Formal specification for a Cardano wallet

(Version 1.2)

AN IOHK TECHNICAL REPORT

Duncan Coutts
duncan@well-typed.com
duncan.coutts@iohk.io

Edsko de Vries
edsko@well-typed.com

July 16, 2018

Abstract

This document is a formal specification of a wallet for Cardano (or any UTxO-based cryptocurrency). The purpose of this document is to help understand some of the subtleties and give a reasonable starting point for tests and implementations.

To the best of our knowledge, no other existing cryptocurrency wallet comes with such a formal specification. We have therefore attempted to formalise the core functionality of the existing wallet and let our knowledge of the difficulties with the current implementation be a guide in deciding which aspects of the wallet needed more careful thought. We also state and (partially) prove various properties of the wallet models we develop, not only to prove its correctness but also to try and capture our intuitions about what a cryptocurrency wallet *is*, exactly.

1 Introduction

1.1 Why bother with a formal specification?

Cryptocurrency wallets are vital components of a cryptocurrency system and deserve to be designed carefully. Wallets observe and interact with the blockchain ledger to keep track of the currency belonging to a user, and allow them to create and submit new transactions.

Wallets answer the critical question of "what is my balance?". To do this we must first establish what the question means, and it turns out to be not as obvious as it might first appear. What is the meaning of your balance when you have pending transactions that you have sent out but that have not yet been confirmed? What is the appropriate concept of balance in a situation where you have received a large incoming payment, have submitted pending transactions based on the receipt of the incoming payment, but the blockchain has subsequently switched fork such that the incoming payment is not yet present?

To build reliable software we must have reasonable and precise answers to these questions and our definitions must cover all cases and points in time, even the unusual cases. Wallets aimed at casual users may be able to get away without addressing these issues, but there are users that use a wallet as part of automated systems with high transaction rates who care deeply about exactly what their balance is at every moment, including the unusual cases.

The best way to craft definitions that give reasonable and precise answers is to take a formal mathematical approach to creating specifications. The art of formal specification is to simplify and focus on what is essential. This means focusing on the hardest parts and ignoring less important aspects. The process of crafting a good specification involves thinking carefully about the problems and exploring variations to try and find definitions that give a simple overall description. The hard work is in finding a result that makes it all look simple and easy.

In this specification we cover the wallet backend and data model of wallets, we ignore the user interface and take a very abstract view of the cryptography and ledger syntax. We focus on the state of the wallet and the state transitions as the ledger grows and new transactions are created. And crucially, we focus on what the wallet balance is for all such states. To answer the "what is my balance?" question, we end up defining three notions of wallet balance, each appropriate for different purposes.

This document is a combination of both specification and design. We start with a relatively abstract specification. We then define a number of further refinements that take into account certain practicalities including: asymptotic complexity for large wallets; database practicalities and chain rollback and forking.

The specification style is constructive and executable and each of the specification refinements can be run in simulation. This is deliberate as it forms the basis of tests for a full implementation.

1.2 Overview

We start by identifying the key wallet operations. We have reduced this to just querying the wallet balance, updating the wallet as new blocks arrive, and adding new outgoing transactions. We initially ignore details like history tracking and related queries.

We identify the minimal state as the wallet's UTxO, derived solely from the blockchain, and a set of pending transactions. The pending transactions are those transactions that we have created and submitted but that have not yet appeared in the blockchain.

We identify two notions of balance for this basic version of the specification: the total balance and the available balance. The available balance of my wallet is what I can include into a transaction right now and spend. Crucially this does not include change from pending transactions that have not been committed yet. The total balance *does* include the expected change from pending transactions.

This initial basic specification, covering the state, operations and balance has a formal description that fits on a single page.

Our initial basic specification ignores issues related to blockchain forking. It is tempting to hope that because each blockchain is *conceptually* always linear, even in the presence of forks, that we can easily extend the basic specification with support for forks. Unfortunately, this is not the case. While the UTxO of a wallet depends only on the history of the 'current' blockchain fork, the set of pending transactions depends on the real-world history of events, including switching forks. So we must take account of the meaning of pending transactions when switching from one blockchain fork to another.

It turns out that there can be very complex situations with pending transactions once we take forking into account. We define an additional notion of *minimum* balance to cover these situations that corresponds roughly to the 'value at risk': the minimum possible balance across all the known possible futures.

We consider the asymptotic complexity of the executable specifications to help ensure that practical implementations with reasonable performance can be achieved.

Having ignored inessential topics like transaction history tracking in the initial basic specification, we extend the spec to cover tracking of metadata in general, and allowing for history tracking in particular.

We cover transaction submission, to demonstrate that it can be handled in a modular way, without changes to the core wallet state.

Finally we cover the topic of transaction input selection. This is a significant topic in its own right as it is a non-trivial problem for UTxO-based currencies. The wallet design means that input selection can be handled independently, without being intertwined with the core wallet specification. The design clarifies that input selection and transaction signing can be handled asynchronously from the other wallet state changes. This is important since transaction signing in particular may need to be handled on a client device or special hardware and may require user confirmation.

1.3 Version history

Version 1.0, May 4, 2018 First public release.

Version 1.1, May 15, 2018 Corrected definition of updateExpected (Figure 8), added missing value of *txInfo* in rollback (Figure 11), minor corrections to the text.

Version 1.2, July 12, 2018 Included conclusions from our study of input selection. Minor grammatical and stylistic corrections to the text.

Contents

List of Figures

Primitive types

$$
\begin{array}{rcll}
txid & \in & \mathsf{TxId} & \text{transaction id} \\
ix & \in & \mathsf{Ix} & \text{index} \\
addr & \in & \mathsf{Addr} & \text{address} \\
c & \in & \mathsf{Coin} & \text{currency value}
\end{array}
$$

Derived types

$$
\begin{array}{rcrcll}
tx & \in & \mathsf{Tx} & = & (inputs, outputs) \in \mathbb{P}(\mathsf{TxIn}) \times (\mathsf{Ix} \mapsto \mathsf{TxOut}) & \text{transaction} \\
txin & \in & \mathsf{TxIn} & = & (txid, ix) \in \mathsf{TxId} \times \mathsf{Ix} & \text{transaction input} \\
txout & \in & \mathsf{TxOut} & = & (addr, c) \in \mathsf{Addr} \times \mathsf{Coin} & \text{transaction output} \\
utxo & \in & \mathsf{UTxO} & = & txin \mapsto txout \in \mathsf{TxIn} \mapsto \mathsf{TxOut} & \text{unspent transaction outputs} \\
b & \in & \mathsf{Block} & = & tx \in \mathbb{P}(\mathsf{Tx}) & \text{block} \\
pending & \in & \mathsf{Pending} & = & tx \in \mathbb{P}(\mathsf{Tx}) & \text{pending transactions}
\end{array}
$$

Functions

$$
\begin{array}{rcll}
\mathsf{txid} & \in & \mathsf{Tx} \to \mathsf{TxId} & \text{compute transaction id} \\
\mathsf{ours} & \in & \mathsf{Addr} \to \mathbb{B} & \text{addresses that belong to the wallet}
\end{array}
$$

Filtered sets

$$
\begin{aligned}
\mathsf{Addr}_{\mathsf{ours}} &= \{a \mid a \in \mathsf{Addr},\ \mathsf{ours}\ a\} \\
\mathsf{TxOut}_{\mathsf{ours}} &= \mathsf{Addr}_{\mathsf{ours}} \times \mathsf{Coin}
\end{aligned}
$$

Figure 1: Basic Definitions

2 Preliminaries

2.1 UTxO-style Accounting

The wallet specification will be based on the formalisation of UTxO style accounting in (Zahnentferner, 2018). The basic definitions are summarised in Figure 1. A full explanation of UTxO style accounting is beyond the scope of this document, and we refer the reader to the aforementioned paper. Here we will only comment on some details.

The computation txid of transaction IDs (hashes) is assumed to be 'effectively' injective[1] so that a transaction ID uniquely identifies a transaction. Transaction indexes, used to index transaction outputs, will typically be natural numbers, but this is not necessary. Currency values are essentially just natural numbers.

Addresses stand for cryptographic public keys. In this presentation we can keep them quite abstract, it is merely a large set of distinct values. The predicate ours tells us if a particular address 'belongs' to our wallet. This corresponds in the real implementation to us being able to identify addresses that correspond to our wallet where we can derive the keypair used to generate that address, and to sign transactions that pay from that address. If it aids comprehension, it may be worth noting that if this specification were elaborated to cover public/private key pairs, then we would model this as a partial function that returns the keypair as evidence $\mathsf{ours} \in \mathsf{Addr} \mapsto (\mathsf{PubKey} \times \mathsf{PrivKey})$.

The intuition behind the unspent transaction outputs type UTxO is that it records all the transaction inputs in our wallet that we have available to spend from, and how much cash is available at each one. We will see that it will be derived solely from the chain, and not any other wallet state. Moreover, the UTxO maintained by the wallet will only include the outputs that are available to the wallet to spend (i.e. range within $\mathsf{TxOut}_{\mathsf{ours}}$), and not the UTxO of the entire blockchain.

Somewhat unusually, we model a block as a *set* of transactions rather than a sequence. For *validating* a block it is essential to represent it as a sequence, but a wallet does not need to validate blocks; it can rely on its associated node to do that. The order of transactions in a block does not turn out to matter for any wallet operation, and the choice of set representation makes it possible to share useful operations between the set of pending transactions and the set of transactions in a block.

[1] A quick counting argument shows this is impossible for given finite representations. The assumption is justified on the basis that we use cryptographically strong hash functions so that computing clashes is computationally impractical.

2.2 Operations on UTxO

For convenience we will define a number of operations to filter UTxOs:

$$\begin{aligned} ins \lhd utxo &= \{i \mapsto o \mid i \mapsto o \in utxo,\ i \in ins\} && \text{domain restriction} \\ ins \not\lhd utxo &= \{i \mapsto o \mid i \mapsto o \in utxo,\ i \notin ins\} && \text{domain exclusion} \\ utxo \rhd outs &= \{i \mapsto o \mid i \mapsto o \in utxo,\ o \in outs\} && \text{range restriction} \end{aligned}$$

Lemma 2.1 (Properties of UTxO operations)**.**

$$ins \lhd u \subseteq u \tag{2.1.1}$$
$$ins \not\lhd u \subseteq u \tag{2.1.2}$$
$$u \rhd outs \subseteq u \tag{2.1.3}$$
$$ins \lhd (u \cup v) = (ins \lhd u) \cup (ins \lhd v) \tag{2.1.4}$$
$$ins \not\lhd (u \cup v) = (ins \not\lhd u) \cup (ins \not\lhd v) \tag{2.1.5}$$
$$(\mathrm{dom}\, u \cap ins) \lhd u = ins \lhd u \tag{2.1.6}$$
$$(\mathrm{dom}\, u \cap ins) \not\lhd u = ins \not\lhd u \tag{2.1.7}$$
$$(\mathrm{dom}\, u \cup ins) \not\lhd u \cup v = (ins \cup \mathrm{dom}\, u) \not\lhd v \tag{2.1.8}$$
$$ins \not\lhd u = (\mathrm{dom}\, u \setminus ins) \lhd u \tag{2.1.9}$$

We omit proofs for most of these properties, as they are straight-forward. Here is just one example:

Proof ((2.1.4)*).*

$$\begin{aligned} & ins \lhd (u \cup v) \\ &= \{i \mapsto o \mid i \mapsto o \in (u \cup v), i \in ins\} \\ &= \{i \mapsto o \mid (i \mapsto o \in u) \vee (i \mapsto o \in v), i \in ins\} \\ &= \{i \mapsto o \mid (i \mapsto o \in u, i \in ins) \vee (i \mapsto o \in v, i \in ins)\} \\ &= \{i \mapsto o \mid i \mapsto o \in u, i \in ins\} \cup \{i \mapsto o \mid i \mapsto o \in v, i \in ins\} \\ &= (ins \lhd u) \cup (ins \lhd v) \end{aligned}$$

□

We will also make use of two preorders on UTxOs:

Definition 2.2 ($u \subseteq v$)**.** We will write $u \subseteq v$ whenever

$$\forall (tx, i) \mapsto (addr, c) \in u.\, (tx, i) \mapsto (addr, c) \in v$$

Definition 2.3 ($u \sqsubseteq v$)**.** We will write $u \sqsubseteq v$ whenever $u \subseteq v$ and moreover

$$\forall (tx, i) \mapsto (addr, c) \in u.\, \forall (tx, i') \mapsto (addr', c') \in v.\, (tx, i') \mapsto (addr', c') \in u$$

The latter preorder corresponds to a subset of the *transactions* in the UTxO, rather than the individual outputs.

Queries

$$\mathsf{totalBalance} \in \mathsf{Wallet} \to \mathsf{Coin}$$
$$\mathsf{availableBalance} \in \mathsf{Wallet} \to \mathsf{Coin}$$

Atomic updates

$$\mathsf{applyBlock} \in \mathsf{Block} \to \mathsf{Wallet} \to \mathsf{Wallet}$$
$$\mathsf{newPending} \in \mathsf{Tx} \to \mathsf{Wallet} \to \mathsf{Wallet}$$

Figure 2: Wallet interface

2.3 Other auxiliary operations

We will make frequent use of the following operations throughout this specification.

$$\mathsf{txins} \in \mathbb{P}(\mathsf{Tx}) \to \mathbb{P}(\mathsf{TxIn})$$
$$\mathsf{txins}\ txs = \bigcup \{inputs \mid (inputs, \_) \in txs\}$$

$$\mathsf{txouts} \in \mathbb{P}(\mathsf{Tx}) \to \mathsf{UTxO}$$
$$\mathsf{txouts}\ txs = \left\{ (\mathsf{txid}\ tx, ix) \mapsto txout \;\middle|\; \begin{array}{ll} tx & \in txs \\ (\_, outputs) & = tx \\ ix \mapsto txout & \in outputs \end{array} \right\}$$

$$\mathsf{balance} \in \mathsf{UTxO} \to \mathsf{Coin}$$
$$\mathsf{balance}\ utxo = \sum_{(\_ \mapsto (\_, c)) \in utxo} c$$

Definition 2.4 (Dependence)**.** We say that transaction t_2 *depends on* transaction t_1 if and only if

$$\exists ix.\ (\mathsf{txid}\ t_1, ix) \in \mathsf{txins}\ \{t_2\}$$

Definition 2.5 (Set of independent transactions)**.** We will refer to a set of transactions txs as a *set of independent transactions* when there are no transactions that depend on other transactions in the set. Formally

$$\mathsf{txins}\ txs \cap \mathrm{dom}(\mathsf{txouts}\ txs) = \emptyset$$

Lemma 2.6 (Properties of balance)**.** There are a couple useful lemmas about balance distributing over other operators.

$$\mathsf{balance}\ (u \cup v) = \mathsf{balance}\ u + \mathsf{balance}\ v \qquad \text{if } \mathrm{dom}\, u \cap \mathrm{dom}\, v = \emptyset \tag{2.6.1}$$
$$\mathsf{balance}\ (ins \lhd\!\!\!\!- u) = \mathsf{balance}\ u - \mathsf{balance}\ (ins \lhd u) \tag{2.6.2}$$

3 The Basic Model

The main wallet interface is shown in Figure 2. There are only a small number of wallet operations of interest. We can:

- enquire as to the balance of the wallet (total balance and available balance).
- make a new wallet state by 'applying' a block to a wallet state
- make a new wallet state by adding a new pending transaction to a wallet state

We intentionally left the definition of Wallet abstract in this figure, as we will consider various different concrete instantiations throughout this specification. We distinguish between queries of the wallet state and atomic updates; we emphasise that the latter should be 'atomic' because although in a purely mathematical specification this is not particularly meaningful, in a real implementation if such updates consist of multiple smaller updates, the intermediate states should not be observable. For many instantiations we will specify state invariants that are expected to be preserved by all state updates.

The most basic model is shown in Figure 3. This model, as indeed every other model in this specification, is *abstract*. We are not concerned with specific data representation formats or low level implementation details. Such issues are important, but should be considered only after we understand the abstract model: it is not useful to consider implementation details until we have a good understanding of the requirements.

3.1 Updating the UTxO

In order to update the UTxO, updateUTxO first adds the new outputs from the block, and then removes the inputs spent in the block. It would be incorrect to use the definition

$$(\mathsf{txins}\ b \mathbin{⋪} utxo) \cup (\mathsf{txouts}\ b \rhd \mathsf{TxOut}_{\mathsf{ours}}) \qquad \text{(incorrect)}$$

The difference crops up when one considers transactions within the block b that depend on each other: that is, where the output of one transaction is used as the input of another within the same block. To make this intuition clearer, we can define a function that computes only the 'new' outputs from a block (outputs that are not spent within that same block):

Definition 3.1 (Block UTxO)**.**

$$\mathsf{new}\ b = \mathsf{txins}\ b \mathbin{⋪} (\mathsf{txouts}\ b \rhd \mathsf{TxOut}_{\mathsf{ours}})$$

We can then prove that updateUTxO adds precisely the new outputs of a block to the UTxO:

Lemma 3.2. $\mathrm{dom}\ u \mathbin{⋪} \mathsf{updateUTxO}\ b\ u = \mathsf{new}\ b$

Proof.

$$\begin{aligned}
&\mathrm{dom}\ u \mathbin{⋪} \mathsf{updateUTxO}\ b\ u \\
=\ &\mathrm{dom}\ u \mathbin{⋪} \Big(\mathsf{txins}\ b \mathbin{⋪} (u \cup (\mathsf{txouts}\ b \rhd \mathsf{TxOut}_{\mathsf{ours}}))\Big) \\
=\ &(\mathrm{dom}\ u \cup \mathsf{txins}\ b) \mathbin{⋪} (u \cup (\mathsf{txouts}\ b \rhd \mathsf{TxOut}_{\mathsf{ours}})) \\
=\ &(\mathrm{dom}\ u \cup \mathsf{txins}\ b) \mathbin{⋪} (\mathsf{txouts}\ b \rhd \mathsf{TxOut}_{\mathsf{ours}}) && \{\ (2.1.8)\ \} \\
=\ &\mathsf{txins}\ b \mathbin{⋪} (\mathsf{txouts}\ b \rhd \mathsf{TxOut}_{\mathsf{ours}}) = \mathsf{new}\ b && \{\ \text{Precondition to applyBlock}\ \}
\end{aligned}$$

□

This proof relies on the precondition to applyBlock, which simply says that new transactions in a new block should have transaction IDs that do not occur in the UTxO of the existing chain (or wallet); this should be a straightforward property of the blockchain.

Note from the definitions of applyBlock and newPending (and by induction from $w_\emptyset$) that the wallet UTxO depends only on the blocks and not the pending transactions.

3.2 Update the pending set

The definition of updatePending is pleasantly simple: the definition covers the case of one of our own transactions being committed, as well as transactions submitted by other instances of our wallet invalidating our pending transactions. Both are covered because all we are doing is removing pending transactions that have had any (or indeed all) of their inputs spent. The fact that it could be made so simple (without making special provisions for these specific cases) was an "ah hah" moment in the early development of this specification.

The precondition to newPending states that new pending transactions can only spend outputs in the wallet's current UTxO. Alternatively, we could require that

$$ins \subseteq \mathrm{dom}(\mathsf{total}(utxo, pending))$$

Wallet state

$$(utxo, pending) \in \mathsf{Wallet} = \mathsf{UTxO} \times \mathsf{Pending}$$
$$w_\emptyset \in \mathsf{Wallet} = (\emptyset, \emptyset)$$

Queries

$$\mathsf{availableBalance} = \mathsf{balance} \circ \mathsf{available}$$
$$\mathsf{totalBalance} = \mathsf{balance} \circ \mathsf{total}$$

Atomic updates

$$\mathsf{applyBlock}\ b\ (utxo, pending) = (\mathsf{updateUTxO}\ b\ utxo,\ \mathsf{updatePending}\ b\ pending)$$
$$\mathsf{newPending}\ tx\ (utxo, pending) = (utxo,\ pending \cup \{tx\})$$

Preconditions

$$\mathsf{newPending}\ (ins, outs)\ (utxo, pending)$$
$$\quad \textbf{requires}\ ins \subseteq \mathrm{dom}(\mathsf{available}\ (utxo, pending))$$
$$\mathsf{applyBlock}\ b\ (utxo, pending)$$
$$\quad \textbf{requires}\ \mathrm{dom}(\mathsf{txouts}\ b) \cap \mathrm{dom}\ utxo = \emptyset$$

Auxiliary functions

$$\mathsf{available}, \mathsf{total} \in \mathsf{Wallet} \to \mathsf{UTxO}$$
$$\mathsf{available}\ (utxo, pending) = \mathsf{txins}\ pending \mathbin{⩤} utxo$$
$$\mathsf{total}\ (utxo, pending) = \mathsf{available}\ \ (utxo, pending) \cup \mathsf{change}\ pending$$

$$\mathsf{change} \in \mathsf{Pending} \to \mathsf{UTxO}$$
$$\mathsf{change}\ pending = \mathsf{txouts}\ pending \rhd \mathsf{TxOut}_{\mathsf{ours}}$$

$$\mathsf{updateUTxO} \in \mathsf{Block} \to \mathsf{UTxO} \to \mathsf{UTxO}$$
$$\mathsf{updateUTxO}\ b\ utxo = \mathsf{txins}\ b \mathbin{⩤} (utxo \cup (\mathsf{txouts}\ b \rhd \mathsf{TxOut}_{\mathsf{ours}}))$$

$$\mathsf{updatePending} \in \mathsf{Block} \to \mathsf{Pending} \to \mathsf{Pending}$$
$$\mathsf{updatePending}\ b\ p = \{tx \mid tx \in p, (inputs, \_) = tx, inputs \cap \mathsf{txins}\ b = \emptyset\}$$

Figure 3: The basic model

This would allow transactions that spend from change addresses, allowing multiple in-flight transactions that depend on each other. We disallow this for pragmatic reasons (long chains of pending transactions make it more difficult for nodes to resubmit transactions that for some reason did not get included in the blockchain). However, as we will see later, once we add support for rollback to the wallet we cannot guarantee anymore that there are no dependent pending transactions, even with this side condition.

One simple property of updatePending we will need later is that

Lemma 3.3.

$$\mathsf{updatePending}\ b\ \mathit{pending} \subseteq \mathit{pending}$$

3.3 Invariants

We would hope to prove the following invariants are true for all wallet values. Proofs would proceed by induction on the wallet construction (empty wallet $w_\emptyset$, applyBlock, newPending). Not all of these invariants will be true in all models.

Invariant 3.4 (Pending transactions only spend from our current UTxO)**.**

$$\mathsf{txins}\ \mathit{pending} \subseteq \mathrm{dom}\ \mathit{utxo}$$

Note that Invariant 3.4 only holds if we do not allow dependent in-flight transactions. If we do allow dependent ones then the spent set of the pending includes change addresses that are not yet in the wallet UTxO. Additionally, as we will see in Section 6, once we add rollback then this invariant no longer holds.

Invariant 3.5 (The wallet UTxO only covers addresses that belong to the wallet)**.**

$$\mathrm{range}\ \mathit{utxo} \subseteq \mathsf{TxOut}_{\mathsf{ours}}$$

Invariant 3.6 (Transactions are removed from the pending set once they are included in the UTxO)**.**

$$\mathrm{dom}(\mathsf{change}\ \mathit{pending}) \cap \mathrm{dom}(\mathsf{available}\ (\mathit{utxo}, \mathit{pending})) = \emptyset$$

Given these invariants, we can prove a few simple lemmas.

Lemma 3.7.

$$\mathrm{dom}(\mathsf{available}\ (\mathit{utxo}, \mathit{pending})) \subseteq \mathrm{dom}(\mathit{utxo})$$

Proof. Follows immediately from (2.1.2). □

Lemma 3.8 (Change not in UTxO)**.**

$$\mathrm{dom}(\mathsf{change}\ \mathit{pending}) \cap \mathrm{dom}(\mathit{utxo}) = \emptyset$$

Proof. Follows from Invariant 3.6 and Lemma 3.7. □

Lemma 3.8 is not very deep. All new transactions should have fresh IDs, and thus cannot be in the existing wallet UTxO.

Finally we can state a lemma that relates change, available, and total:

Lemma 3.9. Given a wallet $w = (\mathit{utxo}, \mathit{pending})$,

$$\mathsf{change}\ \mathit{pending} \ \cup \ \mathsf{available}\ w \ = \ \mathsf{total}\ w \tag{3.9.1}$$

$$\mathsf{balance}\ (\mathsf{change}\ \mathit{pending}) + \mathsf{balance}\ (\mathsf{available}\ w) = \mathsf{balance}\ (\mathsf{total}\ w) \tag{3.9.2}$$

Proof. (3.9.1) follows directly from the definition. (3.9.2) follows from (3.9.1) and (2.6.1) with Invariant 3.6. □

$$
\begin{aligned}
\mathsf{balance}\ u &\in \mathcal{O}\left(|u|\right) \\
\mathsf{txins}\ txs &\in \mathcal{O}\left(\mathsf{nlogn}\ |\mathsf{txins}\ txs|\right) \\
\mathsf{txouts}\ txs &\in \mathcal{O}\left(\mathsf{nlogn}\ |\mathsf{txouts}\ txs|\right)) \\
\mathsf{available}\ (u,p) &\in \mathcal{O}\left(\mathsf{join}\ |\mathsf{txins}\ p|\ |u|\right) \\
\mathsf{change}\ p &\in \mathcal{O}\left(\mathsf{nlogn}\ |\mathsf{txouts}\ p|\right) \\
\mathsf{total}\ (u,p) &\in \mathcal{O}\left(\begin{array}{l} \mathsf{join}\ |\mathsf{txins}\ p|\ |u| \\ +\ \mathsf{join}\ |\mathsf{txouts}\ p|\ |u| \\ +\ \mathsf{nlogn}\ |\mathsf{txouts}\ p| \end{array}\right) \\
\mathsf{availableBalance}\ (u,p) &\in \mathcal{O}\left(\begin{array}{l} |u| \\ +\ \mathsf{join}\ |\mathsf{txins}\ p|\ |u| \end{array}\right) \\
\mathsf{totalBalance}\ (u,p) &\in \mathcal{O}\left(\begin{array}{l} |u| \\ +\ \mathsf{join}\ |\mathsf{txins}\ p|\ |u| \\ +\ \mathsf{join}\ |\mathsf{txouts}\ p|\ |u| \\ +\ \mathsf{nlogn}\ |\mathsf{txouts}\ p| \end{array}\right) \\
\mathsf{newPending}\ tx\ (u,p) &\in \mathcal{O}\left(\log |p|\right) \\
\mathsf{updateUTxO}\ b\ u &\in \mathcal{O}\left(\begin{array}{l} \mathsf{join}\ |\mathsf{txins}\ b|\ |u| \\ +\ \mathsf{join}\ |\mathsf{txouts}\ b|\ |u| \end{array}\right) \\
\mathsf{updatePending}\ b\ p &\in \mathcal{O}\left(\mathsf{nlogn}\ |\mathsf{txins}\ b| + \sum_{(inputs,\_)\in p} \mathsf{join}\ |inputs|\ |\mathsf{txins}\ b|\right)
\end{aligned}
$$

Figure 4: Algorithmic complexity of the operations in the basic model

3.4 Complexity

The goal of this specification is not merely to describe what the *correct* behaviour of a wallet should be, but also to study the asymptotic complexity of the key operations of the wallet. Put another way, we want to study how the performance of the wallet scales when the wallet's state gets larger. Specifically, we would like to find out which operations are the most expensive, and how we might address that.

The basic model is intended to be comprehensible, not efficient. Let us take the initial description as a naïve implementation and consider the asymptotic complexity of the major operations. We will explore other approaches with better asymptotic complexity in later sections.

Many of the basic operations we need to consider are set and map operations implemented using ordered balanced trees. Many of these operations have the following complexity, where M and N are the sizes of the two sets or maps.

$$
\begin{aligned}
\mathsf{nlogn}\ N &= N \cdot \log N \\
\mathsf{join}\ M\ N &= M \cdot \log\ (N/M+1) \quad \text{for } M \leq N
\end{aligned}
$$

This join comes from Blelloch et al. (2016); observe that when $M = N$, $\mathcal{O}\left(\mathsf{join}\ M\ N\right)$ is is simply $\mathcal{O}\left(M\right)$, and when N is much larger than M (our typical case), $\mathcal{O}\left(\mathsf{join}\ M\ N\right)$ is bounded by $\mathcal{O}\left(M \cdot \log N\right)$. The complexity of the major operations are then as given in Figure 4.

It is worth knowing the expected order of magnitudes of the sizes of the UTxO and pending sets. The UTxO can be quite large, for example $|utxo| \leq 10^6$, while the pending set will typically be small, usually around $|pending| \leq 3$, while $|pending| = 100$ would be extreme. Similarly, the number of inputs and outputs in any individual transaction is not large (at most a few hundred, and typically much less than that). The current bound on the total size of a transaction is 64kB.

4 Caching Balance

The asymptotic complexity of the naïve implementations (Section 3.4) are in fact mostly good enough. If we assume that the number of pending transactions, and the number of inputs and outputs for individual transactions is not large, then the only problematic operations are availableBalance and totalBalance, which are both linear in $|u|$ (the size of the UTxO).

In this section we derive a variation on the basic model which caches the balance for the UTxO to address this problem.

4.1 Factoring out the UTxO balance

Lemma 4.1.

$$\mathsf{availableBalance}\ (utxo, pending) = \mathsf{balance}\ utxo - \mathsf{balance}\ (\mathsf{txins}\ pending \lhd utxo)$$

Proof.

$$\begin{aligned}
&\mathsf{availableBalance}\ (utxo, pending) \\
&= \mathsf{balance}\ (\mathsf{available}\ (utxo, pending)) \\
&= \mathsf{balance}\ (\mathsf{txins}\ pending \not\lhd utxo) \\
&= \mathsf{balance}\ utxo - \mathsf{balance}\ (\mathsf{txins}\ pending \lhd utxo) && \{\ (2.6.2)\ \}
\end{aligned}$$

□

Lemma 4.2.

$$\mathsf{totalBalance}\ (utxo, pending) = \mathsf{availableBalance}\ (utxo, pending) + \mathsf{balance}\ (\mathsf{change}\ pending)$$

Proof.

$$\begin{aligned}
&\mathsf{totalBalance}\ (utxo, pending) \\
&= \mathsf{balance}\ (\mathsf{available}\ (utxo, pending) \cup \mathsf{change}\ pending) \\
&= \mathsf{balance}\ (\mathsf{available}\ (utxo, pending)) + \mathsf{balance}\ (\mathsf{change}\ pending) && \{\ (2.6.1),\ \text{Invariant 3.6}\ \} \\
&= \mathsf{availableBalance}\ (utxo, pending) + \mathsf{balance}\ (\mathsf{change}\ pending)
\end{aligned}$$

□

Note the complexity of these operations

$$\begin{aligned}
\mathsf{balance}\ utxo &\in \mathcal{O}\ (|utxo|) \\
\mathsf{balance}\ (\mathsf{txins}\ pending \lhd utxo) &\in \mathcal{O}\ (\mathsf{join}\ |\mathsf{txins}\ pending|\ |utxo|) \\
\mathsf{balance}\ (\mathsf{change}\ pending) &\in \mathcal{O}\ (\mathsf{nlogn}\ |\mathsf{txouts}\ pending|)
\end{aligned}$$

Only the first is expensive. This suggests that we should at least cache the balance of the UTxO. If we only cache the UTxO balance then the available and total balances are not too expensive to compute. Of course we could cache more, but each extra value we cache adds complexity to the design, and additional proof obligations.

4.2 Keeping cached balance up to date

Now that we've factored out the common balance $utxo$ term, let's cache this as a new field σ in the wallet state. Since applyBlock is the function that modifies the UTxO, we will need to modify it to additionally update the cached UTxO balance.

Our starting point is

$$\begin{aligned}
&\mathsf{applyBlock}\ b\ (utxo, pending, \sigma) = (utxo', pending', \sigma') \\
&\quad \text{where} \\
&\qquad utxo' = \mathsf{updateUTxO}\ b\ utxo \\
&\qquad pending' = \mathsf{updatePending}\ b\ pending \\
&\qquad \sigma' = \mathsf{balance}\ utxo'
\end{aligned}$$

If we focus on the interesting bits and expand this out a couple steps we get

$$\begin{aligned}
utxo' &= \mathsf{txins}\ b \mathbin{⩤} (utxo \cup (\mathsf{txouts}\ b \rhd \mathsf{TxOut}_{\mathsf{ours}})) \\
\sigma' &= \mathsf{balance}\ utxo'
\end{aligned}$$

For convenience we define

$$utxo^{+} = \mathsf{txouts}\ b \rhd \mathsf{TxOut}_{\mathsf{ours}}$$

And use it, giving us

$$\begin{aligned}
utxo^{+} &= \mathsf{txouts}\ b \rhd \mathsf{TxOut}_{\mathsf{ours}} \\
utxo' &= \mathsf{txins}\ b \mathbin{⩤} (utxo \cup utxo^{+}) \\
\sigma' &= \mathsf{balance}\ (\mathsf{txins}\ b \mathbin{⩤} (utxo \cup utxo^{+}))
\end{aligned}$$

Applying (2.6.2) to distribute balance over $⩤$ gives us

$$\sigma' = \mathsf{balance}\ (utxo \cup utxo^{+}) - \mathsf{balance}\ (\mathsf{txins}\ b \lhd (utxo \cup utxo^{+}))$$

Relying on the precondition to applyBlock, we can apply (2.6.1) to distribute balance over $\cup$ to give us

$$\begin{aligned}
\sigma' &= \mathsf{balance}\ utxo + \mathsf{balance}\ utxo^{+} - \mathsf{balance}\ (\mathsf{txins}\ b \lhd (utxo \cup utxo^{+})) \\
&= \sigma + \mathsf{balance}\ utxo^{+} - \mathsf{balance}\ (\mathsf{txins}\ b \lhd (utxo \cup utxo^{+}))
\end{aligned}$$

The extra things we have to compute turn out not to be expensive

$$\begin{aligned}
\mathsf{balance}\ (\mathsf{txouts}\ b \rhd \mathsf{TxOut}_{\mathsf{ours}}) &\in \mathcal{O}\ (\mathsf{nlogn}\ |\mathsf{txouts}\ b|) \\
\mathsf{balance}\ (\mathsf{txins}\ b \lhd utxo) &\in \mathcal{O}\ (\mathsf{join}\ |\mathsf{txins}\ b|\ |utxo|)
\end{aligned}$$

Putting everything back together, and defining $utxo^{-}$ for symmetry, gives us an extension of the basic model with a cached UTxO balance, shown in Figure 5.

5 Prefiltering

5.1 Motivation

The applyBlock b operation is problematic in a setting where it is implemented as an operation on a local wallet database and where the database implementation keeps a transaction log containing all the inputs of each transaction. The log would contain the full blocks received by the wallet, which – at current constants of a maximum block of 2 MB and a slot length of 20 seconds – would mean a worst-case log growth rate of 360 MB/hour.

Wallet state

$$(utxo, pending, \sigma) \in \mathsf{Wallet} = \mathsf{UTxO} \times \mathsf{Pending} \times \mathsf{Coin}$$
$$w_\emptyset \in \mathsf{Wallet} = (\emptyset, \emptyset, 0)$$

State invariant

$$\sigma = \mathsf{balance}\ utxo$$

Queries

$$\mathsf{availableBalance}\ (utxo, pending, \sigma) = \sigma - \mathsf{balance}\ (\mathsf{txins}\ pending \lhd utxo)$$
$$\mathsf{totalBalance}\ (utxo, pending, \sigma) = \mathsf{availableBalance}\ (utxo, pending, \sigma) + \mathsf{balance}\ (\mathsf{change}\ pending)$$

Atomic updates

$$\mathsf{applyBlock}\ b\ (utxo, pending, \sigma) = (utxo', pending', \sigma')$$
$$\begin{aligned} \text{where}\ \ pending' &= \mathsf{updatePending}\ b\ pending \\ utxo^+ &= \mathsf{txouts}\ b \rhd \mathsf{TxOut}_{\mathsf{ours}} \\ utxo^- &= \mathsf{txins}\ b \lhd (utxo \cup utxo^+) \\ utxo' &= \mathsf{txins}\ b \not\lhd (utxo \cup utxo^+) \\ \sigma' &= \sigma + \mathsf{balance}\ utxo^+ - \mathsf{balance}\ utxo^- \end{aligned}$$

Figure 5: Basic model with cached balance

Wallet state

As in the basic model with cached balance (Figure 5).

Atomic updates

$$\mathsf{applyBlock}\ b = \mathsf{applyBlock}'\ \Big(\mathsf{txins}\ b \cap \mathrm{dom}(utxo \cup utxo^+), utxo^+\Big)$$
$$\text{where}\ utxo^+ = \mathsf{txouts}\ b \rhd \mathsf{TxOut}_{\mathsf{ours}}$$

Auxiliary

$$\mathsf{applyBlock}'\ (txins_b, txouts_b)\ (utxo, pending, \sigma) = (utxo', pending', \sigma')$$
$$\begin{aligned} \text{where}\ \ pending' &= \{tx \mid tx \in pending, (inputs, \_) = tx, inputs \cap txins_b = \emptyset\} \\ utxo^+ &= txouts_b \\ utxo^- &= txins_b \lhd (utxo \cup utxo^+) \\ utxo' &= txins_b \not\lhd (utxo \cup utxo^+) \\ \sigma' &= \sigma + \mathsf{balance}\ utxo^+ - \mathsf{balance}\ utxo^- \end{aligned}$$

Figure 6: Wallet with prefiltering

5.2 Derivation

The goal is to define an auxiliary function to applyBlock which only needs the 'relevant' information from the block. Since applyBlock is only defined in terms of the inputs and outputs of the block, we can easily define a variation applyBlock′, shown in Figure 6, that accepts these as two separate arguments.

Letting $utxo^+ = \mathsf{txouts}\ b \rhd \mathsf{TxOut}_{\mathsf{ours}}$ we trivially we have that

$$\mathsf{applyBlock}\ b = \mathsf{applyBlock}'\ (\mathsf{txins}\ b, utxo^+)$$

but we haven't gained much yet because although we only pass in 'our' outputs, we still pass in *all* inputs of the block. However, Lemma 5.1 shows how we can filter the inputs also, justifying the definition of the wallet with prefiltering (Figure 6).

Lemma 5.1.

$$\mathsf{applyBlock}\ b = \mathsf{applyBlock}'\ \Big(\mathsf{txins}\ b \cap \mathrm{dom}(utxo \cup utxo^+), utxo^+\Big)$$

Proof. Since there are three separate uses of $txins_b$ in applyBlock′, proving Lemma 5.1 boils down to showing three things:

1. (Definition of $utxo^-$)

$$\begin{aligned} &\mathsf{txins}\ b &&\lhd (utxo \cup utxo^+) \\ =\ &\mathsf{txins}\ b \cap \mathrm{dom}(utxo \cup utxo^+) &&\lhd (utxo \cup utxo^+) \end{aligned}$$

2. (Definition of $utxo'$)

$$\begin{aligned} &\mathsf{txins}\ b &&\not\lhd (utxo \cup utxo^+) \\ =\ &\mathsf{txins}\ b \cap \mathrm{dom}(utxo \cup utxo^+) &&\not\lhd (utxo \cup utxo^+) \end{aligned}$$

3. (Definition of *pending*′)

$$\forall (ins, outs) \in pending \cdot \left(\begin{array}{lll} & ins \cap (\mathsf{txins}\ b) & = \emptyset \\ \text{iff} & ins \cap (\mathsf{txins}\ b \cap \mathrm{dom}(utxo \cup utxo^+)) & = \emptyset \end{array} \right)$$

Equalities (1) and (2) follow immediately from Lemma (2.1.6) and (2.1.7). The backwards direction of (3) is trivial; the forwards direction follows from Invariant 3.4. □

5.3 Consequences

The downside is that in order to do this prefiltering we need to know the current value of our $utxo$, which means we need to do a database read and then a database write in two separate transactions. While in principle this means we might suffer from the lost update problem, in practice block updates need to be processed sequentially *anyway*. It does however impose a proof obligation on the rest of the system:

Proof Obligation 5.2. *Only* applyBlock *modifies the wallet's UTxO.*

Note that there are at least two possible alternative approaches:

- The wallet runs as part of a full node, and that full node maintains the full UTxO of the blockchain. When the full node receives a new block, that block must be consistent with the state of the blockchain, and hence the full node can decorate all inputs with the corresponding addresses before passing the block to the wallet. This would make the filtering operation in the wallet trivial and stateless. (This is in fact the case for the current wallet implementation.)
- We can also push the problem further upstream and specify that the resolved addresses must be listed alongside the transaction IDs in the transaction inputs themselves. This would effectively be a form of caching in the blockchain, and may be beneficial elsewhere also.

Wallet state

$$checkpoints \in \mathsf{Wallet} = [\mathsf{UTxO} \times \mathsf{Pending}]$$
$$w_\emptyset \in \mathsf{Wallet} = [(\emptyset, \emptyset)]$$

Atomic updates

$$\begin{aligned}
&\mathsf{applyBlock}\ b\ ((utxo, pending) : checkpoints) = \\
&\quad ((\mathsf{updateUTxO}\ b\ utxo, \mathsf{updatePending}\ b\ pending) : (utxo, pending) : checkpoints) \\
&\mathsf{newPending}\ tx\ ((utxo, pending) : checkpoints) = \\
&\quad (utxo, pending \cup \{tx\}) : checkpoints \\
&\mathsf{rollback}\ ((utxo, pending) : (utxo', pending') : checkpoints)) = \\
&\quad (utxo', pending \cup pending') : checkpoints
\end{aligned}$$

Auxiliary

$$\begin{aligned}
\mathsf{available}\ (utxo, pending) &= \mathsf{txins}\ pending \not\triangleleft utxo && \text{(unchanged)} \\
\mathsf{change}\ pending &= \mathsf{txins}\ pending \not\triangleleft (\mathsf{txouts}\ pending \rhd \mathsf{TxOut}_{\mathsf{ours}})
\end{aligned}$$

Figure 7: Basic model with rollback

6 Rollback

The possible presence of forks in the blockchain means that we may occasionally have to roll back and 'undo' calls to applyBlock, reverting to an older version of the UTxO. When we *apply* a block, pending transactions may become confirmed and are therefore removed from the *pending* set. When we roll back, those transactions may once again become pending and should therefore be reintroduced into *pending*. However, the converse is *not* true: when we roll back, currently pending transactions will *remain* pending. After all, those pending transactions may still make it into the block chain; indeed, may already have made it into the fork that we are transitioning to. In other words, rolling back may *increase* the size of the pending set but never decrease it.

6.1 Model

The basic model with support for rollback is shown in Figure 7. In this model the wallet state is a non-empty list of *checkpoints*, the value of the wallet at various times throughout its lifetime; each call to applyBlock introduces a new checkpoint. The initial wallet is the singleton list; We cannot roll back a wallet that contains only a single checkpoint (a rollback before any blocks have been applied would anyway not make semantic sense).

After a rollback we have transactions in *pending* that spend inputs that are not available in the wallet's UTxO (we will explore this issue in detail in Section 7). Nonetheless, the definition of available can remain pretty much unchanged, since that only considers inputs that *are* in the UTxO anyway.

However, rollbacks also mean that we may end up with pending transactions that depend on other pending transactions. This means that the definition of change must be modified to remove any outputs that are spent by other pending transactions. Note that the precondition to newPending continues to ensure that we cannot introduce any *new* dependent pending transactions. This is still useful for keeping the number of dependent transactions down.

6.2 Properties

Lemma 6.1.

$$\mathsf{rollback} \circ \mathsf{applyBlock}\ b = \mathsf{id}$$

Proof.

$$\begin{aligned}
& \mathsf{rollback}\ (\mathsf{applyBlock}\ b\ ((u,p):cs)) \\
= \ & \mathsf{rollback}\ ((\mathsf{updateUTxO}\ b\ u, \mathsf{updatePending}\ b\ p) : (u,p) : cs) \\
= \ & (u, (\mathsf{updatePending}\ b\ p) \cup p) : cs \\
= \ & (u,p) : cs && \{\ \text{Lemma 3.3}\ \}
\end{aligned}$$

□

Moreover, we have

Lemma 6.2.

$$\mathsf{rollback} \circ \mathsf{newPending}\ tx \circ \mathsf{applyBlock}\ b = \mathsf{newPending}\ tx$$

(if we ignore the side condition to newPending).

Proof.

$$\begin{aligned}
& \mathsf{rollback}\ (\mathsf{newPending}\ tx\ (\mathsf{applyBlock}\ b\ ((u,p):cs))) \\
= \ & \mathsf{rollback}\ (\mathsf{newPending}\ tx\ ((\mathsf{updateUTxO}\ b\ u, \mathsf{updatePending}\ b\ p) : (u,p) : cs)) \\
= \ & \mathsf{rollback}\ ((\mathsf{updateUTxO}\ b\ u, (\mathsf{updatePending}\ b\ p) \cup \{tx\}) : (u,p) : cs) \\
= \ & (u, ((\mathsf{updatePending}\ b\ p) \cup \{tx\}) \cup p) : cs \\
= \ & (u, p \cup \{tx\}) : cs && \{\ \text{Lemma 3.3}\ \}
\end{aligned}$$

□

6.3 Invariants

Invariant 6.3. *Given a pending set in one of the wallet's checkpoints,*

$$\forall (ins_1, outs_1), (ins_2, outs_2) \in pending \text{ where } (ins_1, outs_1) \neq (ins_2, outs_2).\ ins_1 \cap ins_2 = \emptyset$$

Proof (sketch). By induction on the wallet construction. The cases for the empty wallet, applyBlock and newPending are straight-forward. The case for rollback is trickier. We have two pending sets *pending* and *pending*$'$, and we know that the invariant holds for both. Is it possible that there is a transaction t_1 in *pending* that spends the same input as a different transaction t_2 in *pending*$'$?

To answer this question, we must take into account how the wallet's checkpoints are created: applyBlock introduces a new checkpoint, possibly reducing the pending set, and then new pending transactions are inserted using newPending.

First, consider the case where $t_2 \in pending$ (i.e., t_2 was not removed by the call to applyBlock). In that case, we cannot have $t_1 \in pending$ due to the side condition to newPending.

In the case where $t_2 \notin pending$, this must be because t_2 has been included in the blockchain and hence applyBlock removed it from the pending set. But in this case, applyBlock will also have removed all of t_2's inputs from the UTxO, and hence it's not possible that the new transaction t_1 used any of these same inputs. □

6.4 Memory requirements

Obviously, storing all checkpoints of the UTxO leads to unbounded memory usage. Thankfully however the blockchain protocol defines a 'security parameter' k which guarantees that we will never have to roll back past k slots, and hence don't have to store more than k checkpoints. Currently, k is set to 2160; for a typical user, the UTxO and pending sets will not be large and keeping track of the last 2160 values will not be a huge deal.

We can also give a more precise upper bound on the memory requirements. Instead of storing k UTxO checkpoints, it would also suffice to store the UTxO as it was k slots ago, and store all k blocks since the last checkpoint. Since a block has a maximum size of 2 MB, this means we need to store at most a little over 4 GB of data.

As far as the pending transactions go, with the conservative estimate of 100 pending transactions per slot, it'd be a maximum 216,000 transactions (plus some administrative overhead). At a maximum transaction size of 64 kB, this

adds an additional 13.5 GB; however, at more typical values of 3 pending transactions this reduces to 405 MB, and with a more typical average transaction size of 4 kB, to a mere 25 MB.

Since rollbacks are relatively rare (especially having to roll back far), it would be fine to store this information on disk rather than in memory, and hence these memory requirements are no big deal at all. Probably the best engineering trade-off will be to store a few checkpoints in memory and the rest on disk.

6.5 Switching to a fork

Although rollback is a useful primitive operation on the wallet state, in practice the wallet will never ever actually rollback, but rather switch to a different fork. The disambiguation rule in the underlying blockchain protocol (either Ouroboros or Ouroboros Praos) states that this can only happen if that other fork is *longer* than the current one.

It will therefore be useful to provide a higher-level operation that combines rolling back with applying the blocks in the new fork:

$$\mathsf{switch}\ n\ \mathit{blocks} = \mathsf{applyBlocks}\ \mathit{blocks} \circ \mathsf{rollbacks}\ n \tag{6.3.1}$$

where rollbacks n calls rollback n times, and applyBlocks calls applyBlock for all blocks in order. Such an operation is important because it means that the intermediate state of the wallet during the switch is not visible to the user.

Implementation Note. The current Cardano API does not provide something equivalent to switch, instead providing only hooks that correspond to applyBlocks and rollbacks. The wallet kernel however could batch up the calls to rollbacks, and not apply the n rollbacks until it has at least $n + m$ ($m \geq 1$) blocks to apply. This solution is not ideal, as it is unclear what value to set m to; probably the only workable solution is to set a time bound. However, since all these applyBlocks will come very close together (*maybe* even as a single call), in practice this can probably work reasonably well.

6.6 Omitting checkpoints

Not all wallets are created at the start of the blockchain, of course.

Lemma 6.4. A wallet created after n blocks have already been created will have checkpoints

$$\underbrace{(\emptyset,\emptyset) : \cdots : (\emptyset,\emptyset)}_{n \text{ empty checkpoints}} : \underbrace{(\emptyset,\emptyset)}_{\text{initial checkpoint}}$$

Proof (sketch). We can assume that there have been no calls to `newPending`. Each call to `applyBlock` will create a new empty checkpoint, because `updateUTxO` will not consider any of the addresses in those blocks to be "`ours`" (and there are no pending transactions to remove for `updatePending`). Similarly, if there have been previous rollbacks, those would simply have stripped off empty checkpoints. □

Lemma 6.4 gives us a very useful optimisation opportunity: we can simply leave all the empty checkpoints as implicit, adding a single case to rollback:

$$\texttt{rollback}\ [(\emptyset,\emptyset)] = [(\emptyset,\emptyset)]$$

Perhaps this seems like a trivial fact, but it's not vacuous: it does put constraints on what we can and cannot record in checkpoints. Looking ahead, we have to make sure that when we add in the expected UTxO (Figure 8) and block metadata (Figure 11), or when we choose a concrete instantiation of the block metadata (Section 9.2.2), we don't break this property. Fortunately, the expected UTxO is subset of the UTxO and must therefore also be empty, and the block metadata only records information about transactions that affect this wallet and about addresses that the wallet owns.

7 Minimum Balance

In the basic model we have

Lemma 7.1 (Bounds on totalBalance in basic model)**.**

$$\mathsf{availableBalance}\ (\mathit{utxo}, \mathit{pending}) \leq \mathsf{totalBalance}\ (\mathit{utxo}, \mathit{pending}) \leq \mathsf{balance}\ \mathit{utxo}$$

Proof (sketch). The lower bound is trivial. The upper bound follows from Invariant 3.4 ($\mathsf{txins}\ \mathit{pending} \subseteq \mathrm{dom}\ \mathit{utxo}$). □

However, Invariant 3.4 no longer holds in the presence of rollbacks. Suppose an incoming transaction t_1 transfers a large sum to the wallet, and the wallet subsequently creates a pending transaction t_2 that transfers a small percentage of that sum to another address. If we now roll back, transaction t_2 will be spending an input that isn't (yet) in the wallet's UTxO. The change from that transaction will consequently *increase* the wallet's UTxO. While not incorrect (after all, t_2 can only be confirmed if t_1 is too), it is of course rather strange for *change* to increase one's balance. In fact, totalBalance really only makes sense when the pending transactions only spent outputs from the UTxO:

Definition 7.2 (Precondition totalBalance)**.**

$$\begin{aligned}&\mathsf{totalBalance}\ (\mathit{utxo}, \mathit{pending})\\ &\quad \textbf{requires}\ \mathsf{txins}\ \mathit{pending} \subseteq \mathrm{dom}\ \mathit{utxo}\end{aligned}$$

We will refer to incoming transactions that have been rolled back as *expected transactions*. In other words, expected transactions are transactions (such as t_1 in the example above) that we expect to be included in the blockchain, but haven't yet. We will refer to the corresponding unspent outputs as the *expected UTxO*.

Note. It is of course possible that such missing transactions (t_1) *never* make it into the new fork, in which case dependent pending transaction (t_2) should eventually be removed from *pending*. This problem may arise even without rollbacks, however, and cannot be solved until we introduce a TTL value for transactions (Section 10.4).

In this section we will see how by keeping track of the expected UTxO we can give a clearer picture of the wallet's balance, even in the presence of rollbacks. The extended model is shown in Figure 8. When the wallet rolls back, the unspent outputs that are removed from the *utxo* get added to *expected*. Conversely, when the wallet applies a block, any confirmed outputs are removed from *expected*. Put another way, during rollback anything that is removed from the actual UTxO gets added to the expected UTxO, and when applying a block anything that is added to the actual UTxO gets removed from the expected UTxO.

7.1 Properties

A trivial fact we need later is that

Lemma 7.3. $\mathsf{updateExpected}\ b\ \mathit{expected} \subseteq \mathit{expected}$

When we roll back a block but remember the UTxO we used to have, we can, in a sense, 'anticipate' what we think the future might look like; in other words, we add the new outputs in that block to our expected UTxO (`new` was defined in Section 3.1):

Definition 7.4 (Anticipate a block)**.**

$$\mathsf{anticipate}\ b\ ((\mathit{utxo}, \mathit{pending}, \mathit{expected}) : \mathit{checkpoints}) = ((\mathit{utxo}, \mathit{pending}, \mathit{expected} \cup \mathsf{new}\ b) : \mathit{checkpoints})$$

We can now formalise the above intuition, providing the moral equivalent of Lemma 6.1 in the last section, where we didn't track the expected UTxO yet.

Lemma 7.5. $\mathsf{rollback} \circ \mathsf{applyBlock}\ b = \mathsf{anticipate}\ b$

Wallet state

$$checkpoints \in [\mathsf{UTxO} \times \mathsf{Pending} \times \mathsf{UTxO}]$$
$$w_\emptyset \in \mathsf{Wallet} = [(\emptyset, \emptyset, \emptyset)]$$

Atomic updates

$$\begin{aligned}
&\mathsf{applyBlock}\ b\ ((utxo, pending, expected) : checkpoints) = \\
&\quad (\mathsf{updateUTxO}\ b\ utxo, \mathsf{updatePending}\ b\ pending, \mathsf{updateExpected}\ b\ expected) \\
&\quad : (utxo, pending, expected) : checkpoints \\
&\mathsf{newPending}\ tx\ ((utxo, pending, expected) : checkpoints) = \\
&\quad (utxo, pending \cup \{tx\}, expected) : checkpoints \\
&\mathsf{rollback}\ ((utxo, pending, expected) : (utxo', pending', expected') : checkpoints) = \\
&\quad (utxo', pending \cup pending', expected \cup expected' \cup (\mathrm{dom}\, utxo' \mathbin{\not\lhd} utxo)) : checkpoints
\end{aligned}$$

Auxiliary

$$\mathsf{updateExpected}\ b\ expected = \mathrm{dom}(\mathsf{new}\ b) \mathbin{\not\lhd} expected$$

Figure 8: Model with rollback and expected UTxO

Proof.

$$\begin{aligned}
&\mathsf{rollback}\ \Big(\mathsf{applyBlock}\ b\ ((u, p, e) : cs)\Big) \\
&= \mathsf{rollback}\ ((\mathsf{updateUTxO}\ b\ u, \mathsf{updatePending}\ b\ p, \mathsf{updateExpected}\ b\ e) : (u, p, e) : cs) \\
&= ((u, (\mathsf{updatePending}\ b\ p) \cup p, (\mathsf{updateExpected}\ b\ e) \cup e \cup (\mathrm{dom}\, u \mathbin{\not\lhd} \mathsf{updateUTxO}\ b\ u)) : cs) \\
&= ((u, p, e \cup (\mathrm{dom}\, u \mathbin{\not\lhd} \mathsf{updateUTxO}\ b\ u)) : cs) && \{\text{ Lemma 7.3 }\} \\
&= ((u, p, e \cup \mathsf{new}\ b) : cs) = \mathsf{anticipate}\ b\ ((u, p, e) : cs) && \{\text{ Lemma 3.2 }\}
\end{aligned}$$

□

7.2 Invariants

We should have the invariant that the expected UTxO and actual UTxO are always disjoint:

Invariant 7.6. *For each checkpoint* $(utxo, pending, expected)$ *in a wallet* w*,*

$$\mathrm{dom}\, utxo \cap \mathrm{dom}\, expected = \emptyset$$

Like the actual UTxO, the expected UTxO belongs to the wallet

Invariant 7.7. *For each checkpoint* $(utxo, pending, expected)$ *in a wallet* w*,*

$$\mathrm{range}\, expected \subseteq \mathsf{TxOut}_{\mathsf{ours}}$$

This is the equivalent of Invariant 3.5, and follows straight-forwardly from it because *expected* is derived from *utxo*. Finally, we have to weaken Invariant 3.4 to:

Invariant 7.8.

$$\mathsf{txins}\ pending \subseteq \mathrm{dom}(utxo \cup expected)$$

(Note that *expected* may include change terms from previously confirmed pending transactions.)

7.3 Minimum balance

In the basic model, when the wallet submits a bunch of pending transactions, it expects all of those transactions to eventually be included in the blockchain. Of course, there is no guarantee that this will happen: maybe some will, maybe none will, or maybe all will be included. The situation is more complicated in the model with rollback. When a pending transaction (possibly transitively) depends on an expected output, it can never happen that that pending transaction is confirmed but the expected incoming transaction is not. Let's refer to these subsets of confirmed pending and expected transactions as *possible futures*. Then a reasonable question we might ask is: what is the wallet's *minimum balance across all possible futures*?[2]

Note. We assume that there is no other instance of the wallet 'out there', so that this wallet is the only one to transfer funds from the wallet to other accounts. In other words, we assume that expected transactions can only *increase* the wallet's balance (and pending transactions can only *decrease* it). If this assumption is not satisfied, the concept of minimum balance becomes meaningless, since there is then always a possible future in which the 'other wallet' spends all of the wallet's funds. For somewhat similar reasons, the wallet's *maximum* balance is not a particular interesting notion; if we stipulate 'assuming no incoming transactions' then the maximum balance is simply the current balance, and if we don't make such a stipulation, then the balance of the wallet is unbounded (or bounded by the cryptocurrency's cap).

Since we only keep track of the expected UTxO, and therefore lack dependency information about expected transactions, we cannot determine if two expected transactions can both be confirmed in the blockchain. We will therefore conservatively[3] estimate the minimum balance as

Definition 7.9 (Minimum balance)**.**

$$\mathsf{minimumBalance}\ (utxo, pending, expected) = \min_{\substack{e \sqsubseteq expected \\ p \subseteq pending \\ \mathsf{txins}\ p \subseteq \mathrm{dom}(utxo \cup e)}} \mathsf{totalBalance}\ (utxo \cup e, p)$$

Lemma 7.10.

$$\mathsf{minimumBalance}\ (utxo, pending, expected) \leq \mathsf{totalBalance}\ (utxo \cup expected, pending)$$

Proof.

$$\begin{aligned}
&\mathsf{minimumBalance}\ (utxo, pending, expected) \\
&= \min_{\substack{e \sqsubseteq expected \\ p \subseteq pending \\ \mathsf{txins}\ p \subseteq \mathrm{dom}(utxo \cup e)}} \mathsf{totalBalance}\ (utxo \cup e, p) \\
&= \min\Big\{ \ldots, \mathsf{totalBalance}\ (utxo \cup expected, pending), \ldots \Big\} && \{\ \text{Invariant 7.8}\ \} \\
&\leq \mathsf{totalBalance}\ (utxo \cup expected, pending)
\end{aligned}$$

□

7.4 Bounds on totalBalance

Now that we keep track of the expected UTxO, we can state the equivalent of Lemma 7.1 for the wallet with rollback:

Lemma 7.11 (Bounds on totalBalance in the presence of rollback)**.**

$$\begin{aligned}
&\mathsf{availableBalance}\ (utxo, pending) \\
\leq\ &\mathsf{minimumBalance}\ (utxo, pending, expected) \\
\leq\ &\mathsf{totalBalance}\ (utxo \cup expected, pending) \\
\leq\ &\mathsf{balance}\ (utxo \cup expected)
\end{aligned}$$

[2] This is somewhat akin to the financial concept of 'value at risk'.

[3] Note that the absence of some dependency information only gives us *more* freedom in picking the elements of this set; thus, missing dependency information may make the lower bound that we establish less accurate, but it will still be a lower bound.

Proof. The first inequality is trivial. The second was proven in Lemma 7.10. The final one comes from Invariant 7.8. □

While in addition we trivially have that

$$\mathsf{totalBalance}\ (utxo, pending) \leq \mathsf{totalBalance}\ (utxo \cup expected, pending)$$

as discussed at the start of Section 7, that expression $\mathsf{totalBalance}\ (utxo, pending)$ is not particularly meaningful when $\mathsf{txins}\ pending \not\subseteq \mathrm{dom}\ utxo$. Having said that, when we *do* have that inclusion, minimum balance and total balance coincide:

Lemma 7.12. If $\mathsf{txins}\ pending \subseteq utxo$,

$$\mathsf{minimumBalance}\ (utxo, pending, expected) = \mathsf{totalBalance}\ (utxo, pending)$$

Intuitively this makes sense: if the pending transactions only spent outputs from the UTxO, the balance is minimised when all pending transactions get confirmed and none of the expected ones.

7.5 Expected UTxO *versus* expected transactions

In the wallet as specified in this section we keep track of the expected UTxO, rather than the expected transactions themselves. At first glance it may seem that keeping track of the expected transactions would have some benefits:

- We would know the dependencies of the expected transactions.
- The wallet could resubmit those (already signed) expected transactions to be included in the blockchain, in order to make sure that transactions that transfer large sums to the wallet will be included in the new fork after a rollback.

However, even if we did know the direct dependencies of expected transactions, we would still not be able to accurately tell if two expected transactions might both be included in the blockchain. It is entirely possible that two expected transactions t_1 and t_2 have different inputs t_1' and t_2', but those dependent transactions t_1' and t_2' share some inputs. Worse, t_1' and t_2' might have nothing to do with the wallet (neither transfer funds from nor transfer funds to the wallet). In the worst case we would need the entire blockchain to accurately determine if two transactions share transitive dependencies. (Not to mention that computing an accurate minimum balance would be an expensive computation.)

Similarly, since expected transaction may depend on the *entire* blockchain up until this point, and that *entire* blockchain might have been rolled back, in order to be able to resubmit expected transactions the wallet would have to keep track of the entire blockchain. Even if we take into account the security parameter k, it would mean that in the worst case the wallet would have to keep track of the last k blocks *in their entirety* (as opposed to just the inputs from and outputs to addresses owned by the wallet).

8 Efficiency of minimumBalance

In general we can have pending transactions dependent on each other, expected transactions dependent on each other, as well as pending transactions depending on expected transactions and vice versa. Figure 9 shows some of the possibilities, and computes in which possible future the wallet's balance is minimum.

Single independent transaction

$(t_1 : c_1)$

$$\Delta = \begin{cases} 0 & \overline{t_1} \\ c_1 & t_1 \end{cases}$$

include if $c_1 \leq 0$

One transaction dependent on one other (picking assuming $c_1 \leq 0$)

$(t_1 : c_1)$ | $(t_2 : c_2)$

$$\Delta = \begin{cases} 0 & \overline{t_1}\,\overline{t_2} \\ c_2 & \overline{t_1}\,t_2 \\ c_2 + c_1 & t_2\,t_2 \end{cases}$$

c_2		pick
< 0		$t_1\,t_2$
> 0	$c_1 + c_2 \leq 0$	$t_1\,t_2$
	otherwise	$\overline{t_1}\,\overline{t_2}$

Linear chain (picking assuming $c_1 \leq 0$)

$(t_1 : c_1)$ | $(t_2 : c_2)$ | $(t_3 : c_3)$

$$\Delta = \begin{cases} 0 & \overline{t_1}\,\overline{t_2}\,\overline{t_3} \\ c_3 & \overline{t_1}\,\overline{t_2}\,t_3 \\ c_3 + c_2 & \overline{t_1}\,t_2\,t_3 \\ c_3 + c_2 + c_1 & t_1\,t_2\,t_3 \end{cases}$$

c_2	c_3		pick
< 0	< 0		$t_1 t_2 t_3$
< 0	> 0	$(c_1 + c_2) + c_3 \leq 0$	$t_1 t_2 t_3$
		otherwise	$\overline{t_1 t_2 t_3}$
> 0	< 0	$c_1 + c_2 \leq 0$	$t_1 t_2 t_3$
		otherwise	$\overline{t_1 t_2 t_3}$
> 0	> 0	$c_1 + (c_2 + c_3) \leq 0$	$t_1 t_2 t_3$
		otherwise	$\overline{t_1 t_2 t_3}$

One transaction dependent on two others (picking assuming $c_1 \leq 0$)

$(t_1 : c_1)$ — $(t_2 : c_2)$, $(t_3 : c_3)$

$$\Delta = \begin{cases} 0 & \overline{t_1}\,\overline{t_2}\,\overline{t_3} \\ c_3 & \overline{t_1}\,\overline{t_2}\,t_3 \\ c_2 & \overline{t_1}\,t_2\,\overline{t_3} \\ c_3 + c_2 & \overline{t_1}\,t_2\,t_3 \\ c_3 + c_2 + c_1 & t_1\,t_2\,t_3 \end{cases}$$

c_2	c_3		pick
< 0	< 0		$t_1\,t_2\,t_3$
< 0	> 0	$c_1 + c_3 \leq 0$	$t_1\,t_2\,t_3$
		otherwise	$\overline{t_1}\,t_2\,\overline{t_3}$
> 0	< 0	$c_1 + c_2 \leq 0$	$t_1\,t_2\,t_3$
		otherwise	$\overline{t_1}\,\overline{t_2}\,t_3$
> 0	> 0	$c_1 + (c_2 + c_3) \leq 0$	$t_1\,t_2\,t_3$
		otherwise	$\overline{t_1}\,\overline{t_2}\,\overline{t_3}$

Two transactions depending on a single other (picking assuming $c_1 \leq 0, c_2 \leq 0$)

$(t_1 : c_1)$, $(t_2 : c_2)$ — $(t_3 : c_3)$

$$\Delta = \begin{cases} 0 & \overline{t_1 t_2 t_3} \\ c_3 & \overline{t_1 t_2} t_3 \\ c_3 + c_2 & \overline{t_1} t_2 t_3 \\ c_3 + c_1 & t_1 \overline{t_2} t_3 \\ c_3 + c_2 + c_1 & t_1 t_2 t_3 \end{cases}$$

c_3		pick
< 0		$t_1 t_2 t_3$
> 0	$(c_1 + c_2) + c_3 \leq 0$	$t_1 t_2 t_3$
	otherwise	$\overline{t_1 t_2 t_3}$

Common ancestor

$(t_1 : c_1)$ — $(t_2 : c_2)$, $(t_3 : c_3)$ — $(t_4 : c_4)$

$$\Delta = \begin{cases} 0 & \overline{t_1 t_2 t_3 t_4} \\ c_4 & \overline{t_1 t_2 t_3} t_4 \\ c_4 + c_3 & \overline{t_1 t_2} t_3 t_4 \\ c_4 + c_2 & \overline{t_1} t_2 \overline{t_3} t_4 \\ c_4 + c_3 + c_2 & \overline{t_1} t_2 t_3 t_4 \\ c_4 + c_3 + c_2 + c_1 & t_1 t_2 t_3 t_4 \end{cases}$$

Direct and indirect dependency

$(t_1 : c_1)$ — $(t_2 : c_2)$, $(t_3 : c_3)$, $(t_1 : c_1)$

$$\Delta = \begin{cases} \overline{t_1 t_2 t_3} \\ \overline{t_1 t_2} t_3 \\ \overline{t_1} t_2 t_3 \\ t_1 t_2 t_3 \end{cases}$$

Figure 9: Some possible dependency graphs between transactions

8.1 Computing the minimum balance

As discussed, however, we don't have accurate dependency information available for expected transactions. In the definition of minimumBalance we therefore range over all possible selections of expected transactions. The goal of an algorithm to compute the minimum balance then is to pick a set of expected and pending transactions that minimises the wallet's balance, such that known dependencies of the pending transactions are all satisfied. Clearly, we want to pick as many pending transactions as possible (since they reduce the balance), but pick expected transactions only when the decrease in balance from the dependent pending transactions makes up for the increase in balance from the expected transactions.

Lemma 8.1. Given an expected UTxO e, computing

$$\min_{\substack{p \subseteq pending \\ \mathsf{txins}\ p \subseteq \mathrm{dom}(utxo \cup e)}} \mathsf{totalBalance}\ (utxo \cup e, p)$$

can be done in $\mathcal{O}\left(|pending|\right)$ time.

Proof (sketch). The set of pending transactions forms a DAG, which we can traverse in topological order (itself a well-known linear operation), keeping track of the transactions we selected so far. For each pending transaction we simply check if all its dependencies have been selected; if so, we include it. There is no need to backtrack on any of these decisions:

- Including a pending transaction is always a good thing (decreases the balance and can only increase the possibilities for including further transactions)
- If we cannot include the transaction because some of its dependencies are missing, then any pending transactions we will encounter later in the topological order will not change this.

□

This would suggest that the complexity of minimumBalance is exponential in $|expected|$ and linear in $|pending|$, but we can do a bit better.

Lemma 8.2 (Grouping expected transactions)**.** Group the expected transactions such that two expected transactions are in the same group iff their sets of dependent pending transactions overlap. Then when we compute the minimum balance we can consider each group of expected transactions separately.

Lemma 8.3. The complexity of minimumBalance is given by $\mathcal{O}\left(e \cdot (2^g + p)\right)$ with e the number of groups of expected transactions (Lemma 8.2), g the size of the largest such group, and p the number of pending transactions.

If g is very large, we can always fall back on a more conservative estimate. For instance, since it is always sound to forget some dependencies, we can forget any dependencies from pending transactions on expected ones. This means that we can always use Lemma 7.12 to approximate minimumBalance using totalBalance.

Note. Expected transactions that coexisted in the blockchain at some point are obviously compatible with each other. However, keeping track of this information would not help us here: we already assume that *all* expected transactions are compatible with each other. Knowing when expected transactions are *not* compatible with each other would allow us to establish more tighter bounds on the minimum balance, but presence or absence of transactions in particular forks is not sufficient to conclude incompatibility.

8.2 Further efficiency improvements

In Section 8 we point out that we can group the expected transactions such that two expected transactions are in the same group if and only if their dependent pending transactions overlap. Then decisions in one group clearly cannot affect decisions in another, so that we reduce the complexity of finding the minimum balance from exponential in the number of expected transactions (which may be large) to exponential in the size of the largest group (which will be much smaller).

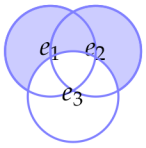
In this section we make an observation that can reduce this complexity further:

Lemma 8.4. Whenever we discover a set of expected transactions $\mathcal{E}$ such that $\mathcal{E}$ together with the set of pending transactions $\mathcal{P}$ that depend *only* on expected transactions in $\mathcal{E}$ (or on other pending transactions in $\mathcal{P}$), then this set $\mathcal{E}$ will be a subset of the set of expected transactions in the possible future with the minimum balance. In other words, it will not be necessary to backtrack on any of the decisions in $\mathcal{E}$.

Proof (sketch). If $\mathcal{E}$ is a singleton set $\{e_1\}$, then it's obvious that whatever we discover later, including e_1 cannot increase the minimum balance we find.

So, let's consider the smallest more interesting example: let's suppose we find a set $\{e_1, e_2\}$ of two expected transactions such that these two expected transactions, together with the pending transactions that depend *only* on those two transactions, decrease our balance. Furthermore, let's assume that there is at least one other expected transaction that we may wish to include. Let's denote this visually as

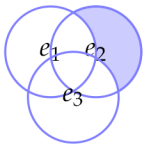
$$(e_1 + e_2) - \quad <0 \tag{8.4.1}$$



where the Venn diagram depicts all the pending transactions.

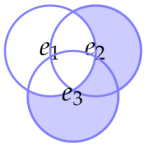
Let's also assume that including either e_1 or e_2 by themselves is *not* sufficient to decrease the balance (otherwise we're back at the trivial case):

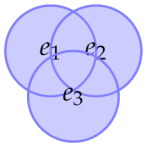
$$e_2 - \quad >0 \tag{8.4.2}$$



Now the question becomes: might it be the case that we might have to backtrack on the decision to include e_1? If so, that would mean that picking $\{e_2, e_3\}$ (without e_1) must be a better choice than picking *all* of e_1, e_2, e_3:

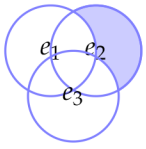
$$(e_2 + e_3) - \quad < (e_1 + e_2 + e_3) - \tag{8.4.3}$$

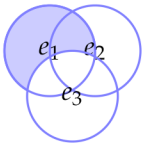




From (8.4.1) we get

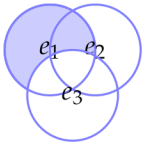
$$e_2 - \quad < -e_1 + \tag{8.4.4}$$

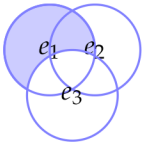




Combining that with (8.4.2) we get

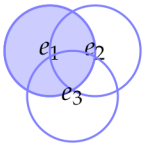
$$0 < -e_1 + \quad \text{i.e.} \quad e_1 - \quad <0 \tag{8.4.5}$$





Finally, from (8.4.3)

$$0 < e_1 - \tag{8.4.6}$$



but that contradicts (8.4.5). □

8.3 Balance caching and prefiltering

Balance caching (Section 4) and prefiltering (Section 5) translate easily to the wallet with rollback and expected UTxO; the full wallet model is shown in Figure 10. The only slightly subtle point is that during prefiltering we need to take the expected UTxO into account as well as the actual UTxO.

Wallet state

$$\begin{aligned} &\mathit{checkpoints} \in [\mathsf{UTxO} \times \mathsf{Pending} \times \mathsf{UTxO} \times \mathsf{Coin}] \\ &w_\emptyset \in \mathsf{Wallet} = [(\emptyset, \emptyset, \emptyset, 0)] \end{aligned}$$

Atomic updates

$$\begin{aligned} &\mathsf{applyBlock}\ b = \mathsf{applyBlock}'\left(\mathsf{txins}\ b \cap \mathrm{dom}(\mathit{utxo} \cup \mathit{expected} \cup \mathit{utxo}^+), \mathit{utxo}^+\right) \\ &\qquad \text{where } \mathit{utxo}^+ = \mathsf{txouts}\ b \rhd \mathsf{TxOut}_{\mathsf{ours}} \\ &\mathsf{newPending}\ \mathit{tx}\ ((\mathit{utxo}, \mathit{pending}, \mathit{expected}, \sigma) : \mathit{checkpoints}) = \\ &\qquad (\mathit{utxo}, \mathit{pending} \cup \{\mathit{tx}\}, \mathit{expected}, \sigma) : \mathit{checkpoints} \\ &\mathsf{rollback}\ ((\mathit{utxo}, \mathit{pending}, \mathit{expected}, \sigma) : (\mathit{utxo}', \mathit{pending}', \mathit{expected}', \sigma') : \mathit{checkpoints}) = \\ &\qquad (\mathit{utxo}', \mathit{pending} \cup \mathit{pending}', \mathit{expected} \cup \mathit{expected}' \cup (\mathrm{dom}\ \mathit{utxo}' \not\lhd \mathit{utxo}), \sigma') : \mathit{checkpoints} \end{aligned}$$

Auxiliary

$$\begin{aligned} &\mathsf{applyBlock}'\ (\mathit{txins}_b, \mathit{txouts}_b)\ (\mathit{utxo}, \mathit{pending}, \mathit{expected}, \sigma) : \mathit{checkpoints} = \\ &\qquad (\mathit{utxo}', \mathit{pending}', \mathit{expected}', \sigma') : (\mathit{utxo}, \mathit{pending}, \mathit{expected}, \sigma) : \mathit{checkpoints} \\ &\qquad \text{where} \quad \begin{aligned} \mathit{pending}' &= \{\mathit{tx} \mid \mathit{tx} \in \mathit{pending}, (\mathit{inputs}, \_) = \mathit{tx}, \mathit{inputs} \cap \mathit{txins}_b = \emptyset\} \\ \mathit{expected}' &= \mathrm{dom}\ \mathit{txouts}_b \not\lhd \mathit{expected} \\ \mathit{utxo}^+ &= \mathit{txouts}_b \\ \mathit{utxo}^- &= \mathit{txins}_b \lhd (\mathit{utxo} \cup \mathit{utxo}^+) \\ \mathit{utxo}' &= \mathit{txins}_b \not\lhd (\mathit{utxo} \cup \mathit{utxo}^+) \\ \sigma' &= \sigma + \mathsf{balance}\ \mathit{utxo}^+ - \mathsf{balance}\ \mathit{utxo}^- \end{aligned} \end{aligned}$$

Figure 10: Full wallet model

Parameters

$$
\begin{array}{lll}
\mathsf{TxMeta} & & \text{meta information about transactions} \\
(\mathsf{BlockMeta}, \uplus) & & \text{meta information about blocks (monoid)} \\
\mathsf{txMeta} & :: \mathsf{Tx} \to \mathsf{TxMeta} & \\
\mathsf{blockMeta} & :: \mathbb{P}(\mathsf{Tx}) \to \mathsf{BlockMeta} &
\end{array}
$$

Wallet state

$$
\begin{aligned}
\mathsf{TxInfo} &= \mathsf{TxId} \mapsto \mathsf{TxMeta} \\
\mathsf{Wallet} &= [\mathsf{UTxO} \times \mathsf{Pending} \times \mathsf{BlockMeta}] \times \mathsf{TxInfo}
\end{aligned}
$$

Atomic updates

$$
\begin{aligned}
&\mathsf{applyBlock}\ b\ ((utxo, pending, blockMeta) : checkpoints, txInfo) = (\\
&\quad (\mathsf{updateUTxO}\ b\ utxo, \mathsf{updatePending}\ b\ pending, blockMeta \uplus \mathsf{blockMeta}\ b) \\
&\quad : (utxo, pending, blockMeta) : checkpoints,\ txInfo \cup \{\mathsf{txid}\ tx \mapsto \mathsf{txMeta}\ tx \mid tx \in b\}) \\
&\mathsf{newPending}\ tx\ ((utxo, pending, blockMeta) : checkpoints, txInfo) = \\
&\quad (utxo, pending \cup \{tx\}, blockMeta) : checkpoints, txInfo \cup \{\mathsf{txid}\ tx \mapsto \mathsf{txMeta}\ tx\}) \\
&\mathsf{rollback}\ ((utxo, pending, blockMeta) : (utxo', pending', blockMeta') : checkpoints, txInfo) = \\
&\quad ((utxo', pending \cup pending', blockMeta') : checkpoints, txInfo)
\end{aligned}
$$

Figure 11: Tracking metadata

9 Tracking Metadata

A real wallet implementation may need to store more information than we have modelled in the specification so far. For instance, users may wish to know *when* their pending transactions got confirmed in the blockchain (as opposed to merely *that* they were confirmed), or what the effect was of a particular transaction on their balance (rather than merely being able to see the current balance).

In Section 9.1 we first study this kind of *metadata* from an abstract point of view; here the main question we want to answer is how this metadata relates to the state of the wallet, and in particular to rollbacks. Then in Section 9.2 we will see how we can instantiate the abstract model from Section 9.1 to track the metadata required by the actual Cardano (V1) wallet.

9.1 Abstract model

Our abstract model of tracking metadata is shown in Figure 11. We distinguish between *block* metadata, BlockMeta, and *transaction* metadata, TxMeta. The key idea is that the block metadata depends on the state of the blockchain, but the transaction metadata does not. Specifically:

- Transaction metadata TxMeta is assumed to be stateless; once we have seen a transaction we can compute its metadata and this will never change. Consequently, rollbacks don't change the transaction metadata that the wallet records.
- Block metadata BlockMeta is assumed derivable from a block; on rollback we simply revert to the previous value of the block metadata.

Note on prefiltering. The model we show in Figure 11 does not implement prefiltering (Section 5). In order to make metadata tracking work well with prefiltering it suffices to make sure that blockMeta can take a prefiltered block as

argument. In an actual implementation, this means that if blockMeta needs any additional information other than the transactions, the prefiltering function needs to ensure that this information is included in the prefiltered block.

9.2 Transaction history

This section is a bit more technical in nature and studies how the transaction metadata reported in the current wallet 'V1 REST API' can be defined in terms of the abstract model from Figure 11.

The transaction metadata reported in the REST API divides into three categories: information that we can model as part of TxMeta, information that we can model as part of BlockMeta, and information that is derived from the state of the wallet proper.

9.2.1 Static information

The static information (TxMeta) is the most straight-forward. This includes

- transaction ID
- total amount
- inputs and outputs (both in terms of addresses)
- the transaction creation time (as a timestamp in microseconds, not as a slot number)
- whether or not the transaction is *local* (*all* input and output addresses are owned by the wallet)
- the transaction's *direction*: *incoming* if the transaction increases the wallet's balance, or *outgoing* otherwise[4]

9.2.2 Information dependent on chain status

The V1 API reports some information that is dependent on the chain status.

- How deep the transaction lives in the blockchain (counting from the tip); somewhat confusingly, it refers to this as the number of 'confirmations'.
- Whether or not an address has been *used*. An address *addr* is considered *used* if and only if
  1. ours *addr*
  2. There exists *at least one* confirmed transaction $(inputs, outputs)$ where $\exists c.(addr, c) \in outputs$
- Whether or not an address *is a change address*. Since this information is derived from the blockchain, it can only be approximated. This field is currently defined as follows: an address *addr* is considered a change address if and only if
  1. ours *addr*
  2. There exists *exactly one* confirmed transaction $(inputs, outputs)$ where $\exists c_{change}.(addr, c_{change}) \in outputs$
  3. There is at least one other output $(addr_{other}, c_{other}) \in outputs$ where $\neg(\mathsf{ours}\ addr_{other})$
  4. All *inputs* $i \in inputs$ refer to outputs $(addr_{in_i}, c_{in_i})$ where $\mathsf{ours}\ addr_{in_i}$

We can model this as follows: our block metadata contains the block number that each confirmed transactions got confirmed in, as well as a mapping from addresses to address metadata. The depth of a confirmation can be derived from the block number and the current slot number. The address metadata consisting of two booleans indicating whether the address is used and whether or not it is a change address.

$$\begin{aligned} \mathsf{BlockMeta} &= (\mathsf{TxId} \mapsto \mathsf{BlockNumber}) \times (\mathsf{Addr} \mapsto \mathsf{AddrMeta}) \\ \mathsf{AddrMeta} &= \mathsf{Bool} \times \mathsf{Bool} \end{aligned}$$

[4] This roughly matches the informal usage of 'incoming' and 'outgoing' elsewhere in this document.

For a single block this information is derived according to the definitions above. For the monoidal operator combining the block metadata from two different blocks, it suffices to take the union of the block numbers (since a transaction ID can only exist in one of the two blocks) and take the pointwise combination of the address metadata using

$$(isUsed, isChange) \uplus (isUsed', isChange') = (isUsed \vee isUsed', isChange \veebar isChange')$$

An address is used if its used in either of the two blocks, and is considered a change address if it's considered a change address in *one* of the two blocks, but not both (hence the use of exclusive or $\veebar$).

9.2.3 Transaction status

The API reports transaction status as one of[5]

`Applying` In terms of our model, this means that the transaction is in the pending set.

`InNewestBlocks` The transaction has been included in the blockchain, but may still be rolled back.

`Persisted` The transaction has been included in the blockchain, and can no longer be rolled back.[6]

`WontApply` The wallet has given up on trying to get a pending transaction into the blockchain; perhaps because the transaction has been invalidated, or perhaps simply because some time limit has expired.

For incoming transactions, only `InNewestBlocks` and `Persisted` are applicable. We can derive transaction status from the block metadata as described in Section 9.2.2 and the wallet's state as follows:

$$\begin{cases} \texttt{Applying} & tx \in pending \\ \begin{cases} \texttt{InNewestBlocks} & d \leq k \\ \texttt{Persisted} & d > k \end{cases} & \mathsf{txid}\ tx \mapsto n \in blockMeta\ (d \text{ derived block depth}) \\ \begin{cases} \texttt{WontApply} & outgoing \\ not\ shown & incoming \end{cases} & otherwise \end{cases}$$

The correct status when a transaction is in the wallet's pending set ($tx \in pending$) or the transaction is confirmed ($\mathsf{txid}\ tx \mapsto n \in blockMeta$) is obvious. The status for transactions that the wallet is aware of but are neither in the wallet's *pending* set, nor confirmed in the blockchain, is a bit more subtle.

- If the transaction is an *incoming* transaction (i.e., increases the wallet's balance) then it can never have been pending; the only way we might end up in this situation is when this transaction was included in a block that has since been rolled back. We don't report a 'rolled back' status for such transactions, but rather simply exclude from the wallet's history.
- If the transaction is an *outgoing* transaction (decreases the wallet's balance), it may be that it was pending at some point, but got removed from *pending* (without being included in the blockchain), perhaps because it was invalidated by another transaction (or the transaction submission layer gave up on it; see Section 10). We will report a `WontApply` status for such a transaction.
- There is however a second way that we might end up with such an outgoing transaction: it may have been created by *another instance* of the same wallet. Reporting such a transaction as `WontApply` is perhaps somewhat confusing, as it was never in the `Applying` state in *this* wallet. However, it *was* of course in `Applying` state in the other wallet, and reporting the transaction as `WontApply` also in this wallet has the benefit that both instances of the wallet will give the same status for this transaction.[7]

[5] The current wallet supports an additional status `Creating`, but we will not include this in the reimplementation.

[6] The current wallet may actually report a transaction as `Persisted` even before k blocks, depending on the wallet's 'assurance level'. This is of course misleading: such a transaction may in fact still be rolled back.

[7] If we wanted to extend the invariant that 'multiple instances of the same wallet all eventually converge to the same state'—which we don't currently prove—to also include this concrete transaction status, then this is the only possible choice. Moreover, the property 'has this transaction ever been pending' is not a property that we could infer when the recover a wallet's status from the block chain, and hence this property would not be 'stable'.

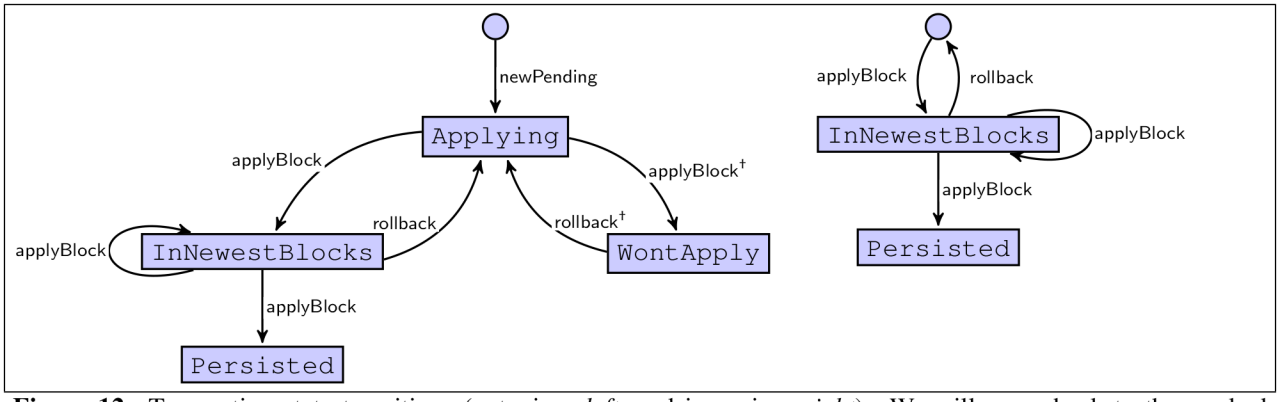


Figure 12: Transaction state transitions (outgoing, *left*, and incoming, *right*). We will come back to the marked transitions (†) in Section 10.1.

$$\begin{aligned} \mathsf{addPending} &:: \mathbb{P}(\mathsf{Tx}) \to \mathsf{Submission} \to \mathsf{Submission} \\ \mathsf{remPending} &:: \mathbb{P}(\mathsf{Tx}) \to \mathsf{Submission} \to \mathsf{Submission} \\ \mathsf{tick} &:: \mathsf{Submission} \to (\mathbb{P}(\mathsf{Tx}), \mathsf{Submission}) \end{aligned}$$

Figure 13: Transaction submission layer

The diagram in Figure 12 shows visually how the status of transactions can change over dynamically; the left diagram shows outgoing transactions, the right shows incoming transactions. The empty circle at the top of the diagram means that the transaction is not included in the wallet's reported history.

Although we do not include incoming transactions in the wallet's transaction history after they get rolled back, we do *not* remove their metadata from *txInfo*. After all, it can still be useful in order to be able to provide the wallet's user with information about the expected UTxO (a feature that the current wallet does not have).

10 Transaction Submission

An actual implementation of the wallet needs to broadcast pending transactions to the network, monitor when they get included in the blockchain, re-submit them if they don't get included, and perhaps eventually decide to give up on them if for some reason they do not get included.

This functionality does not need to be part of the wallet proper. In Section 10.1 we will discuss the interface to this component, and in Section 10.2 we will give a concrete simple implementation (which however still ignores any actual networking issues).

10.1 Interface

The interface to the transaction submission layer is shown in Figure 13. It is of a similar nature as interface to the wallet itself (Figure 2). Just like the wallet expects to be notified of events such as 'new block arrived' and 'user submitted a new transaction', the submission layer expects to be notified when the set of pending transactions grows or shrinks, and whenever a time slot has passed (more on that below).

It is the responsibility of the wallet to

- call addPending on newPending and (possibly) on rollback
- call remPending on applyBlock and cancel

In addition there must be a thread that periodically calls tick, to give the submission layer a chance to resubmit transactions that haven't made it into the blockchain yet. The set of transactions returned by tick are the transactions

that the submission layer gave up on (see below); the wallet should remove such transactions from its *pending* set. From the point of view of the wallet model this corresponds to a new function

$$\begin{aligned}
&\mathsf{cancel} :: \mathbb{P}(\mathsf{Tx}) \to \mathsf{Wallet} \to \mathsf{Wallet} \\
&\mathsf{cancel}\ \mathit{txs}\ (\mathit{checkpoints}, \mathit{txInfo}) = (\mathsf{map}\ \mathsf{cancel}'\ \mathit{checkpoints}, \mathit{txInfo}) \\
&\qquad \text{where } \mathsf{cancel}'\ (\mathit{utxo}, \mathit{pending}, \mathit{blockMeta}) = (\mathit{utxo}, \mathit{pending} \setminus \mathit{txs}, \mathit{blockMeta})
\end{aligned}$$

By the logic of Section 9.2.3, such a transaction would be reported as `WontApply`. Since we removed the transaction from the *pending* set in all checkpoints[8], however, a rollback won't reintroduce it into *pending*; if the user wants to explicitly tell the wallet to try this transaction again they will need to call newPending. Effectively, cancel becomes a secondary way in which a transaction may go from `Applying` to `WontApply` (arrow marked $\mathsf{applyBlock}^\dagger$), and newPending a secondary way to get back from `WontApply` to `Applying` (arrow marked $\mathsf{rollback}^\dagger$).

10.2 Implementation

Figure 14 shows a simple implementation of the submission layer. Part of the goal of this section is to show that the submission layer has sufficient information and does not need further support from the core wallet layer—indeed, does not need to know it exists at all.

The state of the submission layer consists of a mirror copy of the pending set of the wallet, as well as a schedule of which transactions to (re)submit next. The schedule is modelled as a simple list of time slots, recording for each slot the transactions that should be submitted, along with a submission count for each transaction.

- When the submission layer is notified of new pending transactions, it adds those to its *pending* set and schedules them to be submitted in the next slot, recording an initial submission count of 0.
- When the wallet tells the submission layer that some transactions are no longer pending (because they have been confirmed, because they have become invalid, or for other reasons), the submission layer simply removes them from its local *pending* set.
- The submission layer is parameterised over a 'resubmission function' ϱ. At the start of each time slot, the submission layer calls ϱ to resubmit the set of transactions that are due, possibly dropping some transactions that have reached a maximum submission count.

Although our model here does not deal with actual networking concerns, a typical side-effectful implementation of ϱ would

- Drop any transactions that have reached their maximum submission count, possibly notifying the user (note that ϱ only gets called for transactions that are still listed as pending).
- Resubmit the remaining transactions to the network, and reschedule them for the next attempt later. If desired, the submission count can be used to implement exponential back-off.

The concept of time slots is essentially private to the submission layer; it can, but does not have to, line up with the underlying blockchain slot length (indeed, we don't need to assume that the underlying blockchain even *has* a slot length).

In principle *pending* could be dropped from the submission layer; the reason that we don't is that this would mean that remPending would have to traverse the entire schedule to remove the transactions from each slot. By keeping a separate *pending* set we avoid this traversal, only checking the pending set at the point where we need it. The wallet's *pending* set and the submission layer's one don't need to be in perfect sync:

- If $t \in \mathit{pending}_{\mathsf{Wallet}}$ but $t \notin \mathit{pending}_{\mathsf{Submission}}$, it might mean that the wallet hasn't informed the submission layer yet of a new transaction, and it will just be submitted a little bit later, or it might mean that the submission layer removed a transaction from its pending set because it's given up on it, but the wallet hasn't reacted to the notification from the submission layer yet.[9]

[8] If the overhead of traversing all checkpoints is too large, an alternative implementation strategy would be to maintain an explicit *cancelled* set of transaction as part of the wallet's state.

[9] We could alternatively insist that the submission layer doesn't remove any transactions from its pending set until the wallet tells it so. Relaxing that restriction however allows us to state an invariant that anything in the submission layer's pending set must also be scheduled.

Types

$$\mathit{schedule} \in \mathsf{Schedule} = [\mathsf{Tx} \mapsto \mathbb{N}]$$

Resubmission parameter

$$\varrho \in (\mathsf{Tx} \mapsto \mathbb{N}) \times \mathsf{Schedule} \to \mathbb{P}(\mathsf{Tx}) \times \mathsf{Schedule}$$

State

$$(\mathit{pending}, \mathit{schedule}) \in \mathsf{Submission} = \mathsf{Pending} \times \mathsf{Schedule}$$

Atomic updates

$$\begin{aligned}
\mathsf{addPending}\ \mathit{txs}\ (\mathit{pending}, \mathit{schedule}) &= (\mathit{pending} \cup \mathit{txs}, \{\mathit{tx} \mapsto 0 \mid \mathit{tx} \in \mathit{txs}\} : \mathit{schedule}) \\
\mathsf{remPending}\ \mathit{txs}\ (\mathit{pending}, \mathit{schedule}) &= (\mathit{pending} \setminus \mathit{txs}, \mathit{schedule})
\end{aligned}$$

$$\begin{aligned}
\mathsf{tick}\ (\mathit{pending}, []) &= (\emptyset, (\mathit{pending}, [])) \\
\mathsf{tick}\ (\mathit{pending}, \mathit{due} : \mathit{schedule}) &= (\mathit{dropped}, (\mathit{pending}, \mathit{schedule}')) \\
&\quad \text{where } (\mathit{dropped}, \mathit{schedule}') = \varrho(\mathit{pending} \lhd \mathit{due}, \mathit{schedule})
\end{aligned}$$

Figure 14: Submission layer implementation

- If $t \notin \mathit{pending}_{\mathsf{Wallet}}$ but (still) $t \in \mathit{pending}_{\mathsf{Submission}}$ then (depending on the submission count) the submission layer may resubmit a transaction which has already been included in the blockchain, or report the transaction as 'dropped'. The former is harmless; the latter at worst simply confusing. Moreover, this may happen even if the wallet and the submission layer *are* synchronised: it's entirely possible that the transaction has been included in the blockchain but the wallet hasn't been informed of the block yet.

Although the specification uses a simple list for its schedule, if the overhead of a linear scan over all time slots to reschedule transactions is unacceptable, it can of course easily use a different list-like datatype such as a fingertree[10] (Hinze and Paterson, 2006).

10.3 Persistence

The state of the submission layer does not need to be persisted. If the wallet is shutdown for some period of time, the submission layer can simply be re-initialised from the state of the wallet, starting the submission process afresh for any transactions that the wallet still reports as pending. As long as the submission layer is able to report 'time until dropped' for still pending transactions, so that the user can see that all pending transactions have been reset to the initial expiry time of say 1 hour, it will be clear to the user what happened. This should be sufficient even for exchange nodes (especially since they will shutdown the wallet only very rarely).

If this reset to 1 hour (or whatever the expiry time is) is not acceptable, then the state of the submission layer *does* need to be persisted. The creation time of the transactions cannot be used, since this is a static value and will not change when the transaction gets explicitly resubmitted by the user after the submission layer decided to drop it.

10.4 Transactions with TTL

Dropping transactions after a certain time has passed is merely a stop-gap measure. Once a transaction has been broadcast across the network, it may be included at any point, possibly long after the submission layer has given up on it (unless the chain includes confirmed transactions that spends one or more of the transaction's inputs).

[10] Available in Haskell as `Data.Sequence`.

$$\mathsf{selectInputs} \in \mathsf{UTxO} \to \mathbb{P}(\mathsf{TxOut}) \to \mathsf{Maybe}\ \mathsf{Tx}$$

$$\begin{aligned} \mathsf{Just}\ (\mathit{inputs}, \mathit{outputs}') = \mathsf{selectInputs}\ \mathit{utxo}\ \mathit{outputs} \\ \textbf{ensures}\ \ \mathit{inputs} &\subseteq \mathrm{dom}\ \mathit{utxo} \\ \mathrm{range}\ \mathit{outputs}' &\supseteq \mathit{outputs} \\ \mathrm{range}\ \mathit{outputs}' \setminus \mathit{outputs} &\subseteq \mathsf{TxOut}_{\mathsf{ours}} \end{aligned}$$

Figure 15: Specification of input selection

The proper solution to this problem is to introduce a time-to-live (TTL) value for transactions, stating that the transaction must be included in the blockchain before a certain slot and simply dropped otherwise. A proper treatment of TTL would require revisiting every aspect of this specification; for now we just make a few observations:

- Once a TTL has been introduced, the core wallet *itself* can remove transactions from its *pending* set once the TTL has expired.
- This means that the submission layer does not need to implement expiry anymore, although it may still wish to keep track of a submission count so that it can implement exponential back-off.
- Persistence for the submission layer becomes even less important. The expiry of a transaction is now determined by the state of the blockchain, and moreover once a transaction *is* expired it cannot be resubmitted again (a new transaction, with a new TTL, must be signed).

11 Input selection

In this wallet specification we assume that new transactions to be submitted are provided to newPending fully formed. In reality this is preceded by a process known as *input selection* or *coin selection* which, given a set of desired outputs (that is, a *payment* that the user wishes to make), selects one or more inputs from the wallet's UTxO to cover that payment and the transaction fee, returning any change back to the wallet. The result of input selection is a fully formed transaction which can then be passed to newPending. This is summarised more formally in Figure 15.

Input selection is a large topic which merits a detailed study in its own right. Moreover, since input selection has multiple mutually incompatible goals, there is no single one-size-fits-all input selection algorithm. We will start with listing some goals that an input selection algorithm may have in Section 11.1, and some different use cases in Section 11.2. The remainder of this section will then continue by describing the algorithm we propose to use in the wallet, along with a detailed analysis (by means of simulation) of how the algorithm performs under various circumstances.

11.1 Goals

In this section we list some goals that a particular input selection algorithm may have. As is well-known (Lopp, 2015), many of these goals compete with each other. We may therefore wish to give wallet users some influence over this process, enabling them to prioritise some goals over others.

Low transaction fees. Transactions must include a transaction fee, which is based on the size of the transaction (Section 12). A good input selection algorithm will attempt to keep these transaction fees low. One complication here is that the fee depends on the size of the transaction, but the size of the transaction may depend on the fee since we may need to add more inputs to the transaction in order to cover the fee. Thus fee calculation and input selection are interdependent. There are situations where it is not immediately obvious that there is a terminating algorithm for selecting inputs and fees optimally. Note that minimising transaction fees *over time* does not necessarily mean that *every individual* transaction will be as small as possible.

Cryptographic security. Once an input at an address has been spent, its public key is publicly known and is arguably no longer suitable for very long term storage of funds due to the evolution of cryptography. The standard solution with input selection is to add a constraint that if we pick one input then we must pick all other inputs that were output to the same address. This results in no more funds remaining at the address (assuming an address non-reuse strategy such that there are no later payments to that address).

Privacy. The privacy goal is to make it impractical for other people observing the transactions in the ledger to tie an identity to all the funds belonging to that identity. For instance, it is preferable to have transactions with single inputs only, since otherwise attackers can reasonably assume[11] that of all the transactions inputs belong to the same identity (Reid and Harrigan, 2011). On the output side, we may wish to take steps to ensure that attackers cannot easily identify which output is the change output (Ermilov et al., 2017). For instance, for single payment transactions, we could try to ensure that the change output is roughly as large as the payment itself. Some systems give users the ability to override input selection on a per-transaction bases (sometimes known as "coin control"), since some transactions are more sensitive than others.

UTxO size. A transaction with a single input and two outputs will increase the size of the global UTxO by one entry, and (provided one of those is a change address), leave the size of the wallet's own UTxO unchanged; since incoming transactions will always grow the size of the wallet's UTxO, this means with such transactions the size of the wallet's UTxO will also grow without bound over time. Since the UTxO is kept in memory, this is undesirable. Instead input selection should attempt to keep the size of the UTxO steady. More specifically, if incoming transactions grow the wallet's UTxO by n entries on average, and the ratio of incoming transactions to outgoing transactions is $r : 1$, then ideally outgoing transactions should shrink the wallet's UTxO by $r \times n$ entries on average.

Distribution of magnitude of unspent outputs. Input selection can try to keep the distribution of the magnitude of unspent outputs close some ideal distribution. An obvious example is to avoid "dust": many tiny unspent outputs that result from change outputs. More generally, input selection can be given an ideal distribution a priori, or keep one dynamically based on the payment requests that come in. An example of an a-priori known requirement would be the ability to make payments of a certain size; if the UTxO only contains small unspent outputs, then for very large payments the resulting transaction might exceed the maximum transaction size. Conversely, if the UTxO only contains large outputs, the wallet may be forced to rely on unconfirmed transactions to maintain throughput, something we've expressly disallowed in this specification (Section 3.2) because it has negative consequences on networking performance. The privacy consequences of this kind of UTxO maintenance are far from obvious, but we note that some authors claim it may actually *help* (Ron and Shamir, 2013, Section 2).

11.2 Use cases

As an example of how different users might prioritise different goals, we will consider two use cases, at opposite ends of the spectrum: small end users and exchange nodes.

Exchange nodes. Exchanges have high rates of incoming and outgoing transactions, large overall balances and will tend to have large UTxOs. For this use case we are concerned with asymptotic complexity (due to the large UTxO) and have a goal of high throughput, but we are not overly concerned with the goals of achieving privacy or minimising fees. Exchanges tend to follow deposit policies which are incompatible with the cryptographic security goal as described above, so this is not a goal of the policy. Exchanges may occasionally need to make very large payments.

Individual users. For individual users we assume a low rate of incoming and outgoing transactions and a comparatively small UTxO. For this use case we are not too concerned with asymptotic complexity as the UTxO is assumed to be small, nor with a goal of high throughput. We are concerned with the privacy goal, as individual users are able to use their wallets in a way that preserves a degree of privacy. We are somewhat concerned with keeping fees reasonably low. Users may want the ability to 'empty their wallet' (i.e., create a single transaction that spends all of the wallet's UTxO, sending it to a single output address.)

[11] In systems such as Bitcoin there are services known as "laundry services", "mixers" or "tumblers" (de Balthasar and Hernandez-Castro, 2017), which are trusted third-parties that combine payments from various users into a single transaction, to break this assumption.

11.3 Self organisation

The term 'self organisation' refers to the emergence of complex behaviour (typically in biological systems) from simple rules and random fluctuations. In this section we will see how we can take advantage of self organisation to design a simple yet effective coin selection algorithm.

In this section we present the conclusions of an in-depth study of coin selection that we have done as part of the the redesign of the Cardano wallet. We will describe some of the problems and trade-offs that need to be made. We then propose a novel coin selection algorithm. The algorithm is efficient and straight-forward to implement, and through studying simulations of the algorithm under various conditions we will see that it is very effective. As our starting point, we will use the random algorithm described by Mark Erhardt in his masters thesis (Erhardt, 2016).

11.4 Dust

An obvious strategy that many coin selection algorithms use in some form or other is "try to get as close to the requested value as possible". The problem with such an approach is that it tends to create a lot of *dust*: small unspent outputs that remain unused in the user's wallet because they're not particularly useful. For example, consider the 'largest first' algorithm: a simple algorithm which considers all unspent outputs of the wallet in order of size, adding them to a running total until it has covered the requested amount.

Figure 16 shows the effect of this algorithm. In order to evaluate a coin selection policy we need an "event stream" of deposits and withdrawals to evaluate it against. For the particular simulation shown in Figure 16 we use a 3:1 ratio between deposits and withdrawals (i.e., 3 times more deposits than withdrawals), with both deposits and withdrawals normally distributed. We will consider many more different event streams later in this section.

There are various things to see in this graph, but for now we want to focus on the UTxO histogram and its size. Note that as time passes, the size of the UTxO increases and increases, up to about 60k entries after about 1M cycles (with 3 deposits and 1 payment per cycle). A wallet with 60k entries is *huge*, and looking at the UTxO histogram we can see why this happens: virtually all of these entries are dust. We get more and more small outputs, and those small outputs are getting smaller and smaller.

11.5 Cleaning up

Erhardt makes the following very astute observation:

> "If 90% of the UTxO is dust, then if we pick an unspent output *randomly*, we have a 90% change of picking a dust output."

He concludes that this means that a coin selection algorithm that simply picks unspent outputs *at random* might be pretty effective; in particular, effective at collecting dust. Indeed, it is. Consider the simulation shown in Figure 17.

Notice quite how rapidly the random coin selection reduces the size of the UTxO once it kicks in. If you look at the inputs-per-transaction histogram, you can see that when the random input selection takes over, it first creates a bunch of transactions with 10 inputs (we limited transaction size to 10 for this simulation), rapidly collecting dust. Once the dust is gone, the number of inputs shrinks to about 3 or 4, which makes perfect sense given the 3:1 ratio of deposits and withdrawals.

We will restate Erhardt's observation as our first self organisation principle:

> **Self organisation principle 1**. Random selection has a high probability of picking dust outputs *precisely when* there is a lot of dust in the UTxO.

It provides a very promising starting point for an effective coin selection algorithm, but there are some improvements we can make.

11.6 Active UTxO management

Figure 18 shows a simulation of a pure "select randomly until we reach the target value" coin selection algorithm. The first observation is that this algorithm is doing *much* better than the largest-first policy in terms of the size of the UTxO, which is about *2 orders of magnitude* smaller: a dramatic improvement. However, if we look at the UTxO histogram, we can see that there is room for improvement: although this algorithm is good at collecting dust, it's also

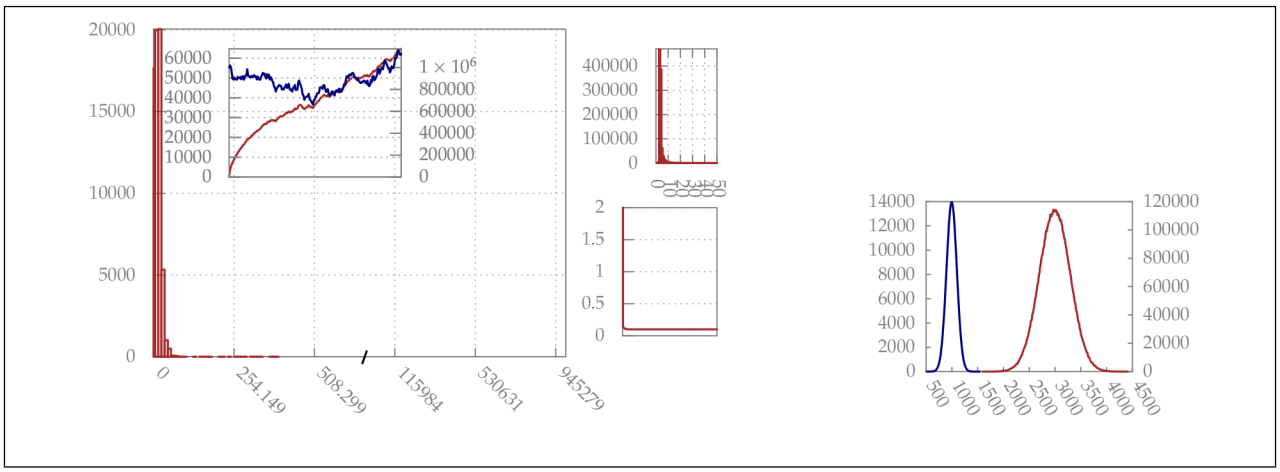


Figure 16: Simulation of largest-first coin selection. Main histogram shows UTxO entries; inset graph shows UTxO balance in blue and UTxO size in red, histogram top-right shows number of inputs per transaction, graph bottom right shows the change:payment ratio (more on that below). Graph on the far right shows the distribution of deposits (blue, right axis) versus payments (red, left axis). In this case, both are normally distributed with a mean of 1000 and 3000 respectively, and we have a deposit:payment ratio of 3:1; modelling a situation where we have frequent smaller deposits, and less frequent but larger payments (withdrawals). The wallet starts with an initial balance of 1M.

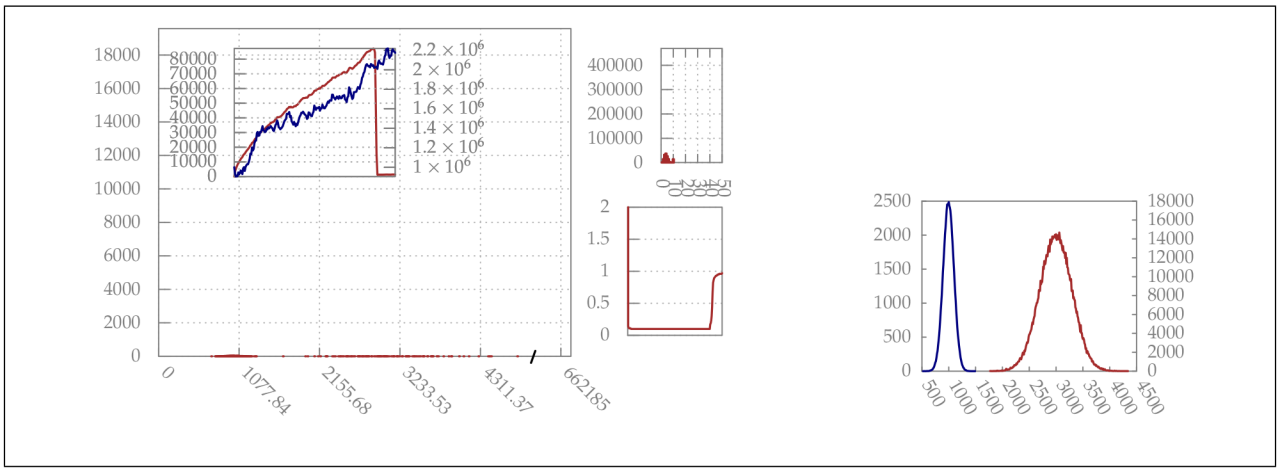


Figure 17: Same distribution and ratio as in Figure 16; we run the largest-first algorithm for 1M cycles, and then random coin selection for another 150k cycles.

still *generating* quite a bit of dust. The UTxO histogram has two peaks. The first one is approximately normally distributed around 1000, which are the deposits that are being made. The second one is near 0, which are all the dust outputs that are being created.

This brings us to the topic of active UTxO management. In an ideal case, coin selection algorithms should, over time, create a UTxO that has "useful" outputs; that is, outputs that allow us to process future payments with a minimum number of inputs. We can take advantage of self organisation again:

> **Self organisation principle 2**. If for each payment request for value x we create a change output roughly of the same value x, then we will end up with a lot of change outputs in our UTxO of size x *precisely when* we have a lot of payment requests of size x.

We will consider some details of how to achieve this in the next section. For now, Figure 19 shows what the effect of this is on the UTxO. The graph in the bottom right, which we've ignored so far, records the change:payment ratio. A value near zero means a very small change output (i.e., dust); a very high value would be the result of using a huge UTxO entry for a much smaller payment. A value around 1 is perfect, and means that we are generating change outputs of equal value as the payments.

Note that the UTxO now follows precisely the distribution of payment requests, and we're not generating dust anymore. One advantage of this is that because we have no dust, the average number of inputs per transaction can be lower than in the basic algorithm.

Just to illustrate this again, Figure 20 shows the result of the algorithm but now with a 3:1 ratio of deposits and withdrawals. We have two bumps now: one normally distributed around 1000, corresponding to the the deposits, and one normally distributed around 3000, corresponding to the payment requests that are being made. As before, the median change:payment ratio is a satisfying round 1.0.

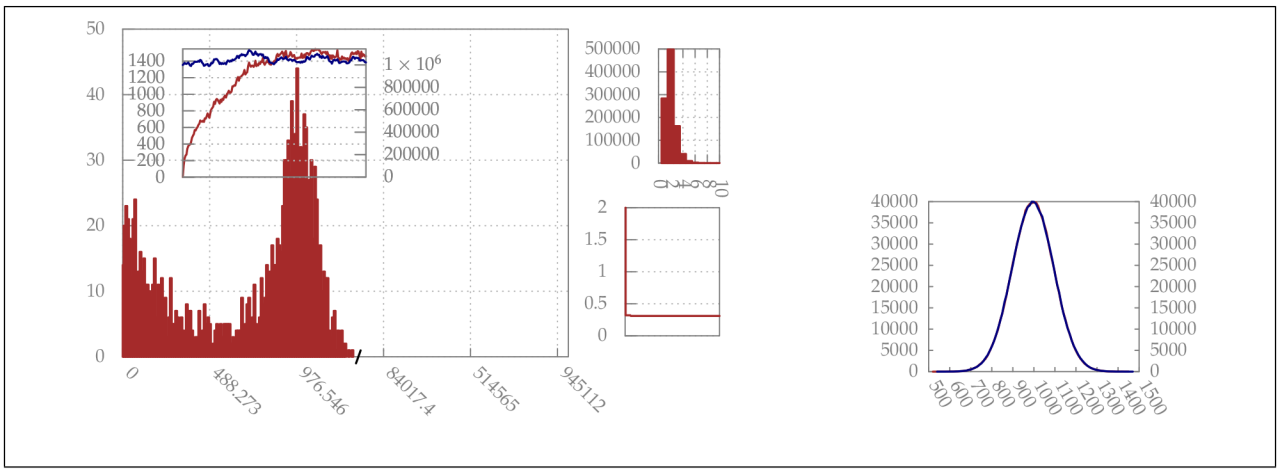


Figure 18: Random-until-value-reached, for a 1:1 ratio of deposits and withdrawals, both drawn from a normal distribution with mean 1000.

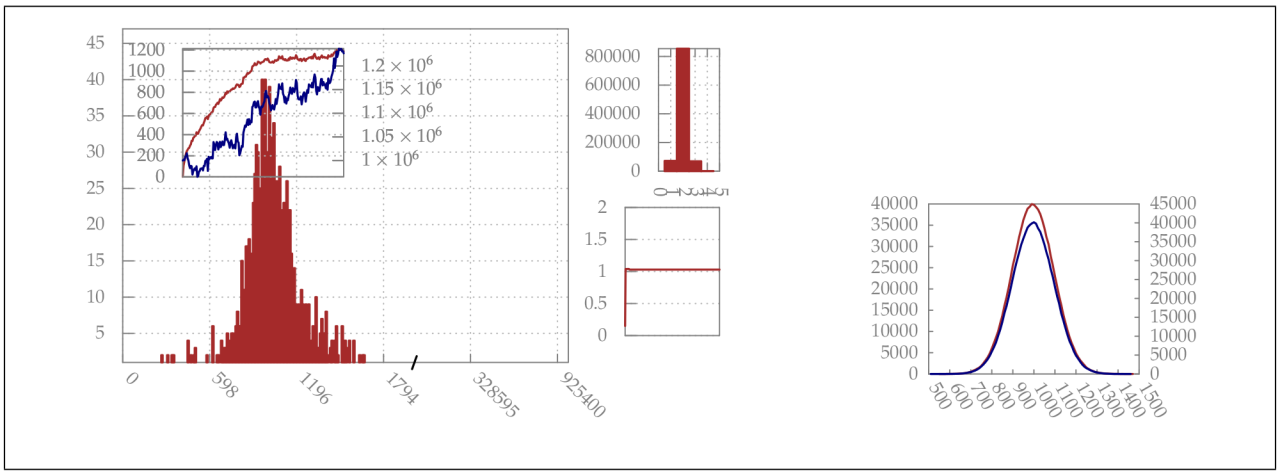


Figure 19: Same deposits and withdrawals as in Figure 18, but now using the "pick randomly until we have a change output roughly equal to the payment" algorithm.

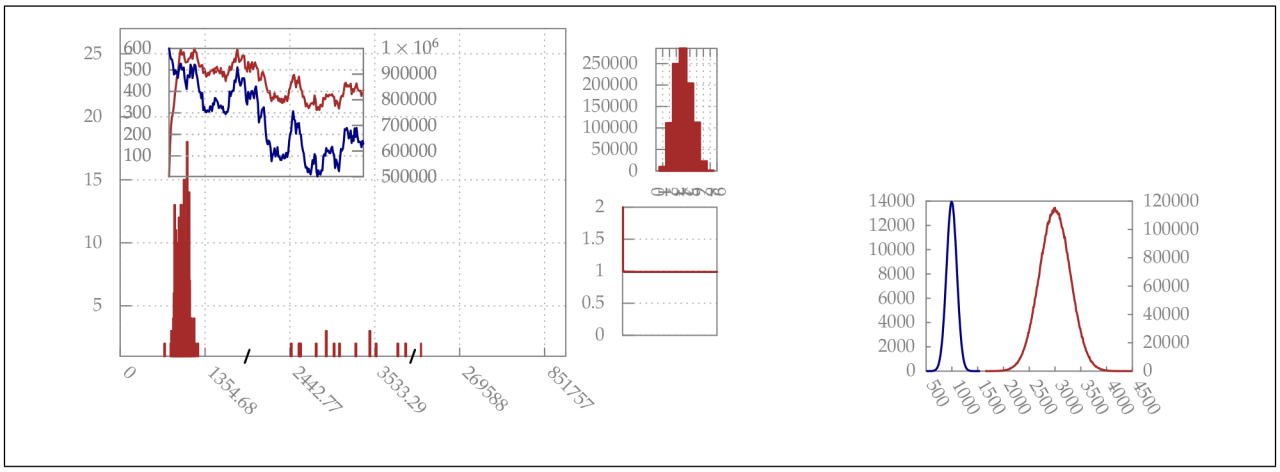


Figure 20: Same algorithm as in Figure 19, but now with 3:1 deposits:payments (i.e., many small deposits, fewer but larger payments).

1. Randomly select outputs from the UTxO until the payment value is covered.
   (In the rare case that this fails because the maximum number of transaction inputs has been exceeded, fall-back on the largest-first algorithm for this step.)
2. Randomly select outputs from the UTxO, considering for each output if that output is an *improvement*. If it is, add it to the transaction, and keep going. An output is considered an improvement when:
   (a) It doesn't exceed the specified upper limit
   (b) Adding the new output gets us closer to the ideal change value
   (c) It doesn't exceed the maximum number of transaction inputs.

Figure 21: The `Random-Improve` algorithm. Side note for point (2a): we use twice the value of the payment as the upper limit. Side note for point (2b): it might be that without the new output we are slightly below the ideal value, and with the new output we are slightly above; that is fine, as long as the absolute distance decreases.

11.7 The `Random-Improve` algorithm

We are now ready to present the coin selection algorithm we propose. The basic idea is simple: we will randomly pick UTxO entries until we have reached the required value, and then continue randomly picking UTxO entries to try and reach a total value such that the the change value is roughly equal to the payment.

This presents a dilemma though. Suppose we have already covered the minimum value required, and we're trying to improve the change output. We pick an output from the UTxO, and it turns out to be huge. What do we do? One option is to discard it and continue searching, but this would result in coin selection frequently traversing the entire UTxO, resulting in poor performance.

Fortunately, self organisation comes to the rescue again. We can set an upper bound on the size of the change output we still consider acceptable (we will set it to twice the payment value). Then we take advantage of the following property.

> **Self organisation principle 3.** Searching the UTxO for additional entries to improve our change output is only useful if the UTxO contains entries that are sufficiently small enough. But *precisely when* the UTxO contains many small entries, it is less likely that a randomly chosen UTxO entry will push the total above the upper bound we set.

In other words, our answer to "what do we do when we happen to pick a huge UTxO entry?" is "we stop trying to improve our selection". Figure 21 shows the full algorithm.

11.8 Evaluation

The algorithm (Figure 21) is deceptively simple. Do the self organisation principles we isolated really mean that order will emerge from chaos? Simulations suggest, yes, it does. We already mentioned how random input selection does a great job at cleaning up dust in Figure 17; what we didn't emphasise in that section is that the algorithm we simulated there is actually our `Random-Improve` algorithm. Notice how the median change:payment ratio is initially very low (indicative of a coin selection algorithm that is generating a lot of dust outputs), but climbs rapidly back to 1 as soon as `Random-Improve` kicks in. We already observed that it does indeed do an excellent job at cleaning up the dust, quickly reducing the size of the UTxO. The simulations in Figures 19 and 20 are also the result of the `Random-Improve` algorithm.

That said, of course the long term effects of a coin selection algorithm can depend strongly on the nature of the distribution of deposits and payments. It is therefore important that we evaluate the algorithm against a number of different distributions.

11.8.1 Normal distribution, 10:1 deposit:payment ratio

We already evaluated 'Random-Improve' against normally distributed payments and deposits with a 1:1 ratio and a 3:1 ratio; perhaps more typical for exchange nodes might be even higher ratios. Figure 22 shows is a 10:1 ratio.

We see a very similar picture as we did in Figure 20. Since the deposits and payments are randomly drawn (from normal distributions), the UTxO balance naturally fluctuates up and down. What is satisfying to see however is that the *size* of the UTxO tracks the balance rather precisely; this is about as good as we can hope for. Notice also that the change:payment ratio is a nice round 1, and the average number of transaction inputs covers around 10 or 11, which is what we'd expect for a 10:1 ratio of deposits:payments.

11.8.2 Exponential distribution, 1:1 deposit:payment ratio

What if the payments and deposits are not normally distributed? Figure 23 shows `Random-Improve` on exponentially distributed inputs. In an exponential distribution we have a lot of values near 0; for such values it will be hard to achieve a 'good' change output, as we are likely to overshoot the range. Partly due to this reason the algorithm isn't quite achieving a 1.0 change:payment ratio, but at 1.5 it is still generating useful change outputs. Furthermore, we can see that the size of the UTxO tracks the UTxO balance nicely, and the average number of transaction inputs is low, with roughly 53% having just one input. Moreover, when we increase the deposit:payment ratio to 3:1 and then 10:0, the change:payment ratio drops to about 1.1 and then back to 1.0 (graphs omitted).

11.8.3 Erlang

The exponential distribution results in many very small deposits and payments. The algorithm does better on slightly more realistic distributions such as the Erlang-k distributions (for $k > 1$)[12]. Figure 24 shows the results for the 3:1 deposit:payment ratio using the Erlang-3 distribution; the results for other ratios (including 1:1) and other values of k (we additionally simulated for $k = 2$ and $k = 10$) are similar.

11.8.4 More payments than deposits

We have been focusing on the case where we have more deposits and fewer (but larger) payments. Figure 25 shows what happens if the ratio is reversed. In this case we are unable to achieve that perfect 1.0 change:payment ratio, but this is expected: when we have large deposits, then we frequently have no choice but to use those, leading to large change outputs.

We can see this more clearly when we slow things right down, and remove any source of randomness. Figure 26 shows the same 1:10 ratio again, but now only the first 100 cycles, and all deposits exactly 10k and all payments exactly 1k. We can see the large value being deposited, and then shifting to the left in the histogram as it is getting used for deposits, each time decreasing that large output by 1k. Indeed, this takes 10 slots on average, which makes sense given the 10:1 ratio; moreover, the *average value* of the "large output" in such a 10-slot cycle is 5k, explaining why we are getting 5.0 change:payment ratio.

The algorithm however is *not* creating dust outputs; the 1k change outputs it is generating *are* getting used, and the size of the UTxO is perfectly stable. Indeed, back in Figure 25 we can see that the size of the UTxO tracks the balance perfectly; moreover, the vast majority of transactions only use a single input, which is what we'd expect for a 10:0 deposit:payment ratio.

11.8.5 Real data

`MoneyPot.com` Ideally, of course, we run the simulation against *real* event streams from existing wallets. Unfortunately, such data is hard to come by. Erhardt was able to find one such dataset, provided by `MoneyPot.com` (Havar, 2015). Figure 27 shows the results of running our algorithm on this dataset.

A few observations are in order here. First, there are quite a few deposits and payments close to 0, just like in an exponential distribution. Moreover, although we have many values close to 0, we also have some *huge* outliers; the payments are closely clustered together, but there is a 10^9 difference between the smallest and largest deposit, and moreover a 10^5 difference between the largest deposit and the largest payment. It is therefore not surprising that we end up with a relatively large change:payment ratio. Nonetheless, the algorithm is behaving well, with the size of the UTxO tracking the balance nicely, with an average UTxO size of 130 entries. The average number of outputs is 3.03, with 50% of transactions using just one input, and 90% using 6 or fewer.

[12]The exponential distribution discussed in Section 11.8.2 is the Erlang-1 case, of course.

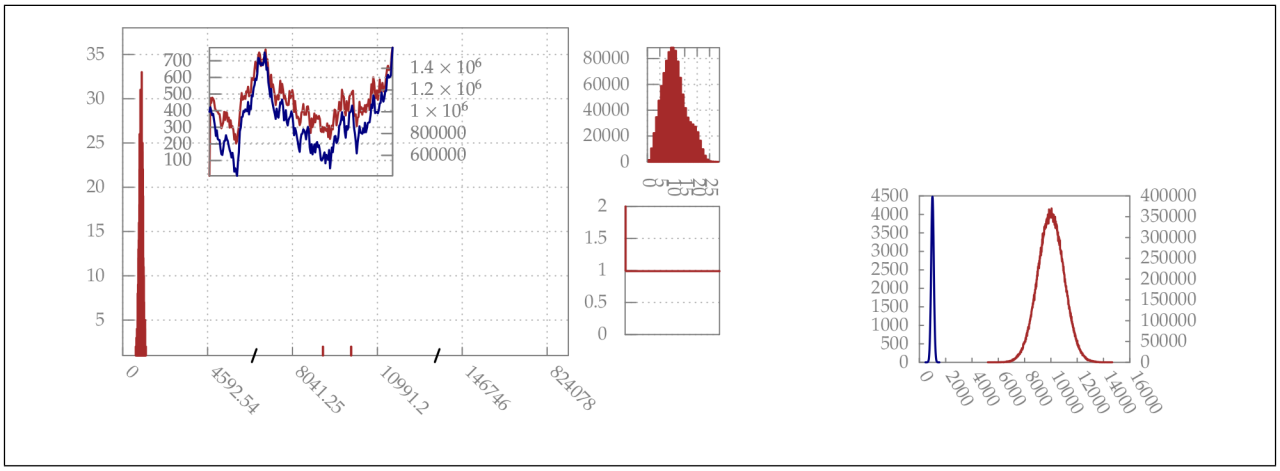


Figure 22: `Random-Improve` with a 10:1 deposit:payment ratio, both normally distributed.

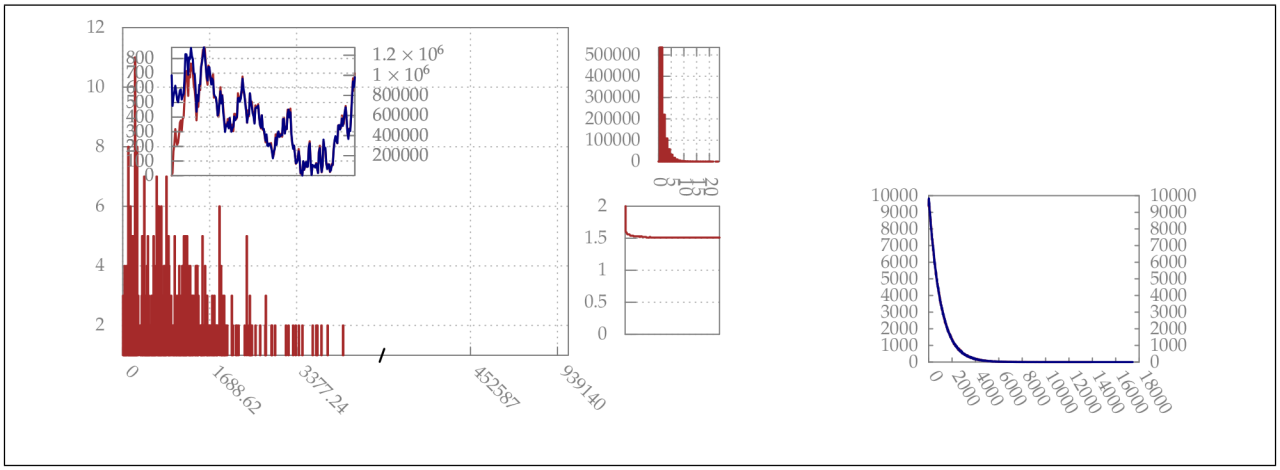


Figure 23: `Random-Improve`, 1:1 deposit:payment ratio, deposits and payments both drawn from an exponential distribution with scale 1000.

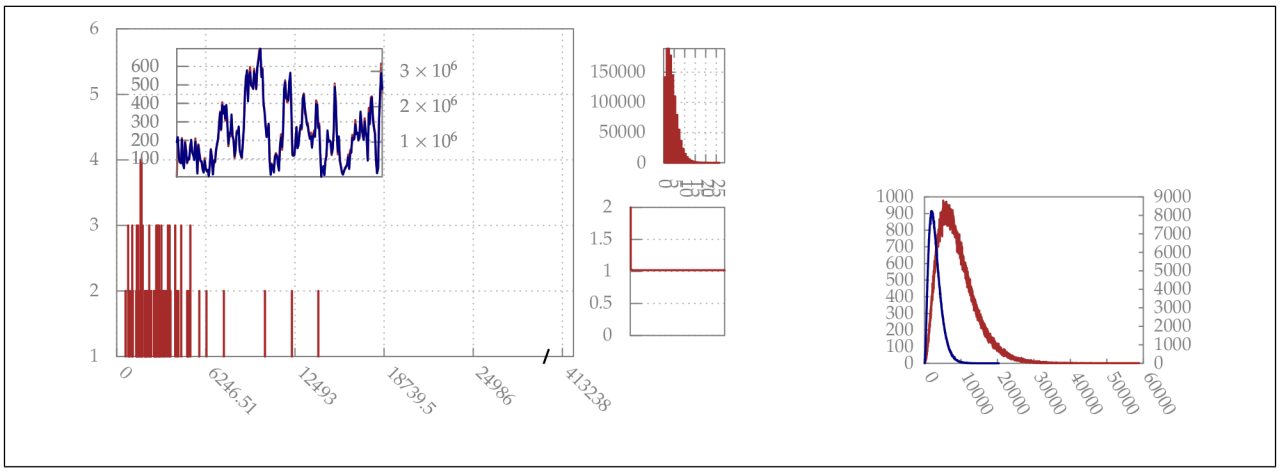


Figure 24: `Random-Improve`, 3:1 deposit:payment ratio, deposits drawn from an Erlang-3 distribution with scale 1000 and payments drawn from Erlang-3 distribution with scale 3000.

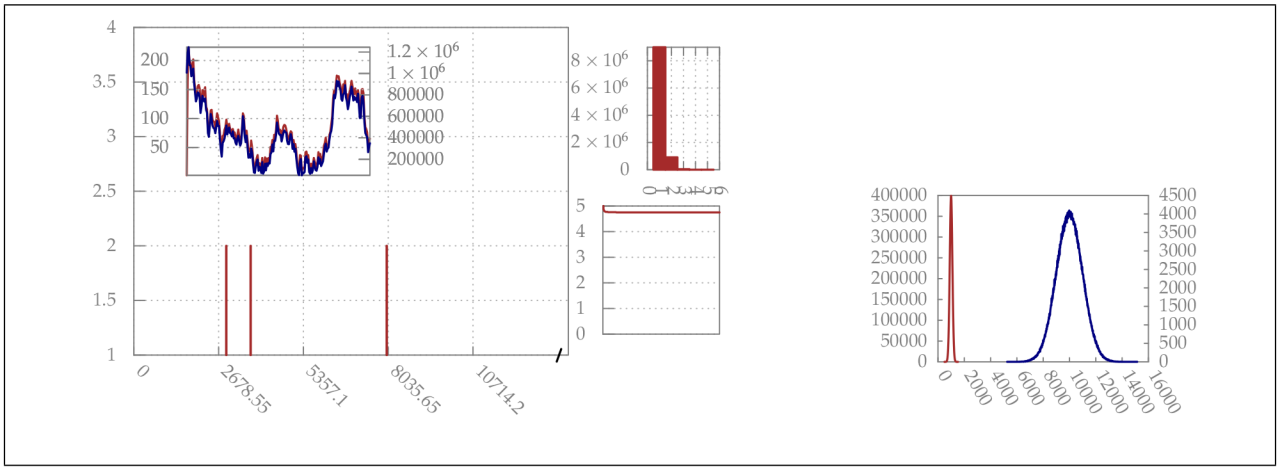


Figure 25: `Random-Improve`, 1:10 deposit:payment ratio, deposits and payments drawn from a normal distribution with mean 10k and 1k, respectively. 1M cycles.

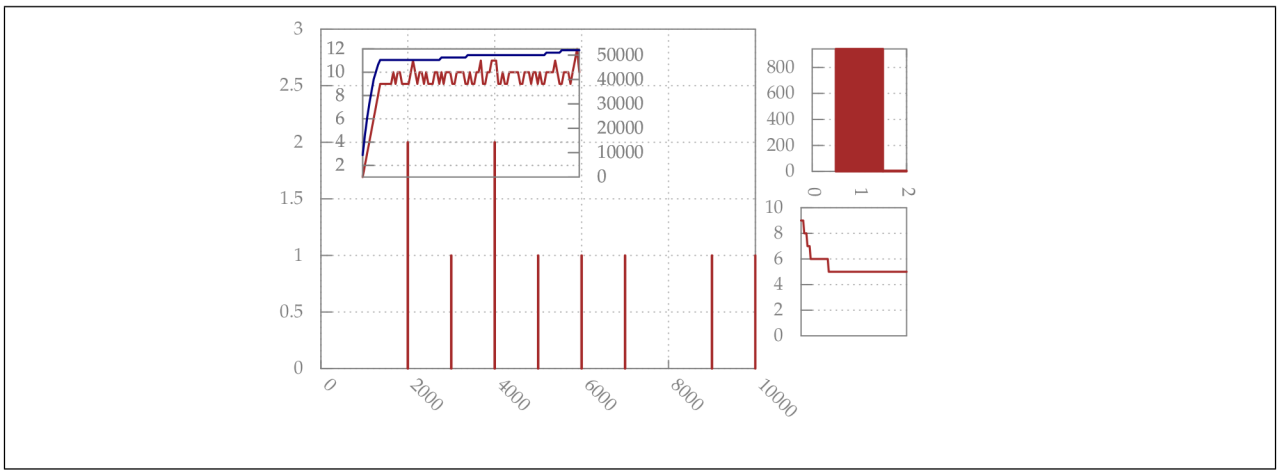


Figure 26: `Random-Improve`, 1:10 deposit:payment ratio, all deposits exactly 10k, all payments exactly 1k (no randomness). First 100 cycles only.

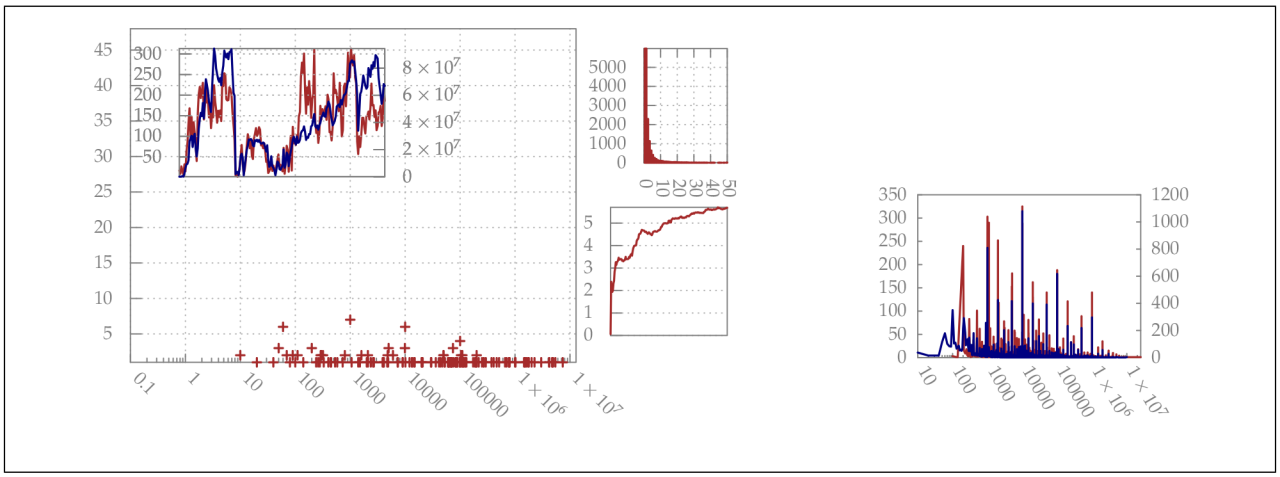


Figure 27: `Random-Improve`, using the MoneyPot data set. There is a roughly 2:1 deposit:payment ratio. Values have been scaled. Log scale on the x-axis.

Cardano Exchange One of the large Cardano exchange node has also helped us with some anonymised data (deposits and payments), similar in nature to the MoneyPot dataset albeit significantly larger. Coming from an exchange node, however, this dataset is *very much* skewed towards deposits, with a deposit:payment ratio of roughly 30:1. Under these circumstances (Figure 28) our coin selection algorithm cannot keep the UTxO size small on its own (exchanges have additional infrastructure in place to address this problem). The UTxO here also contains some truly enormous outputs, which is pushing the change:payment ratio up, although as more withdrawals are being made the algorithm is able to push it down.

11.9 Conclusions

The choice of coin selection algorithm has far reaching consequences on the long term behaviour of a cryptocurrency wallet. To a large extent the coin selection algorithm determines, over time, the shape of the UTxO. Moreover, the performance of the algorithm can be of crucial importance to high-traffic wallets such as exchange nodes.

In his thesis, Erhardt proposes "Branch and Bound" as his preferred coin selection algorithm. Branch and Bound in essence is a limited backtracking algorithm that tries to find an exact match, so that no change output needs to be generated at all. When the backtracking cannot find an exact match within its bounds, the algorithm then falls back on random selection. It does not, however, attempt our "improvement" step, and instead just attempts to reach a minimum but fixed change size, to avoid generating dust. It is hard to compare the two algorithms directly, but on the MoneyPot dataset at least the results are comparable; Erhardt ends up with a slightly smaller average UTxO (109 versus our 130), and a slightly smaller average number of inputs (2.7 versus our 3.0). In principle we could modify our `Random-Improve` algorithm to start with bounded backtracking to find an exact match, just like Erhardt does; we have not done this however because it adds complexity to the algorithm and reduces performance. Erhardt reports that his algorithm is able to find exact matches in 30% of the time. This is *very* high, and at least partly explains why his UTxO and average number of change outputs is lower; in the Cardano blockchain, we would not expect that there *exist* exact matches anywhere near that often (never mind *finding* them).

Instead our proposed `Random-Improve` does no search at all, instead purely relying on self organisation principles, the first of which was stated by Erhardt, and the other two we identified as part of this research. Although in the absence of more real data it is hard to evaluate any coin selection algorithm, we have shown that the algorithm performs well across a large variety of different distributions and deposit:payment ratios. Moreover it is straight-forward to implement and has high performance.

One improvement we may wish to consider is that *when* there are very large deposits, we could occasionally issue a "reorganisation transaction" that splits those large deposits into smaller chunks. This would bring the change:payment ratio down, which would improve the evolution of the UTxO over time and is beneficial also for other, more technical reasons (it reduces the need for dependent transactions). Such reorganisation is largely orthogonal to this algorithm, however, and can be implemented independently.

12 Appendix: Transaction fees

Since this specification is not concerned with blockchain validation, it does not require a formal treatment of transaction fees. Even new transactions submitted to newPending are assumed to be fully formed and valid. Indeed, the only section where we even need the concept of fees is **??** on input selection, where we mention that a transaction constructed by input selection must satisfy the minimum fee requirement. For completeness sake, in this appendix we outline how a transaction fee is represented in Cardano, and how the minimum fee is computed.

Although our formalisation of UTxO style accounting is based on that of Zahnentferner (2018), we diverge slightly from that formalisation and do not represent fees explicitly. Instead, a transaction fee is simply the difference between the transactions inputs and outputs:

$$\begin{aligned}&\mathsf{fee} \in \mathsf{UTxO} \to \mathsf{Tx} \to \mathsf{Coin}\\&\mathsf{fee}_{utxo}\ tx = \mathsf{totalin}_{utxo}\ tx - \mathsf{totalout}\ tx\end{aligned}$$

where

$$\begin{aligned}\mathsf{totalin}_{utxo}(inputs,\ \_\ ) &= \mathsf{balance}\ (inputs \lhd utxo)\\ \mathsf{totalout}\ \ (\_,\ outputs) &= \sum_{(\_,c)\in outputs} c\end{aligned}$$

where totalin is a function of the UTxO only because we need the UTxO to know the size of a particular input (which, after all, is merely a transaction hash and an index). It therefore has the precondition

$$\begin{aligned}&\mathsf{totalin}_{utxo}\ (inputs, \_)\\ &\quad \textbf{requires}\ inputs \subseteq \mathrm{dom}\ utxo\end{aligned}$$

This implicit representation of fees is suitable for this specification since we don't need to reason about fees; moreover, this actually matches how fees are represented in the actual Cardano blockchain (as well as in Bitcoin).

The minimum fee of a transaction is given by some linear function f on the serialised size of the transaction

$$\begin{aligned}&\mathsf{minfee} \in \mathsf{Tx} \to \mathsf{Coin}\\ &\mathsf{minfee} = f \circ \mathsf{size} \circ \mathsf{serialise}\end{aligned}$$

References

Blelloch, G. E., Ferizovic, D., and Sun, Y. (2016). Parallel ordered sets using join. *CoRR*, abs/1602.02120.

de Balthasar, T. and Hernandez-Castro, J. (2017). An analysis of bitcoin laundry services. In Lipmaa, H., Mitrokotsa, A., and Matulevičius, R., editors, *Secure IT Systems*, pages 297–312, Cham. Springer International Publishing.

Erhardt, M. (2016). An Evaluation of Coin Selection Strategies. Master's thesis, Karlsruhe Institute of Technology.

Ermilov, D., Panov, M., and Yanovich, Y. (2017). Automatic bitcoin address clustering. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 461–466.

Havar, R. (2015). The `MoneyPot.com` data set.

Hinze, R. and Paterson, R. (2006). Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197217.

Lopp, J. (2015). The challenges of optimizing unspent output selection. `https://medium.com/@lopp/the-challenges-of-optimizing-unspent-output-selection-a3e5d05d13ef`.

Reid, F. and Harrigan, M. (2011). An analysis of anonymity in the bitcoin system. In *Security and Privacy in Social Networks*.

Ron, D. and Shamir, A. (2013). Quantitative analysis of the full bitcoin transaction graph. In Sadeghi, A.-R., editor, *Financial Cryptography and Data Security*, pages 6–24, Berlin, Heidelberg. Springer Berlin Heidelberg.

Zahnentferner, J. (2018). Chimeric ledgers: Translating and unifying utxo-based and account-based cryptocurrencies. Cryptology ePrint Archive, Report 2018/262. `https://eprint.iacr.org/2018/262`.

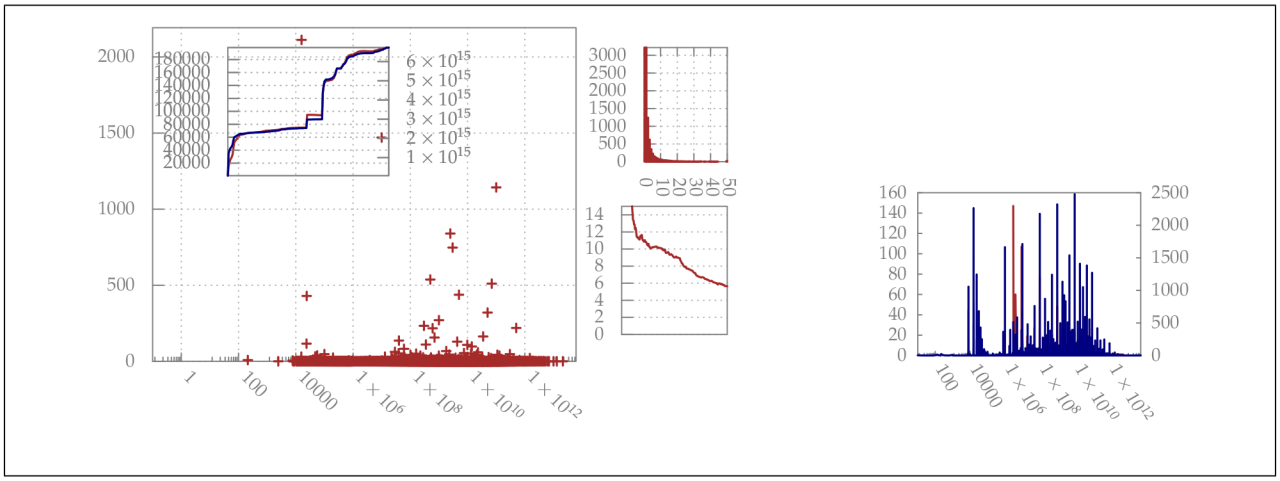


Figure 28: `Random-Improve`, using data set from a large Cardano exchange. There is a roughly 30:1 deposit:payment ratio. Values have been scaled. Log scale on the x-axis.