a new currency code and itself as the issuer. The issuer always has authority to forge more of that currency at any time, and this is the only possible monetary policy enforced by the system. Of course, an issuer could promise the users of her currency to follow a predefined monetary policy (e.g. to keep the total supply fixed), but the users must trust the issuer and at any moment the issuer could break her promise. As the documentation [21] states, "The [Ripple] Ledger has two kinds of money: XRP, and issued currencies. In the XRP Ledger, both types have high precision, although their formats are different.". Fees must necessarily be paid in XRP. Optionally, a currency issuer may also require the payment of an additional fee whenever users transfer his currency, necessarily defined as a percentage of the amount transferred. Ripple also has an interesting feature that allows currency issuers to *freeze* the use of their currencies by some or all users, which is arguably "useful to meet regulatory requirements" [22]. From the point of view of users, however, this is an undesirable risk. Moreover, whereas it may be reasonable to be able to freeze some asset types, freezeable asset types might not deserve to be called currencies, since freezing undermines the ability of an asset type to function as a reliable store of value and medium of exchange.

NXT allows the creation of assets and currencies, where assets have a fixed supply determined at the moment of creation (and, interestingly, may pay dividends to those who hold them) whereas currencies may have more sophisticated monetary policies. However, the monetary policies are still somewhat limited. The issuer may either choose a fixed supply or declare the currency *mintable* and then choose an initial supply, a maximum supply and a hash algorithm through which the current supply may be gradually increased up to the maximum supply by users who solve a cryptographic puzzle (finding a nonce such that its hash is less than a given target hash). Interestingly, when creating a currency, the issuer may also choose whether it is *exchangeable*, *controllable*, *reservable*, *claimable*, and *non-shuffable*, as explained in [23]. As fees must always be paid in NXT's

main currency, created asset types and currencies always have a secondary status, subordinate to the main currency. Moreover, currencies that are reservable and claimable are even more tied to the main currency.

It is also interesting to note that NXT, Waves and Ripple are all account-based ledgers, whereas the multi-currency ledger model presented here is agnostic to the ledger style and may allow both account-based and UTxO-based transactions. Furthermore, whereas payment or transfer transactions in NXT, Waves and Ripple always transfer a single asset or currency from one account to another, the generalized notion of value presented here allows the atomic transfer of multiple assets and currencies in a single account-based transaction.

Note that the comparison with NXT, Waves and Ripple has focused only on currency creation and monetary policies, but these blockchain systems have many additional features that have not been discussed here and that are not supported by the multi-currency ledger. For instance, all these systems have native transaction types for exchanging one asset or currency for another. These transaction types resemble buy and sell orders that are typically seen in brokers for stock exchange, with the crucial difference that the orders and, consequently, are recorded on the blockchain and the whole system is maintained in a decentralized manner. There are mainly two reasons why exchange transactions have not been considered here. Firstly, with multi-currency UTxO-based transactions that allow an anyone-can-pay signature scheme [24], it would already be possible for two parties to atomically exchange currencies. This not necessarily be convenient, though since would first have to share the unsigned transaction among themselves off-chain to sign it together, before submitting it to the blockchain. Secondly, and more importantly, while it would be straightforward to extend the multi-currency ledger model with exchange transactions and have buy and sell offers directly on the ledger, it is not clear that this creates a fair marketplace. Blockchain systems have a "mempool" of transactions that have not been added to the blockchain yet, and depending on the block creation period, it may take a long time for a transaction to be sufficiently confirmed on the blockchain. A user running a node of the blockchain system would be able to see new buy and sell orders as soon as they arrive to her node's mempool, but a user who is interacting with the exchange through, for instance, a web interface that only shows buy and sell orders that have already been included and confirmed on the chain would only see the new orders much later. Clearly the former user has an advantage and might be able to use it to manipulate the market. In the case of an ordinary centralized broker, there are regulations that prevent him from manipulating the market. But in the case of a blockchain-based exchange, the decentralized and anonymous nature of the system makes it harder, if not impossible, to enforce similar regulations. A fair decentralized blockchain-based exchange is, after all, not as trivial as it may seem at first, and is best left for future work.

From a theoretical perspective, it is elegant to use a model where all currencies have an equal status, as proposed here. From a technical perspective, this facilitates code reuse, as the same transaction types may be used for any

currency and there is no need to distinguish transactions manipulating created currencies from those manipulating the main currency. However, from a business point of view, it may be desirable to have a main currency with a special status, since the revenue model of a blockchain's system for its stakeholders may depend on the existence of the main currency. For instance, by requiring that fees must be paid in the main currency, Waves, NXT and Ripple ensure that there is always demand for the main currency that they control. The multi-currency model presented here is agnostic to these business considerations. It does not prevent giving a currency a special status if this is desirable. This would only require additional validity conditions on the transaction types (e.g. requiring that the fee is paid in the main currency).

Although the main property of interest in the discussion about the status of currencies has been whether a currency may be used to pay fees, other properties may be relevant as well. For instance, in a blockchain system that uses proof-of-stake consensus, the status of currencies may differ with respect to whether they count as stake. A valuable currency that is not counted as stake may have crucial implications for the security and the economic dynamics of a proof-of-stake blockchain: large holders of stake-contributing currencies that do not hold the valuable non-stake-contributing currency may be tempted to tamper with the history of the ledger if their stake-contributing currencies are not as valuable; and large holders of the non-stake-contributing currency may feel forced to purchase and hold large quantities of the stake-contributing currency, in order to ensure the security of their non-stake-contributing holdings.

Although the proposed multi-currency ledger model strives to allow currencies to be as mutually independent as they can be, there is an inherent and somewhat inescapable interdependency between currencies located on the same blockchain. If an attacker has an interest in performing an attack (e.g. a denial of service attack or a 51% attack) on the system because of a currency C_1, then a currency C_2 on the same chain may also suffer consequences in case the attack is successful. The use of sidechains [**?**, 14, 15] could help in mitigating this issue by enabling the two currencies to be kept in separate but communicating chains.

Acknowledgments: This paper has benefited from discussions with Duncan Coutts, Manuel Chakravarty, Charles Hoskinson, Philipp Kant, Pablo Lamela Seijas, Gerard Moroney, Chad Nester, Arnaud Spiwack and Philip Wadler.

References

1. S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.
2. "Litecoin." [Online]. Available: https://litecoin.com/
3. 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
4. G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, 2014. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

5. “Bitcoincash.” [Online]. Available: https://www.bitcoincash.org
6. “Colored coins.” [Online]. Available: https://coloredcoins.org
7. “Open assets.” [Online]. Available: https://github.com/OpenAssets
8. “Mastercoin omniprotocol.” [Online]. Available: https://github.com/OmniLayer/spec
9. “Counterparty.” [Online]. Available: https://counterparty.io
10. F. Vogesteller and V. Buterin, “ERC-20 token standard,” *EIPs*, 2015. [Online]. Available: github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md
11. “Nxt website.” [Online]. Available: https://nxtplatform.org
12. “Waves white paper.” [Online]. Available: https://blog.wavesplatform.com/waves-whitepaper-164dd6ca6a23
13. “Ripple transactions documentation.” [Online]. Available: https://ripple.com/build/transactions/
14. 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
15. P. Gazi, A. Kiayias, and D. Zindros, “Sidechains for proof-of-stake blockchain,” in *Submitted*, 2018, pp. 1 – 15.
16. J. Zahnentferner, “Chimeric ledgers: Translating and unifying utxo-based and account-based cryptocurrencies,” Cryptology ePrint Archive, Report 2018/262, 2018, https://eprint.iacr.org/2018/262.
17. ——, “An abstract model of utxo-based cryptocurrencies with scripts,” 2018.
18. “Solidity documentation,” 2016–2017, revision 124ca40d. [Online]. Available: http://solidity.readthedocs.io/en/v0.4.23/
19. “Solidity documentation: Security considerations,” 2016–2017, revision 124ca40d. [Online]. Available: http://solidity.readthedocs.io/en/v0.4.23/security-considerations.html
20. “Waves code: “application.conf”.” [Online]. Available: https://github.com/wavesplatform/Waves/blob/a2e20be0d22a8a2475b5af123f707c8730c6a764/src/main/resources/application.conf
21. “Ripple currency format documentation.” [Online]. Available: https://ripple.com/build/currency-format/
22. “Ripple freeze feature documentation.” [Online]. Available: https://ripple.com/build/freeze/
23. “Nxt wiki: Monetary system.” [Online]. Available: https://nxtwiki.org/wiki/Monetary_System
24. “OP_CHECKSIG,” Bitcoin Wiki, January 2018. [Online]. Available: https://en.bitcoin.it/wiki/OP_CHECKSIG

A Full ERC-20 Token Contract Example

This is the full Solidity code of an ERC-20 compliant smart contract for tokens pegged to and backed by ether, as discussed in Section 6. This is intended to be a minimal example illustrating how ERC-20 Tokens may interact with Ethereum’s native currency.

```
pragma solidity ^0.4.18;
```

```
// ----------------------------------------------------------------------
// This is an example of an ERC-20 compliant token contract
// where tokens have a 1-to-1 correspondence to ethers.
//
// This contract behaves like a simple bank.
// It accepts deposits and withdrawals of a "real" currency
// (ether) from its customers/users and keeps track of
// "virtual" tokens representing ethers in their balances.
//
// The deposited ethers are stored in the contract's address,
// and the created tokens are credited to the sender's balance
// within the contract. Users can convert their tokens back to
// ethers by calling the "withdraw" function.
//
// Symbol        : ETHT
// Name          : EtherToken
// Initial supply: 0
// Decimals      : 18  (as many decimals as ethers)
//
// (c) IOHK - Input Output Hong Kong 2018. The MIT License.
//
// This contract is based on and has borrowed code from the sample
// fixed supply ERC-20 token contract by:
// BokkyPooBah / Bok Consulting Pty Ltd 2017. The MIT Licence.
// from: https://theethereum.wiki/w/index.php/ERC20_Token_Standard
// ----------------------------------------------------------------------

// ----------------------------------------------------------------------
// ERC Token Standard Interface
// https://github.com/ethereum/EIPs/blob/master/EIPS/
// eip-20-token-standard.md
// ----------------------------------------------------------------------
contract ERC20Interface {
    function totalSupply() public constant returns (uint);
    function balanceOf(address tokenOwner)
      public constant returns (uint balance);
    function allowance(address tokenOwner, address spender)
      public constant returns (uint remaining);
    function transfer(address to, uint tokens)
      public returns (bool success);
    function approve(address spender, uint tokens)
      public returns (bool success);
    function transferFrom(address from, address to, uint tokens)
      public returns (bool success);

    event Transfer(address indexed from,
      address indexed to, uint tokens);
    event Approval(address indexed tokenOwner,
      address indexed spender, uint tokens);
}
```

```
library SafeMath {
    function add(uint a, uint b) internal pure returns (uint c) {
        c = a + b; require(c >= a);
    }
    function sub(uint a, uint b) internal pure returns (uint c) {
        require(b <= a); c = a - b;
    }
    function mul(uint a, uint b) internal pure returns (uint c) {
        c = a * b; require(a == 0 || c / a == b);
    }
    function div(uint a, uint b) internal pure returns (uint c) {
        require(b > 0); c = a / b;
    }
}

contract EtherToken is ERC20Interface {
   using SafeMath for uint;

   string public symbol;
   string public name;
   uint8 public decimals;
   uint public _supply;

   mapping(address => uint) balances;
   mapping(address => mapping(address => uint)) allowed;

   consructor() public {
       symbol = "ETHT";
       name = "EtherToken";
       decimals = 18;
       _supply = 0; // initial supply of 0
   }

   function totalSupply() public constant returns (uint) {
       return _supply;
   }

   function balanceOf(address tokenOwner)
     public constant returns (uint balance) {
       return balances[tokenOwner];
   }

   function allowance(address tokenOwner, address spender)
     public constant returns (uint remaining) {
       return allowed[tokenOwner][spender];
   }

   // ------------------------------------------------------------------
   // Transfer ‘amount‘ from caller to ‘to‘
```

```
// - caller's balance must be sufficient
// ------------------------------------------------------------
function transfer(address to, uint amount)
  public returns (bool success) {
    balances[msg.sender] = balances[msg.sender].sub(amount);
    balances[to] = balances[to].add(amount);
    emit Transfer(msg.sender, to, amount);
    return true;
}

// ------------------------------------------------------------
// Approve `spender` to transferFrom(...)
// up to `amount` from caller
// ------------------------------------------------------------
function approve(address spender, uint amount)
  public returns (bool success) {
    allowed[msg.sender][spender] = amount;
    emit Approval(msg.sender, spender, amount);
    return true;
}

// ------------------------------------------------------------
// Transfer `tokens` from `from` to `to`
// - `from`'s balance must be sufficient
// - caller's allowance must be sufficient
// ------------------------------------------------------------
function transferFrom(address from, address to, uint tokens)
  public returns (bool success) {
    balances[from] = balances[from].sub(tokens);
    allowed[from][msg.sender] =
      allowed[from][msg.sender].sub(tokens);
    balances[to] = balances[to].add(tokens);
    emit Transfer(from, to, tokens);
    return true;
}

// ------------------------------------------------------------
// Receive ethers, create tokens,
// add created tokens to caller's balance
// ------------------------------------------------------------
function () public payable {
    _supply = _supply.add(msg.value);
    balances[msg.sender] = balances[msg.sender].add(msg.value);
}

// ------------------------------------------------------------
// Subtract amount of tokens from caller,
// destroy that amount of tokens,
// and send the same amount of ethers to the caller
// - caller's balance must be sufficient
```

```
    // ------------------------------------------------------------------
    function withdraw(uint amount) public returns (bool success) {
        balances[msg.sender] = balances[msg.sender].sub(amount);
        _supply = _supply.sub(amount);
        msg.sender.transfer(amount);
        return true;
    }
}
```

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

hQEMA3mYtjIcCbbOAQf/ci71Krwdlkd3ZzsoAkZdMKQYseQxI1YAxqEeshvnqcDJ
aCbKf5YXMdxEjumXW9EvTzjG8PnkLfviN9tpPWxmTujXOJcoiBkZ/CGSeO2msuhd
tC+W9xn5z0+Z7p4HZPRFfiZ4ZmlBow77JFRMcqqf/OG/eZ+7kkigECZzs9bBE4b+
kG3EF1268D69JVu6q1eiyybF6U9D8OmDK4kRA3LuPyzqrE8sfdUDW8U0AuSUIsfY
YW8Jy6TdgikEDq6NzSS6X/jcjBqO3XM3HJvA7812XIooNSFKyoXWcJHTe9J+zXHe
krtnsI8Q/5US/C3FK/fX77k2gIKNgO8lYZQrQztBLNLA7AGYfZvDVc7WGliw8UJn
wPlJR6DUuOLvo1laej7OhYHvLqFQFoLjSOKzM77uaftXWpEIc+ETY4eD6mEthBnz
kLraN9ZRhu3u8yunuIpBrSIpX5OEXKwOA/aUhTvuR8ZH9GEMVyL3tpqxNbUFbSTp
lXElcCeE7TkcORLpV8E/NzPFk2TlazRl8iHGepLR7vw1DT66H3xN4+qcI2FwtEs8
5rDKq6NDkhPKNC8IsYcdtmNzdy1CK2KbD3lBrEB3io1DvU2sD2Oa+NEsl8MeMknY
YW5GZ75y8W3YfL5/SOpVB6RqanpyZIHcq2P6wMYojpPhnPc5OOchOYGdLHeKwlJL
uEC3dEOMg3LBtkBWQTiNXF/Nk02WS4YmTZJROI7oZnlGuoIZYDkj3r/WaJHTZd7K
7A7BhxJSiDbj5uC2F/cpBduHV6mtdpByv87wXWF1JMNFWcVow0x99gg1j/aCCToE
fjO1/sqpsxoo4jNqr9kREqpVYghoayxNyOfQYfew/aaPOp382gfKEbDeEhNfMV+f
Gq3MDJfGuUmfFyDnGLvr/fZu8o83elb446mwtBj+
=cj7k
-----END PGP MESSAGE-----
```