b
UNIVERSITÄT
BERN

A concurrent DEX on Cardano
How to write scalable apps on a UTXO blockchain

Bachelor Thesis

Peter Brühwiler

from
Bern, Switzerland

Faculty of Science, University of Bern

31. December 2021

Prof. Christian Cachin
Luca Zanolini, Jovana Micic
Cryptology and Data Security Group
Institute of Computer Science
University of Bern, Switzerland

Abstract

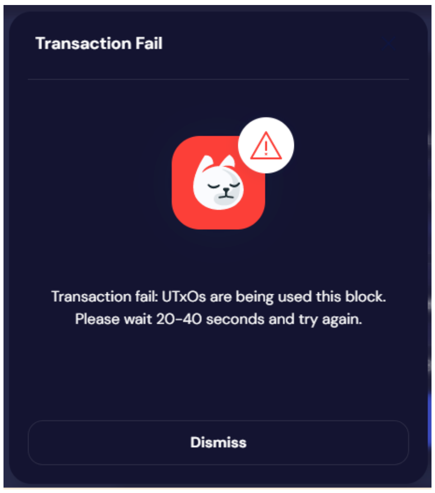
Like Bitcoin, Cardano is an Unspent Transaction Output (UTXO) based blockchain. However while on the Bitcoin blockchain UTXOs are spent by basically signing transactions with a private key (using a very limited scripting language), Cardano allows arbitrary validation logic by introducing the Haskell based Plutus scripting language to define spending conditions. Cardano's UTXO model demands a different design approach for decentralized applications than the more common account model (used for instance by Ethereum). So, simply taking a smart contract from Ethereum and translating it "from Solidity to Plutus" is often a bad idea. In this paper I describe how a 'concurrency problem' can arise when multiple parties simultaneously try to use an 'Ethereum-style' decentralized exchange (DEX) on Cardano and implement a simple solution. Multiple teams in the Cardano ecosystem are implementing or have already implemented concurrent DEXs by the time of writing this paper, but no Automated Market Maker (AMM) style DEX has gone live on the Cardano mainnet yet.

Contents

Chapter 1

Introduction

In 2021, the ’Alonzo’ hard fork brought smart contract capability to the Cardano blockchain. A few days before the mainnet hard fork, the ’concurreny issue’ became a hot topic in some cryptocurrency social media channels. The reason: Minswap, one of the teams building a DEX for the Cardano blockchain, had gone live with a test version of their application on the public testnet and most people wanting to try it out just got this message:


Transaction Fail

Transaction fail: UTxOs are being used this block.
Please wait 20-40 seconds and try again.

Figure 1.1. UTXO missing..

In a blogpost [1], Minswap appologized for the bad user experience but also argued that testnets exist to improve unfinished products: ”Minswap has known about the concurrency challenge since we first began building on Cardano over 6 months ago”, they wrote. ”It’s an issue that every competent team and development lab building DeFi protocols on Cardano must overcome. It’s not a fundamental flaw, but is simply a design challenge that must be addressed.” About a month later they published a Medium article, introducing their solution called Laminar [2] as ”an eUTXO scaling protocol for accounting-style smart contract”. Sundaeswap, another project on Cardano, described the ’concurrency issue’ faced by a UTXO-style DEX already earlier, in their first whitepaper [3] published on June 1 2021. ”Because any given eUTXO can only be spent once, as part of one transaction, it appears as if only one swap can happen per block”, they explained. ”On the Cardano blockchain, there is roughly one block every 20 seconds. This would be abysmal throughput for a decentralized exchange.”

1.1 The extended UTXO model

To understand the 'concurrency issue' it is important to understand Cardanos UTXO architecture. UTXOs are easiest explained with the cash analogy. While users of account based blockchains like Ethereum control an account whose value can be increased or decreased, users of the Cardano blockchain just control a bunch of UTXOs, all containing a certain value. As the first letter in 'UTXO' says, these outputs are 'unspent', exactly like a banknote kept in a physical wallet. And in the same way as a banknote can be spent, these UTXOs can be spent only as a whole. So, a crypto wallet holding a UTXO worth 100 Ada (Ada being Cardano's original currency) wanting to pay 20 Ada to a seller, has to put the whole amount into the transaction. The transaction will then have two outputs: a UTXO worth 20 Ada for the seller and a UTXO worth 80 Ada (minus the transaction fee, to be precise) going back to the buyer. Summarizing the result of the buy transaction: One UTXO has been spent (and will be unusable in the future) and two new UTXOs have been created (and will be usable by the the seller/buyer once the block recording the transaction is included in the ledger state).

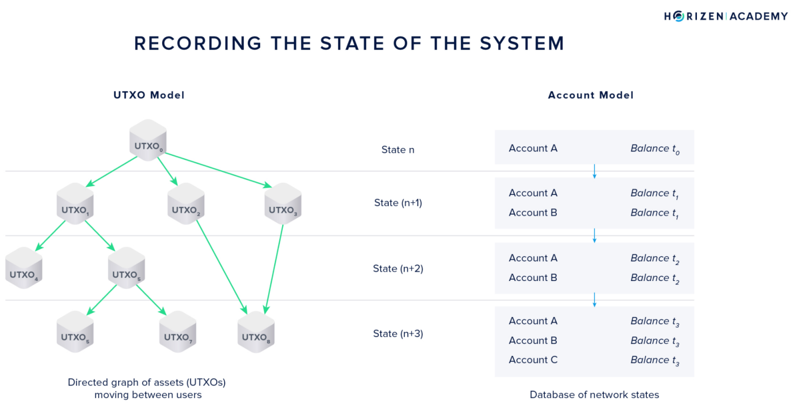


Figure 1.2. UTXO style vs accounting style ledger representation.

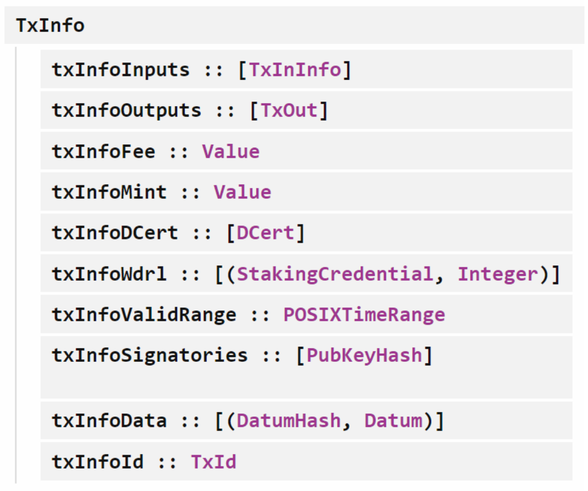
Figure 1.2 shows the difference between the UTXO and account style model: the first manipulates the ledger state through 'destruction and creation' without changing the state of variables, the second changes the ledger state through manipulation of global (account) variables. The reason why both these models exist is explained by Chakravarty et al. [4] as follows: "Ethereum chose the account model explicitly to facilitate more expressive smart contracts. On the other hand, Bitcoin chose UTXO also for good reasons, including that its semantic model stays simple in a complex concurrent and distributed computing environment. This raises the question of whether it is possible to have expressive smart contracts, while keeping the semantic simplicity of the UTXO model." The answer given to this question by Cardano (and the authors of the above cited paper) is the eUTXO, the extended UTXO. There are four major concepts that transform a UTXO into a eUTXO:

- The first extension is that the eUTXO is associated to a 'Validator' (or to be more precise: to an address given by the hash of the validator script) instead of to a public key. The validator script is a function that must evaluate to True in order to unlock UTXOs sitting at the address given by that very same validator script. Similarly in Bitcoin, scriptPubKeys are represented as BitcoinScriptAddresses with the Pay-To-Script Hash (P2SH).
- The second change is that the eUTXO contains a 'Datum', which allows it to carry some state. To be more precise: currently, a eUTXO just contains the hash of the Datum and

the spending transaction must provide the actual Datum value.

- The third extension is the ’Redeemer’: To spend an eUTXO, this Redeemer must - as with the datum - be passed as a parameter into the validator script. The Redeemer typically describes an action. For instance if an eUTXO carries in it’s Datum an exchange rate as state, a Redeemer ’update’ could trigger a certain part of the validation logic in the validator. So a validator script could contain the condition that updating the state of the exchange rate is only possible for the party having created the original eUTXO while other actions like ’use’ can be performed by arbitrary users.

- The last change is that the validator script of an eUTXO sees the whole transaction that is currently being validated (a transaction can include as inputs and outputs multipe UTXOs from different script addresses and private addresses). This information, called the context, is passed into the validator as an additional argument of type ScriptContext. As the authors of the eUTXO paper explain: ”The information supplied in the context enables the validator to enforce much stronger conditions than is possible with a bare UTXO model — in particular, it can inspect the outputs of the current transaction, which is essential for ensuring contract continuity.” In the example mentioned in the previous point, this contract continuity would be: the eUTXO containing the exchange rate can only be spent in a transaction if a new UTXO is produced in the same transaction that also contains the exchange rate and is associated to the same script address. If this validation rule is not enforced, the exchange rate state and thus the contract continuity could get ’lost’. Figure 1.3 shows the most important component of the ScriptContext, the TxInfo record. It contains, as mentioned above, the list of all the outputs and all the inputs of the transaction, and also for example the amount of fees paid for the transaction (txInfoFee) or the amount of new tokens minted (txInfoImint)


```
TxInfo
    txInfoInputs :: [TxInInfo]
    txInfoOutputs :: [TxOut]
    txInfoFee :: Value
    txInfoMint :: Value
    txInfoDCert :: [DCert]
    txInfoWdrl :: [(StakingCredential, Integer)]
    txInfoValidRange :: POSIXTimeRange
    txInfoSignatories :: [PubKeyHash]

    txInfoData :: [(DatumHash, Datum)]
    txInfoId :: TxId
```

Figure 1.3. Some components of the ScriptContext.

Figure 1.4 shows a Bitcoin style spending transaction, and spending UTXOs from private addresses on Cardano works this way as well. The UTXO is represented by the circle in the middle, coming as output from one transaction and going as input into a new transaction. Figure 1.5 shows the spending of a eUTXO sitting at a validator script address (again represented as the circle). For simplicity reasons I will not differentiate between UTXOs and eUTXOs from now on and just call them UTXOs.

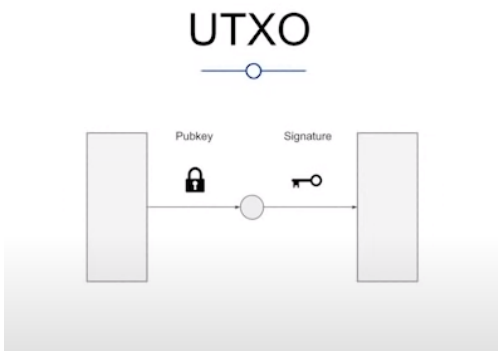


Figure 1.4. From simple private key validation..

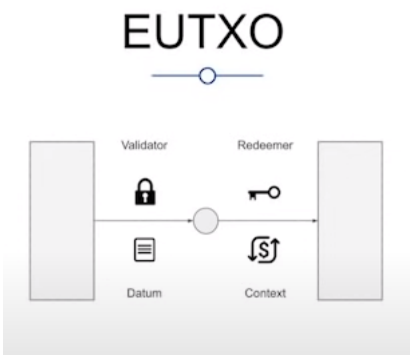
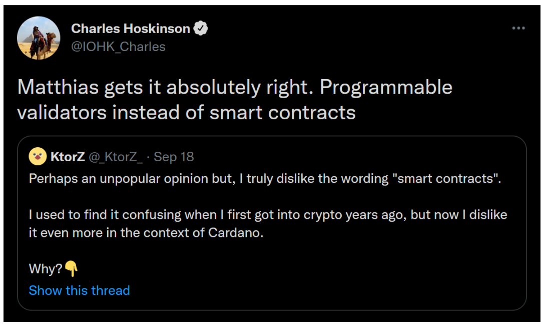


Figure 1.5. ..to arbitrary validation logic.

Another remark about wording in this paper: most of time, the term ’smart contract’ is used to describe programs that are stored on a blockchain and run when certain conditions are met. For clarity reasons I will try to avoid it in this paper, because the term ’smart contract’ is somehow misleading when it comes to Cardano (or any other UTXO style blockchain). Plutus scripts can do nothing else than returning ’True’ or ’False’; they only validate transactions and never initiate an action on the blockchain on their own the way as an Ethereum smart contract can. Such an action (following a successful script evalution) must be done with off-chain code. As Cardano Founder and CEO of the development company of the blockchain Input Outuput Global (IOG) suggested, the term ’programmable validators’ is therefore better suited to describe how Cardano imposes the terms of a contract.



Charles Hoskinson
@IOHK_Charles

Matthias gets it absolutely right. Programmable validators instead of smart contracts

KtorZ @_KtorZ_ · Sep 18
Perhaps an unpopular opinion but, I truly dislike the wording "smart contracts".

I used to find it confusing when I first got into crypto years ago, but now I dislike it even more in the context of Cardano.

Why?👇
Show this thread

Figure 1.6. No more smart contracts.

1.2 The ’concurrency issue’

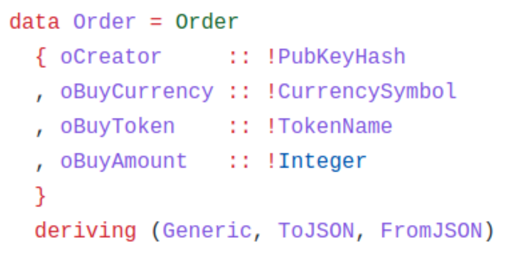
Combining the fact that the state of an Ethereum style smart contract is contained in a UTXO on Cardano (e.g. the ’exchange rate UTXO’ described above) and the fact that a UTXO can only be spent once brings us back to the core of the ’concurrency issue’: what if two users simultaneously want to include the exchange rate contained in the UTXO in their transactions? Only one of them will get the desired input while the other user will have to try building the transaction again one block later. In this concrete case, the ’concurrency issue’ is easily solved: A so-called ’oracle’ that provides the exchange rate as a service can produce (and update) multiple exchange rate UTXOs for multiple users instead of just one, all containing the same exchange rate. However if the UTXO contains as state a liquidity pool instead of an easily reproducible exchange rate, the ’concurrency issue’ becomes more difficult to solve. This is the case for the DEX with automated market maker (AMM) functionality that I will describe in detail below. For now just this: An AMM-DEX UTXO contains two currencies that form a liquidity pool together. How liquidity pools work is described for example in Cryptopedia [5]. One key takeaway from the article is that slippage - the percentage change in the effective price paid in a trade relative to the expected price given by the state of the liquidity pool before the trade - is a big concern in markets/pools with low liquidity. So even though in theory it is possible to split an AMM-DEX UTXO in multiple AMM-DEX UTXOs, each containing a fraction of the two currencies from the original UTXO, this is a bad idea because of increasing slippage.

Given this contention problem over a liquidity pool, a DEX on Cardano could try to avoid having to deal with such a global state altogether and instead implement the orderbook pattern that is traditionally used by centralized exchanges.

The orderbook pattern An order book is basically a list of open buy and sell offers for specific amounts of currency and every buy order needs to be matched with a sell order. IOG describes the order book pattern and Cardano in their docs [6] as a natural fit: ”Every order is a single UTXO, and matching a set of orders means building a transaction that spends the relevant UTXOs. The UTXOs are script UTXOs with a known address and a datum value that holds the quoted price and some bookkeeping information (for example, an address to pay the money to, and an expiration date). The currency value locked in the UTXO is the ’inverse’ of the order.”

While there were lots of discussions in Cardano forums about the development progress of different AMM-DEX projects, the until then little known Muesliswap DEX [7] went live on the mainnet at the end of November using the order book pattern. Three weeks later, it had already approached the number of 50’000 transactions with its backbone smart contract, making it the most used contract in the still short Cardano ’Alonzo’ era network history.

As suggested by IOG, a UTXO paid to the Muesliswap contract contains in the Order(Datum) the name and quantity of the coin the user wants to buy (see Figure 1.7). As value, the UTXO contains the amount of the currency the user wants to sell in exchange for his buy offer. If an independent matchmaker then finds an ’inverse’ offer UTXO in the ledger, he can submit a transaction with both offer UTXOs as inputs. In order for the transaction to be successful, the ’FullMatch’ part of the ’mkOrderValidator’ shown in Figure 1.8 must evaluate to True for both UTXOs. The central function in it being ’correctFull’: It checks that the transaction contains outputs to the two public keys defined in the OrderDatum of the order UTXOs. The CancelOrder Redeemer can be triggered with only one input and comes with only one condition: only the party that has created the UTXO can spend it (i.e. transfer it back to his own wallet). This means that as long as no one else includes the order in a swap transaction with the redeemer (called OrderAction in mkOrderValidator) ’FullMatch’, it can be cancelled by the owner.



```haskell
data Order = Order
  { oCreator     :: !PubKeyHash
  , oBuyCurrency :: !CurrencySymbol
  , oBuyToken    :: !TokenName
  , oBuyAmount   :: !Integer
  }
  deriving (Generic, ToJSON, FromJSON)
```

Figure 1.7. Setting the terms for a currency exchange.

```haskell
{-# INLINABLE mkOrderValidator #-}
mkOrderValidator :: OrderDatum -> OrderAction -> ScriptContext -> Bool
mkOrderValidator od redeemer ctx = case redeemer of
  CancelOrder ->
    traceIfFalse "signature does not match creator in datum" checkSig
  FullMatch ->
    traceIfFalse "expected creator to get all of what she ordered" correctFull
      && traceIfFalse "only matches of pairs of orders allowed" twoParties
```

Figure 1.8. Conditions given by the validator script.

Also the Sundaeswap team argues, that ”an order book model for an exchange, which on Ethereum is disastrously expensive to maintain and update, seems more fundamentally suited to Cardano”. But they see a potential liquidity problem, meaning that no trades can be made because sell and buy offers are too far apart. ”Most pools on decentralized exchanges are thinly traded”, they write in a blogpost [8]: ”Uniswap, for example, has over 8,000 trading pairs. If we rank the pools by trading volume and examine 30, we see only a couple of trades per hour. Consequently,we felt that a pure orderbook without the support of an AMM would be a poor fit.”

Automated Market Maker Automated market makers (AMM) try to solve the problem of illiquid markets by incentivizing users to provide liquidity for a share of the trading fees. The bigger the liquidity, the less slippage will occur, as mentioned above. But even with very low liquidity, trading is always possible whereas an order book would maybe find no matches at all. Protocols like Bancor or Uniswap and basically all DEXs on account style blockchains use AMMs.

1.3 Outlook

Given the two main architectures, I decided to implement a DEX that allows concurrent swaps based on the AMM model. In the following chapters I will introduce my solution to the ’concurrency issue’, which is based on a simple batching mechanism first presented by ”Meld” in a Medium article [9] in mid 2021. The general idea is the following: Instead of letting the users interact directly with the contested liquidity pool state, the application allows them to make reservations. These reservations then get aggregated into one transaction involving the liquidity pool state.

In the following chapter, I will first briefly introduce an AMM DEX that doesn’t address the ’concurrency issue’. Then I will show how to go from there to a concurrent DEX by introducing a reservation layer between the user and the swap transaction. In chapter 3, I will describe this reservation layer in detail, e.g by showing what conditions must be satisfied to make or to retrieve a reservation. In chapter 4 I then present the swap layer which contains the core logic of my application: It validates the swap of the aggregated reservations and makes sure that all swap participants receive the correct amount given the state of the liquidity pool. While chapters 3 and 4 describe on-chain code, in chapter 5 I introduce the importance of off-chain code for decentralized applications on Cardano. I also address the question whether applications on Cardano can be considered as truly decentralized even though they rely heavily on off-chain code.

Chapter 2

Dex-Design

As (non-concurrent) basis for my project, I took the already existing application presented in a lesson of the Plutus Pioneer Program [10] organized by Lars Brünjes, director of education at IOG. The app uses several Uniswap modules [11] from the Plutus library that I used as well, adjusted according to my needs and renamed as Swap modules in the src directory of my project.

The functioning of the original DEX is summarized in Figure 2.1. Tx2 describes the creation of a liquidity pool with currencies A and B. The DEX lets an arbitrary user create such a liqudity pool - this pool technically just being a UTXO sitting at the pool script address as described above. Bob can spend the PoolAB UTXO (resp. use it in his transaction) by fulfilling the conditions given by the PoolAB validator script. The most important part of the validation logic being that during the spending transaction (Tx3), a new UTXO must be created at the same address and that the amounts of Currency A and B contained in this new UTXO must be high enough so that their product is bigger then the product of the amounts of A and B in the old Pool UTXO. The validation logic follows the Uniswap protocol [12] which basically states that the product of currency A and currency B must slightly increase during a Swap, the increased value representing fees for the liquidity providers.

In this concrete example, The product A x B in the UTXO is 1’000 x 2’ 000 = 2’000’000 before the execution of Tx3. After Tx3, which adds 100A to the pool and removes 181B from the pool, the product in the new UTXO is 1’100 x 1’819 = 2’000’900. So, Bob submits Tx3 with - as inputs - the PoolAB UTXO and a UTXO from his own wallet containing 100A. These 100A ’flow’ into the new PoolAB UTXO which in turn allows Bob to decrease the amount of CurrencyB in the pool by 181B. Changing the amounts of currencyA and currencyB in the pool of course also affects the exchange rate offered by the DEX. Before the swap, the rate B/A is equal to 2, after the swap B/A is 1,6536.

Tx4 and Tx5 in Figure 2.1 add respectively remove liquidity from the PoolAB, Tx6 closes the liquidity pool (under the condidtion that nobody except the user executing the transaction has any liquidity left in the pool)

The ’concurrency issue’ in this DEX is the fact that only one user can spend the PoolAB UTXO. Of course, the validation logic forces this user to create a new UTXO containing the updated state for the next user, but this new PoolAB UTXO is only available once the block containing Bobs transaction is added to the blockchain. This limits the DEX usage to one swap per block, with Cardano having a block interval of about 20 seconds.

Reserving instead of swapping To bring concurrency into this system, an additional layer can be introduced in front of Tx3, which I will call reservation layer in this paper. Thanks to

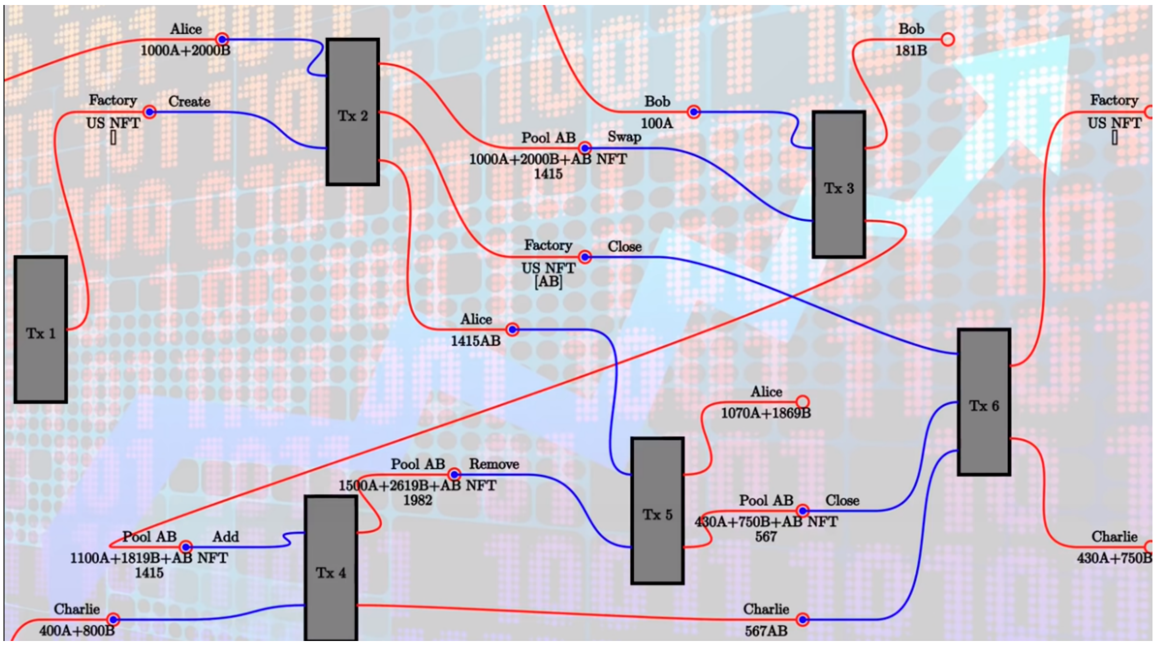


Figure 2.1. Creation (Tx2), use (Tx3), manipulation (Tx4, Tx5) and closure (Tx6) of a liquidity pool.

this layer, users don't interact directly with the Swap state and thus with the UTXO sitting at the address of the Pool validator script anymore. Instead, they create swap orders without touching the PoolAB UTXO. A batcher then combines these swap orders into one final transaction. As mentioned above, this Idea was first presented by "Meld" [9]. Figure 2.2 shows these two layers, respectively transaction phases. Same as in Tx3 in Figure 2.1, the 'Apply' transaction takes as input the State (called PoolAB in Figure 2.1). But instead of just Bob's private UTXO, there are now several inputs, representing the swap orders coming out of several Reserve transactions.

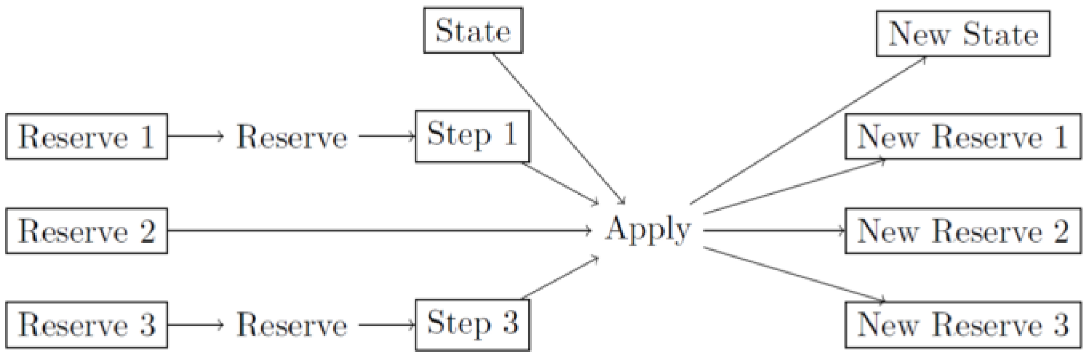


Figure 2.2. Including multiple swap orders into one 'Apply' transaction.

Figure 2.3 shows how these inputs are produced in earlier blocks. In Block 2, Reserve 1 is spent to create Step 1; in Block 3, Reserve 3 is spent to create Step 3; in Block 4 finally, Step 1 and Step 3 together with Reserve 2 and State from Block 1 are spent as part of the Swap transaction. The outputs of this transaction will be used as inputs for the next Swap to repeat the cycle.

In the reservation layer (left part of Figure 2.2), users can spend Reserve UTXOs and convert them into Steps (as done in Block 2 and Block 3 in Figure 2.3). In my DEX and so from now on I will call the 'Reserve' a unused reserve UTXO and the 'Step' a used reserve UTXO.

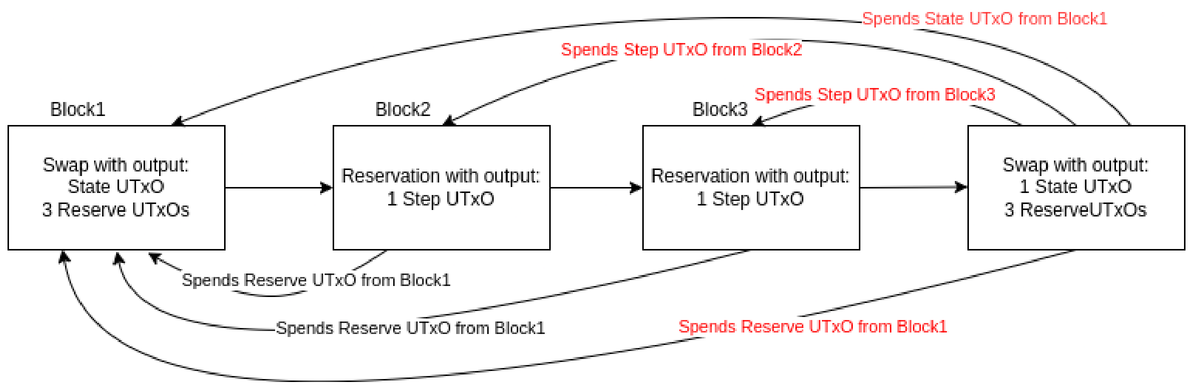

Spends State UTxO from Block1
Spends Step UTxO from Block2
Spends Step UTxO from Block3
Block1
Block2
Block3
Swap with output:
State UTxO
3 Reserve UTxOs
Reservation with output:
1 Step UTxO
Reservation with output:
1 Step UTxO
Swap with output:
1 State UTxO
3 ReserveUTxOs
Spends Reserve UTxO from Block1
Spends Reserve UTxO from Block1
Spends Reserve UTxO from Block1

Figure 2.3. Getting the transaction inputs from previous blocks.

It is important to note that the batcher (i.e. the party that executes the 'Apply' transaction) cannot only include used reserve UTXOs in the transaction but must also include all the unused ones. This guarantees that no reservations are left out of the transaction and thus makes the swap deterministic. This idea was explained in the above-mentioned Medium article [9]. A weakness of this architecture however is that it leads to unnecessarily large transactions volumes if a lot of unused reserve UTXOs have to be included into swap transactions repeatedly. I addressed this problem by making the number N of Reserve UTXOs variable with a simple mechanism that I will describe later in the paper.

The 'Command' datatype below describes the command line interface of the Dex application. In my implementation, I have added the functionalities 'Reserve', 'Retrieve' and 'ReserveFunds', while the first seven commands were already available in the original IOG version (and shown in Figure 2.1). 'ReserveFunds' doesn't trigger a transaction, it only looks for used reserve UTXOs on the blockchain and reports back how many there are and the total value of currencies to swap they contain. 'Reserve' takes as parameters two amounts (Integers) and two currencies (Chars). Executing this command will create a used reserve UTXO containing amount int1 of currency char1 and amount int2 of currency char2. Typically, one of these two amounts will be zero as it makes no sense swapping A against B and B against A at the same time. 'Retrieve' takes as arguments two currencies (chars). This command cancels a Reservation if there is a used reserve UTXO and if the transaction is initiated by the creator of the used reserve UTXO. Retrieve only cancels one reservation, so if a user made several reservations and wants to cancel all of them, he has to submit several retrieve transactions.

```
data Command =
       Funds
     | Pools
     | Create Integer Char Integer Char
     | Add Integer Char Integer Char
     | Remove Integer Char Char
     | Close Char Char
     | Swap Char Char
     | Reserve Integer Char Integer Char
     | Retrieve Char Char
     | ReserveFunds
```

So basically, the first seven commands initiate actions in the Swap layer, while the last three affect the Reservation layer. I will describe the two layers in the following two chapters, starting with the Reservation layer.

Chapter 3

Reservation layer

As already described above, a reserve UTXO can either be used or unused. This state is encoded in the Datum of the UTXO, called ReserveDatum. Addtionally, the Datum tells which liquidity pool the reserve UTXO is made for, how many reserve UTXOs are currently in circulation (encoded by the Integer) and - if the state is 'Used' - which public key made the reservation.

```
data ReserveDatum = Unused LiquidityPool Integer
                    | Used LiquidityPool PubKeyHash Integer
```

The Datum of the reserve UTXOs in Block 1 in Figure 2.3 is 'Unused' while the reserve UTXO produced by the transaction in Block 2 has the Datum 'Used'. Depending on the state of the Reserve UTXO, different actions are possible. These possible spending actions are checked by the reserve validator script and defined by the ReserveRedeemer:

```
data ReserveRedeemer = Reserve | Retrieve | Include | Destroy
```

As the Reserve validator script below shows, a Reserve UTXO in the 'Unused' state can either be reserved, included or destroyed. A Reserve UTXO in the 'Used' state can be retrieved or included. When a new pool is created, a fixed number of Reserve UTXOs must be created in the same transaction and they are all in the 'Unused' state. After having been used and included in a swap, the state of all the Reserve UTXOs is 'Unused' again. The last two conditions are not set by the reserve validator script but by the swap validator script that I will describe in the next chapter.

```
mkReserveValidator u rs ps reserveDatum reserveRedeemer ctx =
    case reserveDatum of
        (Unused lp n) -> case reserveRedeemer of
            Reserve ->
                outputContainsFee     &&
                outputHasToken        &&
                correctLiquidityPool  &&
                correctNrReserveTokens
            Include ->
                poolStateCoinIncluded
            Destroy ->
                uniswapIncluded &&
                outputStateUnused outputDatum
        (Used lp pkh n) -> case reserveRedeemer of
            Retrieve ->
                (Validation.txSignedBy info pkh) &&
                outputHasToken                   &&
                outputStateUnused outputDatum    &&
                correctLiquidityPool             &&
                correctNrReserveTokens
            Include ->
                poolStateCoinIncluded
```

In the following paragraphs I will describe the different redeemer actions Reserve, Retrieve, Include and Destroy one by one.

Reserve To swap A against B, a user creates a used reserve UTXO containing the amount of A he wants to swap and the hash of his public key so that the party performing the swap knows to what address to pay the swapped amount of B to. This is the 'Reserve' action described in Figure 2.2. Figure 3.1 shows a transaction including a Reserve spending action in detail with all the inputs and outputs. This transaction must satisfy the conditions given by the mkReserveValidator function (code above, lines 5 to 8):

- outputContainsFee: The Fee in Ada must be part of the Value of the output reserve UTXO. During the swap transaction, this fee will be paid to the wallet of the user performing the swap.
- correct liquidity pool: Because all Reserve UTXOs from all existing liquidity pools have the same Reserve Address, The lp value in the ReserveDatum associates them with the correct liquidity pool. So, to make a reservation for PoolAB, the user needs to attach a datum with the lp value 'AB'.
- correct number of reserve tokens N: the datum also keeps track of the number of reserve UTXOs currently in circulation. By making a reservation, the user must not change this number. The number of reserve UTXOs only changes during the swap transaction: It increases if the reserve UTXOs are heavily used and decreases if the Reserve UTXOs are not much used.
- It is maybe noteworthy that the Reserve validator doesn't force the user to create a 'Used' Reserve UTXO or to provide his public key in the ReserveDatum, as by not doing so it would be to his own disadvantage.
- outputHasToken: Every Reserve UTXO contains a ReserveCoin in it's value. N of these Coins are minted when the pool is created and burned when the pool is closed. Coins to identify UTXOs are very common on the Cardano blockchain. The reason is that validation only takes place when a UTXO is spent. The creation of a UTXO at a script address on the other hand comes with no restrictions. So nothing can stop a user from creating additional Reserve UTXOs for a certain pool. But having a potentially infinite number of Reserve UTXOs would make it impossible to demand that all reserve UTXOs must be included in the swap transaction in order to make it deterministic. So, instead of collecting all the Reserve UTXOs, the swap transaction collects all the Reserve UTXOs containing a ReserveCoin.

```
ownOutput :: TxOut
ownOutput = case [ o | o <- getContinuingOutputs ctx ] of
    [o] -> o
    _   -> traceError "expected only one Reserve output"

outputHasToken :: Bool
outputHasToken = isUnity (txOutValue ownOutput) rs
```

The first function inspects the transaction context for outputs to the same script address as the address of the UTXO currently being validated (the reserve script address in this case). There must only be one such output, meaning that the validator doesn't allow transactions with two or more simultaneous reservations. The second function checks if the discovered output contains the above mentioned ReserveCoin (rs).

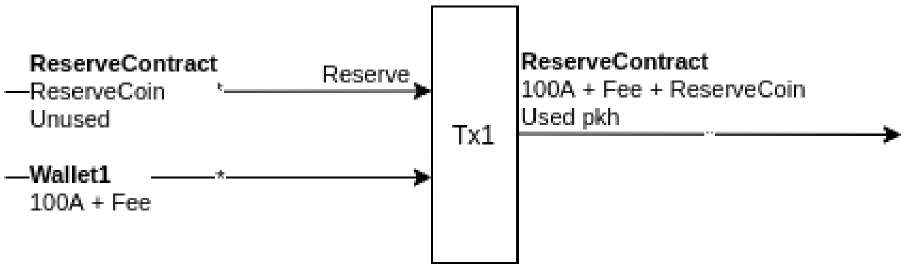


Figure 3.1. Creating a used reserve UTXO.

Retrieve The validation conditions for the 'Retrieve' action are much simpler: The reserve UTXO must be in the 'Used' state as it wouldn't make sense to retrieve a unused reservation. And the public key hash in the ReserveDatum must be the one from the users wallet performing the retrieve action. This guarantees that only the creator of a reservation can retrieve it. Additionally, the user is forced to create a new 'Unused' Reserve UTXO containing the ReserveCoin so that no reserve UTXOs are missing for the next Swap. correctLiquidityPool and correctNrReserveTokens are checked for the same reasons as in a Reserve action: The newly created unused reserve UTXO must be associated to the same liquidity pool as the old one and track the same number of reserve UTXOs in circulation.

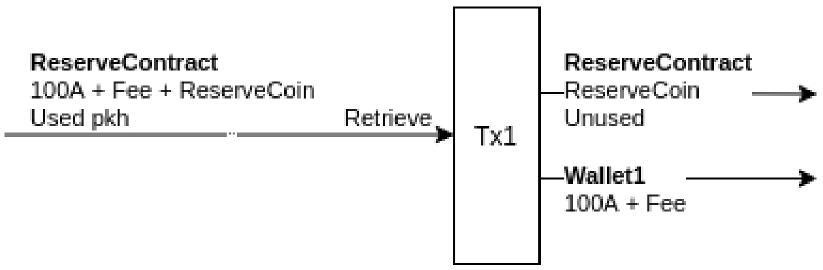


Figure 3.2. Cancelling a reservation and recreating a unused reserve UTXO.

Include The most important action is obviously 'Include'. It can be performed whether the UTXO is 'Used' or 'Unused' and comes - maybe surprisingly - with only one condition: poolStateCoinIncluded. The reason for this is that the validation is delegated to the poolAB. The delegation works as follows: In the same way that a ReserveCoin identifies a Reserve UTXO, a poolStateCoin identifies a Pool UTXO. So by making sure that the poolStateCoin is part of the transaction, we also guarantee that the Pool UTXO is spent in that transaction. And the Pool UTXO can only be spent by satisfying its own validation logic (described in the following chapter). So by requiring the presence of the poolStateCoin, the Reserve validator script delegates the 'Include' validation to the Pool validator script.

This makes sense also for efficiency reasons. As IOG writes in a blog post [13]: 'When adopting such a batching pattern, one should bear in mind that, whenever N orders sitting at the request script are consumed within a single transaction, the request script will be executed N times on transaction submission.' Meanwhile, there is only one single state given by the poolAB UTXO, and so the Pool validator script is only executed once during the swap transaction.

Destroy The fourth and last possible action for a Reserve UTXO is 'Destroy'. This action can only be performed if the UniswapCoin is present in the same transaction. So again, this is a case of delegation of validation logic, this time by making sure that a 'PoolFactory' UTXO containing the UniswapCoin is spent in the same transaction. Above, I used the term 'Pool validation logic' when talking about the spending of the 'Pool' UTXO while in the present context I'm talking about the spending of a 'PoolFactory' UTXO. Under the hood though, there is only one validator script (shown in simplified version below) for both of them. The Factory and the Pool UTXOs are UTXO instances sitting the same address, but differentiated by their Datum (either being 'Factory' with a list of liquidity pools or 'Pool' with a specific liquidity pool and an amount of liquidity coins). Because the 'PoolFactory' instance contains as value the UniswapCoin (us, passed into the validation logic in lines 8, 10 and 11 below), it can be spent with the Redeemers 'Create' or 'Close' as shown in the simplified code below. For a UTXO not containing this coin, validateCreate, validateCloseFactory and validateClosePool would not validate.

```
mkUniswapValidator :: Uniswap
                   -> UniswapDatum
                   -> UniswapAction
                   -> ScriptContext
                   -> Bool
mkUniswapValidator us Factory Create = validateCreate us
mkUniswapValidator _  Pool    Swap   = validateSwap
mkUniswapValidator us Factory Close  = validateCloseFactory us
mkUniswapValidator us Pool    Close  = validateClosePool us
mkUniswapValidator _  Pool    Remove = validateRemove
mkUniswapValidator _  Pool    Add    = validateAdd
mkUniswapValidator _  _       _      = False
```

So, coming back to the Reserve UTXOs: given the delegation of validation, they can and must be destroyed - respectively their ReserveCoins must be burned - when a Pool is closed and so this condition is included in the 'validateClosePool' function. Burning the coins is important, because otherwise lingering ReserveCoins would exist that could be used to create additional valid Reserve UTXOs. A second condition for destroying reserve UTXOs, also given by the mkUniswapValidator, is that the state of the reserve UTXOs being 'destroyed' must be unused. otherwise, the user that made the reservations would loose his money.

Chapter 4

Swap layer

Swap After having described the reservation layer, I'm coming now to the second layer, in which the 'Apply' transaction from Figure 2.2 takes place. As already mentioned, the (used and unused) reserve UTXOs must be included in this transaction with the redeemer 'Include'. And as also mentioned, the validation logic is delegated to the spending of the pool UTXO through the 'poolStateCoinIncluded' function. Figure 4.1 shows all the inputs and outputs of this transaction.

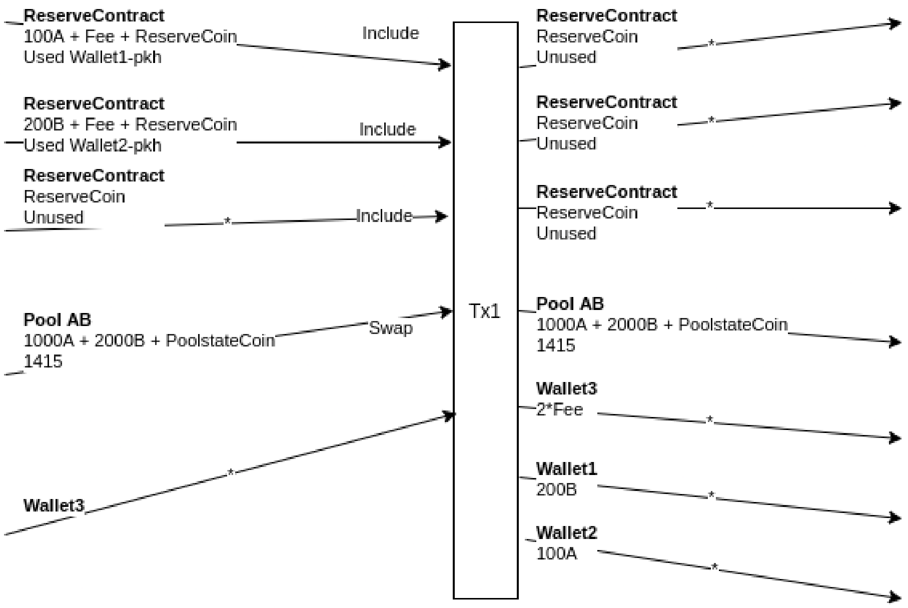


Figure 4.1. Swapping the orders made by Wallet1 and Wallet2.

As Figure 4.1 shows, the Pool UTXO is spent with the 'Swap' redeemer. This triggers the 'validateSwap' function, the ninth line of the mkUniswapValidator script above shown here in more detail:

```
mkUniswapValidator _  c r (Pool lp _) Swap ctx = validateSwap r lp c ctx
```

To evaluate to True, validateSwap must fulfill the following conditions:

```
allClientsReceiveCoins        &&
allReserveOutputsPresent      &&
allReserveInputsPresent       &&
checkSwap oldA oldB newA newB &&
(isUnity inVal c)             &&
(isUnity outVal c)            &&
noUniswapMinting
```

- allReserveInputsPresent: This condition makes sure that all the reservations - whether they are 'Used' or 'Unused' - are included in the swap transaction. The function gets the correct amount of reserve UTXOs from the Datum of any of the reserve UTXOs.
- allReserveOutputsPresent: Part of the swap validation is a mechanism to determine the number N of Reserve UTXOs for the next reservation round. This is done by looking at the ratio r of the used number to the total number of reserve UTXOs. If r is above 0.7, an additional ReserveCoin is minted and included (in an additional Reserve UTXO) in the transaction output. If the ratio is below 0.3 and there are more then 3 reserve UTXOs in the transaction input, a ReserveCoin is burned. The allReserveOutputsPresent function checks that the transaction output contains exactly the new number of reserve UTXOs (nrNewRes).

  ```
  nrNewRes
      | nrUsedRes/nrOldRes >= .7 = nrOldRes + 1
      | nrUsedRes/nrOldRes <= .3 = max 3 (nrOldRes - 1)
      | otherwise                = nrOldRes
  ```

  This is a very simple way of adjusting the number of reserve UTXOs. It could of course be done in a more sophisticated way by determining the newNrReserveTokens based on the information of multiple previous rounds instead of just one.
- allClientsReceiveCoins: This is really the core function of my whole program. It makes sure that everyone that made a reservations will receive the correct amount out of the swap transaction. To validate, the following function must evaluate to True:

  ```
  allClientsReceiveCoins = valuesToPay == valuesPaid
  ```

  'valuesPaid' is a list of all the payments to all the PublicKeys observed in the transaction. It is easy to get thanks to the ScriptContext parameter that contains this information.

  'valuesToPay' is more difficult to construct. The goal for this function is to be deterministic (meaning that given a set of inputs, it always delivers the same output), so that the party building the swap transaction cannot manipulate it. A manipulation could be done for example by first applying all the sell reservations of currencyA to the liquidity pool and thus increasing the relative amount of currencyA in the pool (resp. lowering the relative value of currencyA). Then, the swap executor could first apply his own reservation for buying A before all the others, thus getting currencyA for lower price.

  Meld suggests in the above mentioned article [9] applying timestamps to the reservations and then applying them to the liquidity pool in the order given by these timestamps.

  In my implementation though, the reservations are not applied to the liquidity pool one by one. Instead, the first step is to calculate the difference of the total inputs of currenies A and B, 'SwapInputAmountA x r - SwapInputAmountB'. r is the ratio of 'totalAmountB / totalAmountA' in the liquidity pool before the transaction and thus the current exchange rate. And so 'SwapInputAmountA x r' is the value of the input of CurrencyA expressed in terms of CurrencyB. The second step is to apply 'SwapInputAmountA x r - SwapInputAmountB' to the liquidity pool if the difference is positive (meaning that the value of all CurrencyA inputs is bigger then the value of all CurrencyB inputs). Otherwise, 'SwapInputAmountB - SwapInputAmountA x r' is applied.

The following function returns this amount, the second amount being equal to zero in each case.

```
amountForSwapInput :: Integer -> (A, B) -> (A, B)
amountForSwapInput r (a,b)
    | a * r > b = (a - b / r, 0)
    | otherwise = (0, b - a * r)
```

So only either CurrencyA or CurrencyB is applied to the liquidity pool (or neither, in case of TotalCurrencyA * r = TotalCurrencyB). If CurrencyA is applied, that means that the total value of all the CurrencyA inputs (in terms of CurrencyB) is bigger then the total value of all the CurrencyB inputs. So the users that made reservations to sell CurrencyA will get all the CurrencyB inputs plus the result of the swap. The users that made reservations to sell CurrencyB just get the amount TotalCurrencyB / r from the TotalCurrencyA input.

The result of this logic are the following functions:

```
totalValA' :: Amount A
totalValA'
    | newA - oldA > 0 = totalValB / r
    | otherwise       = (oldA - newA) + totalValA

totalValB' :: Amount B
totalValB'
    | newA - oldA > 0 = (oldB - newB) + totalValB
    | otherwise       = totalValA * r
```

totalValA' is the amount that will be distributed between the users swapping CurrencyB for CurrencyA and totalValB' is the amount that will be distributed between the users swapping CurrencyA for CurrencyB. oldA, newA, oldB and newB are the values in the liquidity pool before and after the swap.

In the transaction shown in Figure 4.1, 'totalValA * r' is equal to 'totalValB' (the exchange rate r, given by 1000A and 2000B, is 2, so the 100 A are worth 200 B). This means that the inputs can be swapped directly. The suppliers (in this case only one, wallet1) of the 100 A get the 200 B and the suppliers of the 200 B (wallet2) get the 100 A. The PoolAB UTXO must of course be included in the transaction anyway as it provides the exchange rate r and as Reserve UTXOs can only be spent in the presence of a PoolStateCoin.

Once the total amount of A and B in the output is known, it must be made sure that the providers of CurrencyA get the correct share of the output of CurrencyB and vice versa. This is done by calculating the percentage of the total input of one currency provided by a user and guaranteeing that he gets the same percentage of the output of the other currency.

For example: If User1 supplies 10 percent of the total input of currencyB, he must get 10 percent of the total output of currencyA.

One problem with this implementation was that the equality condition 'valuesToPay == valuesPaid' doesn't hold if the party performing the Swap transaction himself also made reservations. This is because the value paid to the public key of the party performing the swap includes the fees paid by the other users. To solve the problem, reservations from a public key performing the swap are filterd out before the 'allClientsReceiveCoins' function is run.

The logic behind this is the following: The validator guarantees certain conditions for all third party users paying money to the script address. Once these conditions are satisfied by a transaction and there is still money left to spend, this 'excess' amount goes automatically to the wallet of the party submitting the transaction. In the case of the swap transaction this means that once every user gets a UTXO with the percentage of the output he is entitled to, to party performing the swap can take all the rest (thus including the fees). If he doesn't agree with this output, he simply doesn't submit the transaction.

- The four remaining conditions (checkSwap, isUnity inVal c, isUnity outVal c and noUniswapMinting) were given by the original Uniswap script, with checkSwap being the implementation of the above mentioned UniswapV1 protocol. The values for the arguments oldA and oldB are coming from the Uniswap UTXO being spent in the transaction and the values of newA and newB from the Uniswap UTXO being created in the transaction. Getting the amountForSwapInput for one currency (the other being zero) from the function described on page 16, the app calculates how much of the second currency it can take out of the liquidity pool without invalidating the checkSwap function.

Create/Close As already mentioned, the transaction creating a Pool UTXO must also create the correct amount of Reserve UTXOs. For simplicity reasons, I have hardcoded this number to three in the validator script:

```
amountOf minted r == 3
```

The number of ReserveCoins that must be burned when a pool is closed cannot not be hardcoded because it gets adjusted during every swap transaction in its lifespan. The final number of ReserveCoins is encoded in the Datum of every reserve UTXO and can be retrieved easily:

```
amountOf minted r == negate nrReserveTokens
```

The remaining redeemer actions Remove and Add, invoked to add/remove liquidity to/from a pool are not impacted by my implementation and so I will not describe them here.

Chapter 5

On-chain and off-chain components

A potential flaw making my DEX completely unusable would be to not set a maximum amount for the reservations. I describe this problem in detail because it illustrates very well potential limitations coming with the fact that script validation takes place in the narrow transaction context (ScriptContext). The problem is the following: If the total amount of a currency locked in the reserve UTXOs is bigger then the amount of that currency in the liquidity pool, the swap cannot happen because the amount of a currency in a liquidity pool must never become negative (nor zero, as the product of the two currencies must increase with every swap). At the same time, the swap validator doesn't allow the swap transaction to leave aside any reservations as the inclusion of all the reservations is mandatory to make the swap deterministic. So unless enough reserve UTXOs get cancelled by the users or the amount of the currency in the liquidity pool increases, the swap transaction can never happen.

The easiest solution for this problem would be to just limit the amounts of currencyA and currencyB a user can put into a reserve UTXO to the amounts of

```
oldA/nrReserveTokens
oldB/nrReserveTokens
```

respectively, where oldA and oldB are the amounts of A and B in the pool before the swap. Given these conditions, all reservations together could never exceed the amount contained of the liquidity pool of either currency. But: The PoolAB UTXO that holds the information about oldA and oldB is not included in the reserve transaction. So, because the validator only sees the scriptContext and not the whole ledger, the conditions cannot be checked. Including the PoolAB UTXO is of course no option - as not needing it for every reservation is what made this DEX concurrent in the first place.

The problem can be solved, though, with minimal on-chain computation. Instead of checking the conditions in the reservation layer, all my DEX implementation does is filter out the used reserve UTXOs (each containing amtA of currencyA and amtB of currencyB) during the swap validation with the following lambda function:

```
(amtA, amtB) -> amtA <= oldA/nrReserveTokens &&
                amtB <= oldB/nrReserveTokens
```

The idea behind this is that the swap validator guarantees the correct output if the user follows the rule of not reserving more than 'oldA/nrReserveTokens' of currencyA and 'oldB/nrReserveTokens' of currencyB. Making sure that this doesn't happen, meanwhile, is the job of the off-chain code that I haven't described yet.

The off-chain part of a Plutus application is code written in Haskell (while the On-chain part is written in Plutus, a subset of Haskell, and pre-compiled to Plutus core by the GHC plug-in called Plutus Tx). The job of the off-chain code is basically to build transactions that can be submitted to the blockchain and making sure that these transactions will pass the validation logic. The transaction building, of course, can be done with the 'outside' knowledge of the whole ledger state. So it is easy to check the following conditions before submitting the transaction to the blockchain and abort the process by throwing an error if the conditions are not met:

```
when (amountA >= poolAmtA/nrReserveTokens) ||
     (amountB >= poolAmtB/nrReserveTokens)
      $ throwError "reservation exceeds limit"
```

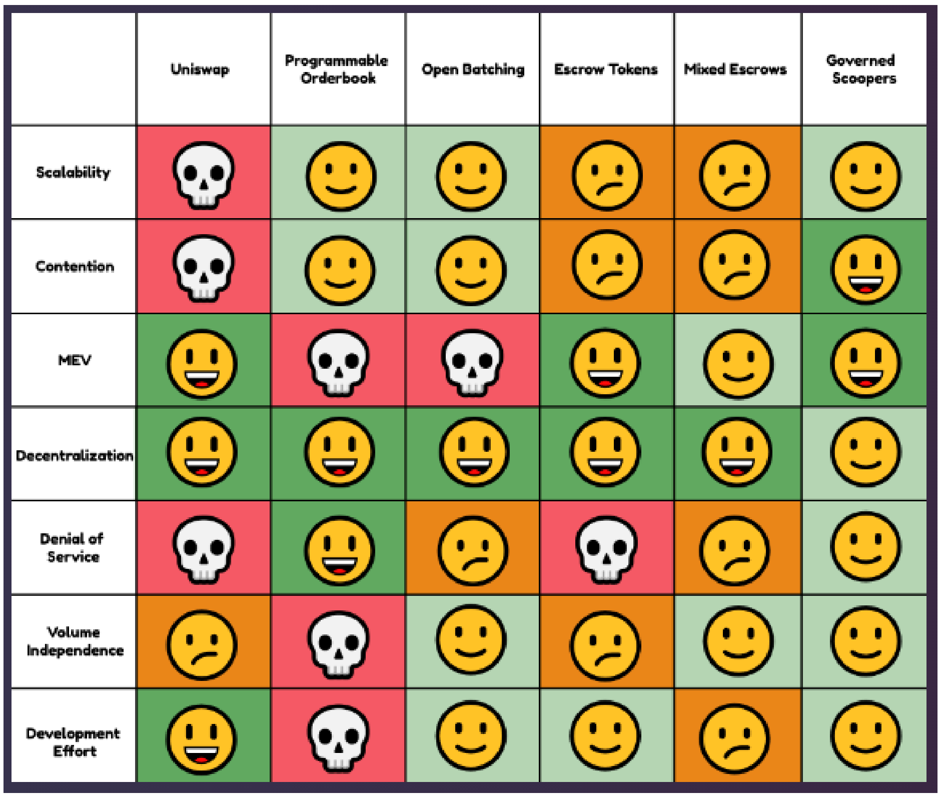
the values of amountA and amountB to be swapped are coming from the uniswap-client (in the app directory of my project), which constructs for the user the cli described on page 9. The uniswap-client submits the command entered by the users via http-request to the uniswap-pab, which is the server component of the Uniswap app. The 'pab' in uniswap-pab stands for Plutus Application Backend [14] and it is - among other things - used to forward user input to the running contract instances defined in the off-chain code. My program submits the transactions defined in the contracts to a simulated blockchain initiated by the Plutus.PAB.Simulator module [15]

The on-chain swap validator script filtering out reservations that exceed a certain amount could be more sophisticated. For example, it could guarantee that users that made reservations exceeding the limit get their money back during the swap transaction. This would come with the advantage of users non having to make sure that the off-chain code (which can be provided by different implementations for the same on-chain code) aborts transactions with reservations that are too big. The disadvantage on the other hand is that there is more on-chain computation which of course is much more expensive then off-chain computation.

Decentralized or not? The example above shows very well the different degrees of decentralization of Cardano apps. One could argue that they are not decentralized at all, since the on-chain part is just a programmable validator that validates transactions and never jumps into action otherwise - so there can be no smart contract that waits for reservations and then, once the are part of the ledger, performs the aggregated swap by itself. But a well written validator guarantees that exactly the intended (and only the intended) swap can be performed by some off-chain code.

If these guarantees are properly encoded in the validator script, users don't need to trust the party performing the swap. That's why Sundaeswap, when comparing different architectures for DEX implementations on Cardano, gave all of them high scores in the category decentralisation (see Figure 5.1). The only model that doesn't get the highest decentralization score, 'Governed Scoopers' is the one they decided to implement. As they explain in a Medium article [16], they first leaned towards the 'Escrow Tokens' model, which is the one I implemented for this thesis. ”Unfortunately”, they note, ”this solution crashes head first into the sizing limits of the Cardano blockchain”, explaining that ”in practice, our first implementation of this was hitting the Cardano protocol limits with 5 or 6 escrow tokens, before even adding in features like governance.”

They decided to sacrifice a little bit of decentralization for better scores in other categories. ”If you can trust your aggregators to consistently and fairly choose which orders to include, you no longer need escrow tokens, for example. You can focus on that contention-less on-ramp, and on protocol composability”, they explain. SundaeSwap will try to establish a ”limited amount of trust in the aggregators” by, as one of the mechanisms, licensing them and giving the SundaeSwap community the power to revoke these licences via a vote.



	Uniswap	Programmable Orderbook	Open Batching	Escrow Tokens	Mixed Escrows	Governed Scoopers
Scalability	💀	🙂	🙂	😕	😕	🙂
Contention	💀	🙂	🙂	😕	😕	😃
MEV	😃	💀	💀	😃	🙂	😃
Decentralization	😃	😃	😃	😃	😃	🙂
Denial of Service	💀	😃	😕	💀	😕	🙂
Volume Independence	😕	💀	🙂	😕	🙂	🙂
Development Effort	😃	💀	🙂	🙂	😕	🙂

Figure 5.1. Scores given to different DEX architectures by Sundaeswap [16].

Chapter 6

Conclusion

Further development The Sundaeswap arguments in the previous chapter show that my DEX implementation is hardly usable as a real world application (in addition to the fact that I didn't use the newest version of the Plutus language and the PAB as these components were still evolving when I started writing this thesis), but it demonstrates how the problem of a single state being under a lot of contention can be solved by introducing an additional layer. This pattern will - in combination with other techniques - surely be used in some AMM-style DEXs appearing on the Cardano blockchain in the coming months.

A quite radical way to avoid triggering the N reserve validator scripts that come with the additional reserve layer while still having the same degree of concurrency was presented in September by IOG in the already mentioned blogpost [13]. The idea is that users only send a notification to the reservation layer and the order itself to their own public key address. All order UTXOs sitting at wallet addresses must contain an 'order' token so that the batcher can identify them and build the aggregated swap transaction. Because the orders are sitting at public key addresses, no validator script has to be run to spend them (the notifications in the reservation layer are only inspected to find the orders, but not spent). This pattern comes with the disadvantage though that all users must be online to sign the aggregated transaction to authorize the spending of their orders.

Additionally, IOG is developing hydra heads [17] as an isomorphic layer 2 solution where DEXs could build their swap transactions in the future. This would solve the problem of the transaction size limit that stopped Sundaeswap from further developing their 'Escrow Tokens' model.

Worth the trouble? Given the difficulties of building a concurrent AMM style DEX on Cardano: Is the UTXO model worth it? My personal opinion is of course of little relevance, especially as I have no experience building DEXs on account-style blockchains. But still, a number of strong arguments in favour of the UTXO model seem obvious to me. First and most importantly: In contrast to f.ex Ethereum, on the Cardano blockchain a correctly built transaction can never fail in mid-script execution. As IOG explains [18], "the success or failure of transaction validation depends only on the transaction itself and its inputs, and not on anything else on the blockchain. As a consequence, the validity of a transaction can be checked off-chain, before the transaction is sent to the blockchain." This means that a transaction only fails if some other transaction concurrently consumes an input that the transaction is expecting - or if the off-chain code is badly written and the transaction is submitted anyway. The first scenario is checked in the phase-1 validation and the second scenario in phase-2 validation [19]. To discourage denial of service attacks through submitting faulty transactions and to guarantee that the validator nodes are compensated for their work in case phase-2 validation fails, a collateral is used. But if already phase-1 validation fails (because the on-chain conditions have changed since the transaction was constructed), the transaction is rejected entirely without any fees or collateral being charged. Another argument for the UTXO model is that the fees for a valid transaction are deterministic because given the same inputs, the validation always produces the same result as the global ledger state does not influence it. This means that transactions on Cardano are not in danger of 'running out of gas' as Ethereum smart contract executions are.

As a last point: Developers might have to find innovative solutions to make their apps concurrent due to the 'local' nature of transaction validation. But the other side of the medal is that this 'local' nature of transaction validation makes a high degree of parallelism possible. "A node could, in principle, validate transactions in parallel, if those transactions do not try to consume the same input", so the above mentioned explainer [18]. "This is great both for efficiency and for reasoning, simplifying the analysis of possible outcomes, and proving that 'nothing bad' can happen."

Bibliography

[1] Minswap, “Minswap testnet reflections.” https://medium.com/minswap/minswap-testnet-reflections-64b01c5e7c45, September 7, 2021.

[2] Minswap, “Introducing laminar — an eutxo scaling protocol for accounting-style smart contract.” https://medium.com/minswap/introducing-laminar-an-eutxo-scaling-protocol-for-accounting-style-sm September 13, 2021.

[3] P. Lanningham, “Sundaeswap fundamentals.” https://sundaeswap.finance/papers/SundaeSwap-2021-06-01-Fundamentals.pdf, June 1, 2021.

[4] M. M. T. Chakravarty, J. Chapman, K. MacKenzie, O. Melkonian, M. P. Jones, and P. Wadler, “The extended UTXO model,” in *Financial Cryptography and Data Security - FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers* (M. Bernhard, A. Bracciali, L. J. Camp, S. Matsuo, A. Maurushat, P. B. Rønne, and M. Sala, eds.), vol. 12063 of *Lecture Notes in Computer Science*, pp. 525–539, Springer, 2020.

[5] Cryptopedia, “What are liquidity pools?.” https://www.gemini.com/cryptopedia/what-is-a-liquidity-pool-crypto-market-liquidity. Last accessed December 2021.

[6] IOG, “What is the order book pattern?.” https://playground.plutus.iohkdev.io/doc/plutus/explanations/order-book-pattern.html. Last accessed December 2021.

[7] Muesliswap, “Muesliswap docs.” https://docs.muesliswap.com/. Last accessed December 2021.

[8] Sundaeswap, “The scooper model.” https://sundaeswap.finance/posts/sundaeswap-scalability, November 4, 2021.

[9] H. N. Quang, “Concurrent deterministic batching on the utxo ledger.” https://medium.com/meld-labs/concurrent-deterministic-batching-on-the-utxo-ledger-99040f809706, September 9, 2021.

[10] L. Brünjes, “Lesson 10, plutus pioneer programm.” https://github.com/input-output-hk/plutus-pioneer-program. accessible at: https://web.archive.org/web/20210626022515/https://github.com/input-output-hk/plutus-pioneer-program/tree/main/code/week10.

[11] IOG, “Uniswap modules.” https://playground.plutus.iohkdev.io/doc/haddock/plutus-use-cases/html/Plutus-Contracts-Uniswap.html. Last accessed December 2021.

[12] H. Adams, N. Zinsmeister, and D. Robinson, “Uniswap v2 core.” https://uniswap.org/whitepaper.pdf, March, 2020.

[13] J.-F. Etienne, “Architecting dapps on the eutxo ledger.” https://iohk.io/en/blog/posts/2021/11/16/architecting-dapps-on-the-eutxo-ledger/, November 16, 2021.

[14] IOG, “Plutus application backend.” https://github.com/input-output-hk/plutus-apps/tree/main/plutus-pab/src/Plutus/PAB. Last accessed December 2021.

[15] IOG, “Plutus pab simulator.” https://playground.plutus.iohkdev.io/doc/haddock/plutus-pab/html/Plutus-PAB-Simulator.html. Last accessed December 2021.

[16] Sundaeswap, “Sundaeswap labs presents: The scooper model.” https://sundaeswap-finance.medium.com/sundaeswap-labs-presents-the-scooper-model-678d6054318d, November 4, 2021.

[17] S. Nagel, “Hydra – cardano’s solution for ultimate layer 2 scalability.” https://iohk.io/en/blog/posts/2021/09/17/hydra-cardano-s-solution-for-ultimate-scalability/, September 17, 2021.

[18] IOG, “Understanding the extended utxo model.” https://docs.cardano.org/plutus/eutxo-explainer. Last accessed December 2021.

[19] IOG, “About the collateral mechanism.” https://docs.cardano.org/plutus/collateral-mechanism. Last accessed December 2021.

Erklärung

Erklärung gemäss Art. 30 RSL Phil.-nat. 18

Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Bern, 25.1.2022
Ort/Datum

[illegible]
Unterschrift