Extended UTxO in production: techniques, trade-offs and a search of better balance

Ilya Oskin
i.oskin@spectrumlabs.fi

November 2023

Abstract

In recent years many novel smart-contract platforms such as Ergo [3], Cardano and Nervos adopted the concept of UTxO introduced in Bitcoin to express state of the ledger. In combination with a scripting language that allows to express predicates on transaction it forms a powerful system of programmable money, a.k.a. Extended UTxO.

In practice, compared to the more adopted [1] systems based on accounts, eUTxO offers a different trade-off between the amount of on-chain computations and the amount of data that has to be transmitted to express state transitions. Thus, coming with its own unique set of approaches, trade-offs and bottlenecks.

In this article we develop an intuition of how to think in eUTxO, analyze our experience of implementing incrementally complex on-chain protocols in terms of eUTxO and try to find a better alternative to eUTxO in terms of balance between on-chain and off-chain.

1 Structure

In section 2 we build an intuition about Extended UTxO paradigm by re-inventing it from scratch and comparing to a more imperative Account paradigm. Then, in section 3, we outline most challenging problems and limitations developers face when building real-word applications on eU-TxO. Finally, we discuss an alternative framework in which we address those limitations in section 3.4.

[1]At the moment as this article is being written.

2 Extended UTxO in essence

As the word "extended" suggests, eUTxO is a continuation of UTxO model introduced in Bitcoin [4]. UTxO relies on immutable data structures (UTxO) that encode pieces of ledger state, those structures can only be created or eliminated. Each structure has a specific locker attached to it. To eliminate a eUTxO, one must prove that he has the key to the locker, typically a signature to prove ownership of the private key corresponding to the public key used a locker.

First attempt to extend this framework with programmability capabilities was done in Bitcoin Script. Simple non Turing-complete scripting language allows to define more sophisticated conditions under which a UTxO can be eliminated: e.g. require signatures corresponding to multiple public-keys, require that the time of transaction is greater than some specified timestamp etc.

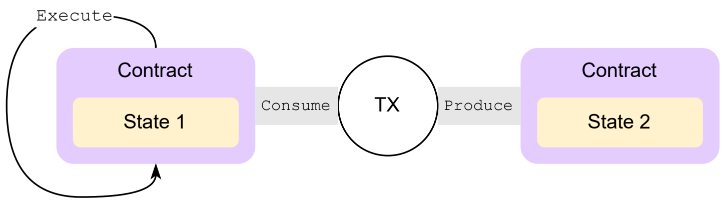

Figure 1: A transaction in UTxO system.

While sufficient for simple applications, Bitcoin Script doesn't allow to define more sophisticated logic required for most useful protocols. The limiting factor is not the lack of Turing-completeness, but the lack of composability, which stems from limited access to the context of the transaction within which the locker script is executed. As shown in [2] an ability to access transaction's inputs (UTxOs that the transaction eliminates) and outputs (UTxOs that the transaction creates) in scripts is sufficient to emulate Turing-completeness even if the scripting language itself is not Turing-complete (i.e. individual scripts don't have loops). Thus, extensive context opened the door for encoding arbitrary on-chain logic by allowing scripts to validate discrete state transitions which can be composed into possibly infinite chain of state transitions 2.

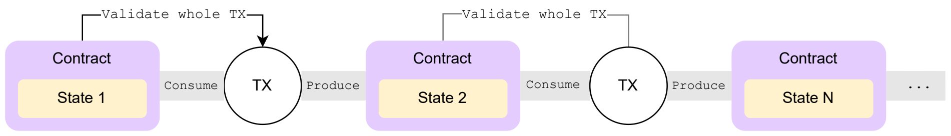

Figure 2: Continous chain of state transitions verified each time the chain extends.

2.1 eUTxO versus Programmable Accounts

It is common to compare eUTxO with its counterpart based on the notion of programmable accounts. Programmable accounts were introduced earlier in [1] and share many similarities with eUTxO. State of the ledger is still comprised of data structures called accounts, each account is owned by a user identified with a public key or a program (smart contract). The difference with eUTxO is that accounts are mutable anf thus, the programmability model is different.

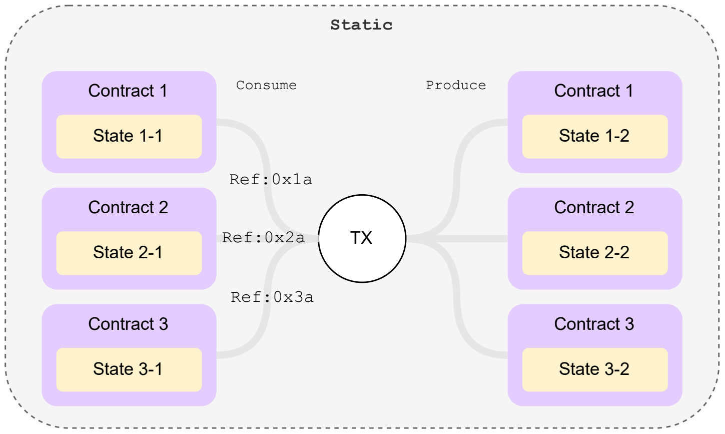

Figure 3: Inputs and outputs of a transaction are defined statically in eUTxO.

While eUTxO scripts serve only as validators of state transitions computed statically 3 and depend only on transaction's local context, most of the stuff with accounts happen in runtime: whole state of the ledger is involved in transactions and is resolved when the transaction is executed (being added to a block) and corresponding mutations to that state are computed based on that states dynamically 4. An ability for one smart-contract to call another in runtime enables composabilty, making it possible for many different protocols to interoperate.

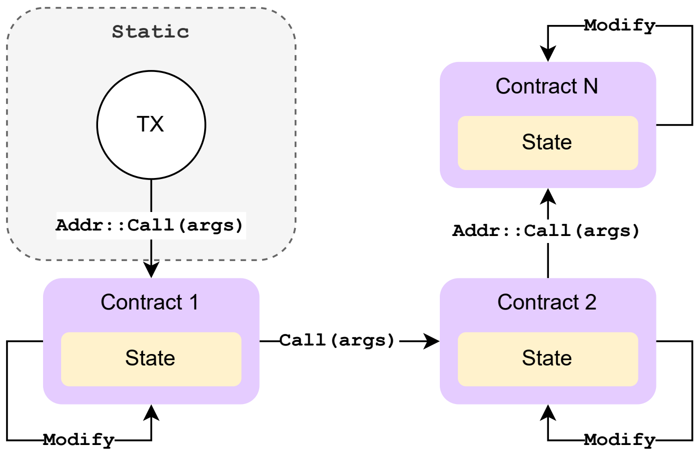

Figure 4: Side effects of a transaction in the programmable accounts model are computed in runtime.

Although the two frameworks discussed above are just different ways of expressing the same things, they come with different trade-offs in terms of utilization of scarce network resources (data and computations) and determinism of computations.

We will use an artificial example of an “petstore” encoded on-chain to illustrate that eUTxO and programmable accounts are two opposite extreme cases of the “resources-determenism” trade-off. The state of the petstore is composed of a balance of the store, and a set of counters indicating how many pets of each type are available 1. To buy a pet for the fixed price the corresponding amount must be deposited to the store.

Balance	Integer
NumCats	Integer
NumDogs	Integer

Table 1: Structure of perstore.

In the case of programmable account the smart-contract will receive the type of the desired pet as an input, fixed amount will be withdrawn from buyer’s account and deposited to store’s balance, the number of available pets of the specified type will be decreased by one, the number of corresponding pets on buyer's will be increased by one.

In the case of eUTxO, initial and final state are known by the time the transaction is formed. Validator script of the store looks up for the next output containing the store's state, checks that the balance of the store was increased by the fixed amount of coins, the counter of the desired pet was decreased by one and the UTxO is guarded with the same script, so the chain of state transitions can be continually validated. Then it looks for buyer's output and validates that it received one desired pet.

The example above demonstrates that in the case of account none of the transaction effects are known until the transaction is actually executed, on the other hand only the minimal amount of data has to be included into the transaction and thus transmitted over the network.

In contrast, in the case of eUTxO, the initial and final states of the petstore are computed in advance, and the validator script only verifies the transition in runtime. The whole updated state of the petstore has to be included into the transaction, although only part of it actually changed.

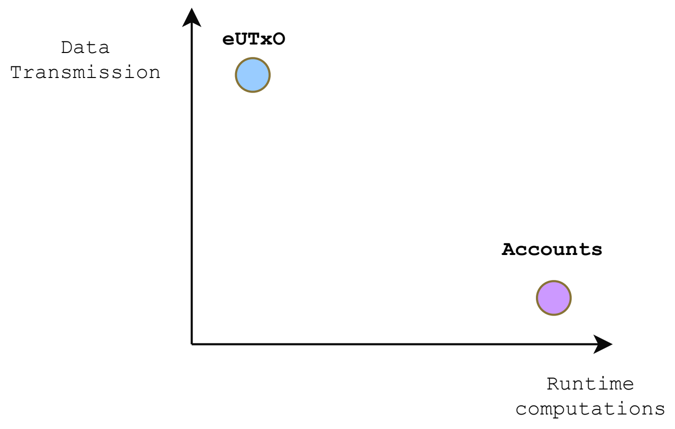

Figure 5: eUTxO is deterministic while requires more data transmission. Account approach is non-deterministic, but requires fewer data transmission.

3 Key challenges and techniques

eUTxO requires developers to think in terms of state transitions and validations. We will use real examples from now on to demonstrate what it means on practice.

3.1 Continuation of states

Continuation of states is the basic technique when it comes to modelling a persistent entity on-chain. We already touched that technique in the petstore example, but omitted many important details. Now we will use AMM liquidity pool [5] as a more realistic example.

Typically, an AMM pool stores two assets and supports three types of operations:

- Deposit liquidity. Both assets in proportion corresponding to the current price in the pool are deposited into it, in exchange pool emits so-called LP tokens, which represent certain share of liquidity in the pool.
- Redeem liquidity. LP tokens are returned to the pool, in exchange the pool returns corresponding share of liquidity in both assets.
- Swap. Exchange one asset for another for the current price according to constant product invariant.

A liquidity pool is a long-lasting entity which lives on-chain. A UTxO, the only construct that we can use to model the pool on-chain, is in contrast ephemeral, i.e. can only be spent once. So pool has to be modelled as a continuation of UTxO, each encoding a discrete pool's state.

To ensure the pool doesn't split or leak its assets illegally during transactions it is common to use a non-fungible token 6.

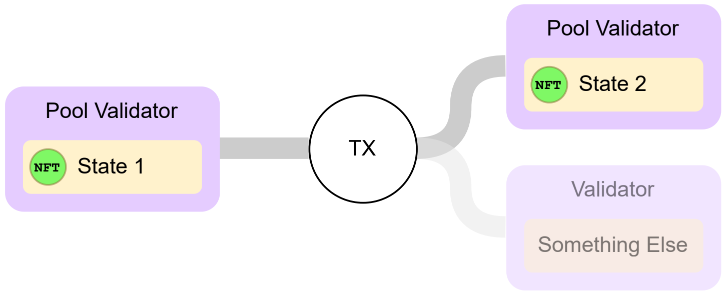

Figure 6: Pool state is uniquely identified with a non-fungible token. Validator must check that the NFT always remains in the pool.

3.2 Shared state

One liquidity pool is used to serve thousands of operations from different users.

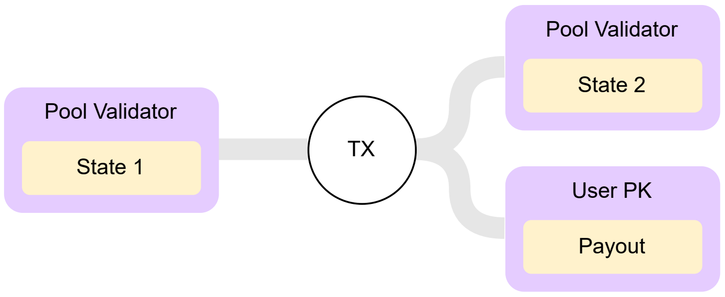

Figure 7: Transaction with AMM poool.

Most straightforward solution would be to perform all operations with the pool directly as shown on 6. In reality, it is not practical. Because of deterministic nature of eUTxO many transaction would refer the same pool state in the event of spikes in demand. Because a UTxO can be spent one once, this race condition would result in cancellation of many transactions.

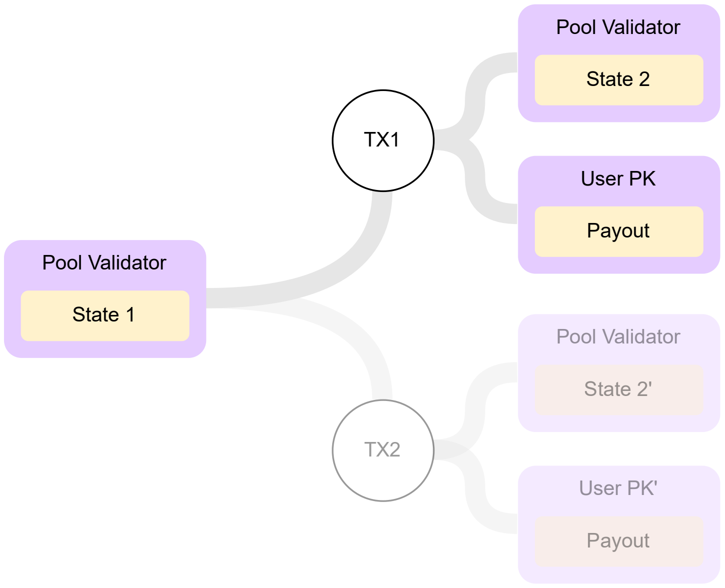

Figure 8: Transaction with AMM pool.

To tackle this issue, a two phase scheme is used on practice: user puts base assets required for an operation into an order and commits it into blockchain. Order's validator script ensures the user gets the desired amount of quote asset.

Another integral part of this scheme is an off-chain service that monitors blockchain and executes orders against actual pool states. Usually, because of the security and stability concerns (off-chain agents may re-order orders to maximize own profits) there is an open network of these agents. Their normal working cycle to race with other agents to submit transaction faster, because many of the transactions they produce interfere and thus are mutually exclusive. Nevertheless, mutually exclusive transactions are propagated in the network to some point anyway, utilizing its throughput in a non-efficient way.

In the case of a DEX and most of the use-cases involving some shared state extensively used by independent agents eUTxO incurs even more costs (compared to imperative approach of Programmable Accounts) in terms of utilization of network resources by requiring on-chain orders. It also incurs additional implementation complexity by requiring developers to implement off-chain services.

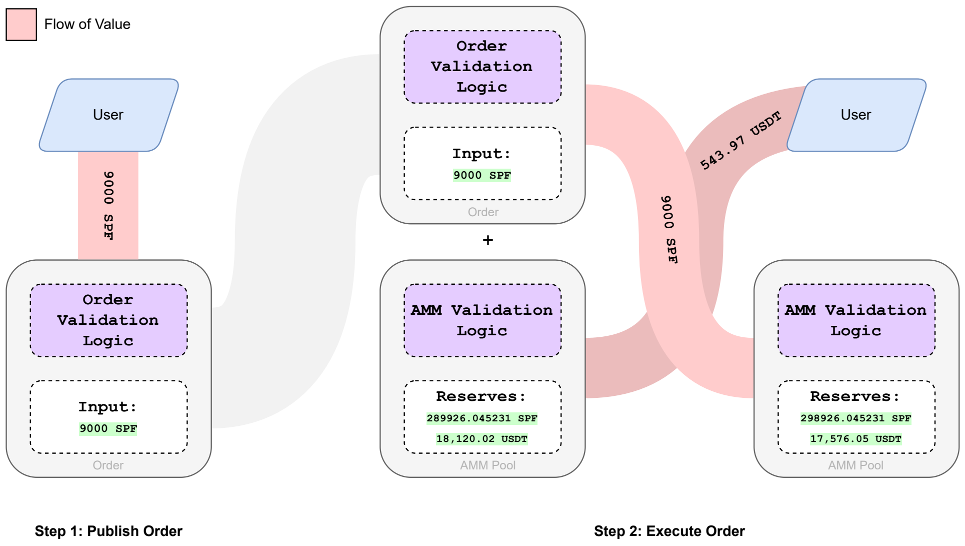

Figure 9: Two-pase scheme. Transaction publishing an order. Transaction executing the order against AMM pool.

3.3 Aggregation and sharding

AMM pool is the most simplistic example of aggregation. It aggregates liquidity from different liquidity providers in a continuous way: assets of the same type are summed.

Cases involving aggregation of discrete units require more sophisticated techniques. Consider a Liquidity Book approach: liquidity is distributed across discrete bins with fixed width. Liquidity can be exchanged at fixed price within each bin. Each bin represents a single price point and the difference between two consecutive bins is the bin step.

Pools of this type have to track state of tens on thousands of bins. That's why large pools can no longer be modelled as a single UTxO, otherwise they simply won't fit into transaction size limits.

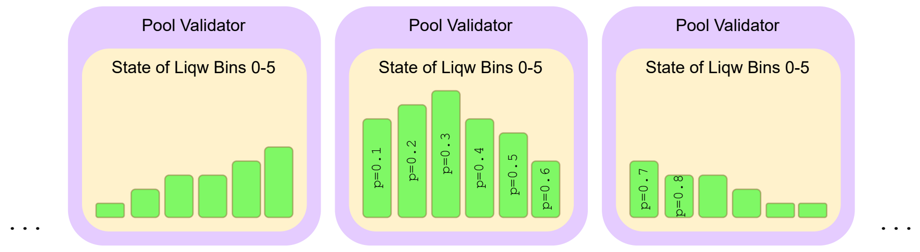

Figure 10: Liqudiity book is split into a possibly infinite number of shards.

In this case the technique of state sharding is useful. A Liquidity Book pool is modelled as an infinite number of discrete UTxOs, each tracking a particular range of bins.

On practice this approach has some limitations.

- The larger the amount of a single swap is, the more shards of the pool are required to execute transaction. In extreme cases all of them may not fit into one transaction and would require a series of transactions.
- In the case if a liquidity provider wants to provide liquidity in a wide range of bins, again, more shards of the pool are required to execute transaction. And again, a chain of transactions is required.

Although possible, and even practical depending on transaction size limits in the target network, this approach requires significant amount of data to be transmitted over the wire (in transactions and then in blocks) compared to typical transaction size limits (from 16kb to 1mb in different eUTxO blockchains).

3.4 Towards a better balance

We are now going to discuss how existing eUTxO approach can be improved to be more accessible to developers, utilize network resources efficiently, while still remaining secure and deterministic.

We argue, that one significant limitation of eUTxO that leads to inefficient utilization of network resources is the requirement to include full version of new state into transaction even if modifications are minimal.

To eliminate this, it would be beneficial to support operations of copying the previous state with modifications of only the required set of fields 11. Then, only new data would have to be included into transaction, transmitted over the wire and go through decentralized consensus.

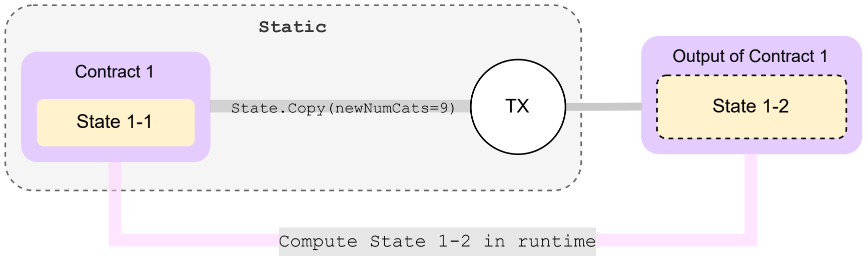

Figure 11: Resulting state is computed using the input state and the updated data.

Although useful to reduce data transmission, this modification alone would introduce another issue – the identifier (which is a hash of UTxO and transaction) of a fresh UTxO would be known only after confirmation of a block it got into. This would make it impossible to implement efficient transaction chaining and would reduce throughput of DEXes built on it significantly.

Another factor that blows up complexity and requires spamming network with similar mutually exclusive transactions is static dispatch of transaction inputs.

To tackle that in our construction we introduce identifier of UTxO composed of:

1. Stable part, which is derived from ID of the transaction that produced it initially and its index in the transaction outputs

2. And ephemeral part that represents version of the UTxO and is derived as a hash of the transaction that produced its latest version and its

index in the transaction outputs. Then we allow to optionally refer to a UTxO input using only stable part of identifier.

$$\boxed{\begin{aligned} \mathrm{TxId} =\ & \mathrm{H(Tx)} \\ \mathrm{StableId} =\ & \mathrm{H(TxId} \times \mathrm{I)} \\ \mathrm{Version} =\ & \mathrm{H(TxId} \times \mathrm{I)} \\ \mathrm{Ref} =\ & \mathrm{StableId} \times \mathrm{Version} \end{aligned}}$$

Table 2: Structure of UTxO reference.

The resulting hybrid construction allows developers to balance trade-off between on-chain and off-chain computations themselves depending on the use case. At the same time the construction preserves local reasoning about the transaction because it still operates on its explicit inputs.

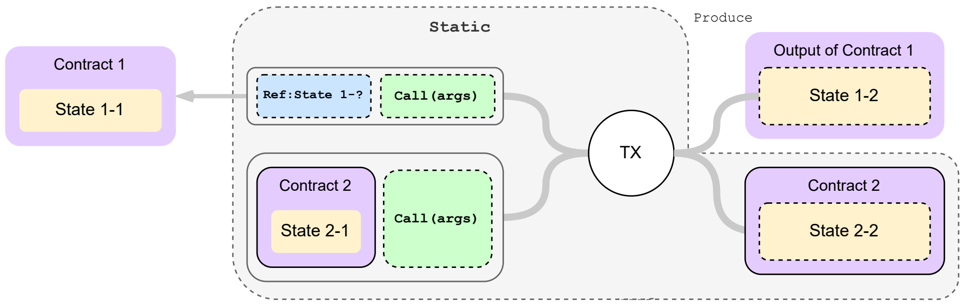

Figure 12: Hybrid-UTxO transaction.

References

[1] Vitalik Buterin. Ethereum white paper: A next generation smart contract & decentralized application platform. 2013.

[2] Alexander Chepurnoy, Vasily Kharin, and Dmitry Meshkov. Self-reproducing coins as universal turing machine, 2018.

[3] Ergo Developers. Ergo: A resilient platform for contractual money, 2019.

[4] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. May 2009.

[5] Jiahua Xu, Krzysztof Paruch, Simon Cousaert, and Yebo Feng. SoK: Decentralized exchanges (DEX) with automated market maker (AMM) protocols. *ACM Computing Surveys*, 55(11):1–50, feb 2023.