Designing EUTxO smart contracts as communicating state machines: the case of simulating accounts

Polina Vinogradova ✉
IOG

Manuel Chakravarty ✉
IOG

James Chapman ✉
IOG

Tudor Ferariu ✉
University of Edinburgh, UK

Michael Peyton Jones ✉
IOG

Jacco Krijnen ✉
Utrecht University, Netherlands

Orestis Melkonian ✉
IOG & University of Edinburgh, UK

Extended abstract

2012 ACM Subject Classification Security and privacy → Formal methods and theory of security

Keywords and phrases UTxO, state machines, message passing

Funding This work was supported by Input Output (iohk.io) through their funding of the Edinburgh Blockchain Technology Lab.

1 Introduction

The extended UTxO model with multi-asset support (EUTxOma) is a smart-contract enabled ledger model that natively supports the tracking and transfer of user-defined assets, which is specified in detail in [2]. There, a state machine formalism is presented that models more complex stateful contracts. Usually, such contracts are represented by linear sequences of transactions, which implies a monolithic state that can only be updated one step at a time by each transaction.

We develop a specification for an account simulation example to illustrate the monolithic-state implementation approach and its alternatives. The account simulation is a minimal example of practical interest to contrast these approaches. We study four implementations of this specification : two single-threaded and two multi-threaded ones. The single-thread approaches we give are a naive, monolithic-state implementation, and a Merkle trie optimization of it.

A key goal of this work, however, is dismantling the single-threaded state machine design into a multi-threaded one, wherein each thread is concerned with tracking only part of the contract state (in our example, a single account), with a focus on establishing a robust mechanism for the threads' cooperation. We compare two modes of thread cooperation for transferring assets between accounts : a direct transfer scheme, where both parties coordinate to submit a single transaction, and a message passing scheme that limits contract interaction to producing and consuming messages only.

2 Account simulation

Specification The API specification is made up of four operations on a single account (`open`, `close`, `withdraw`, `deposit`) and an operation that updates two accounts atomically (`transfer`). The `withdraw` moves assets from outside the account simulation into an account, and `deposit` moves assets from an account to outside the simulation. The `transfer` operation moves assets between two accounts.

Single-threaded implementations The **naive** implementation stores the information for all accounts in a single state-machine state as a key-value map that associates the public key of each account holder to their assets. It suffers from a lack of both parallelism and concurrency; we can only perform a single account operation per transaction (hence also per block, due to the nature of UTXO ledgers). This approach also suffers from an inefficient usage of memory space in the transaction, as the entire key-value map must be included in any transaction operating on the account state.

For the second implementation, we optimize the naive implementation by using a **Merkle trie** structure to reduce the use of transaction memory space. The key-value pairs are stored at the leaves, and the binary-encoded key specifies the path to the leaf with that key. The memory space occupied by state in a transaction transitioning the state machine is reduced from linear to logarithmic in the number of accounts. However, the parallelism and concurrency issues are still not addressed by this optimization.

Multi-threaded implementations We have designed two multi-threaded implementations of the account specification, which model each account with a distinct state machine, each identified by a unique token. There is no master contract which controls or tracks the changes to all accounts. This improves upon the single-threaded versions in terms of concurrency, parallelism, and memory use. The `open`, `close`, `deposit`, and `withdraw` transitions for a single account are still limited to one operation per transaction/block, but can now be simultaneously performed on multiple accounts. Transaction memory use per account transition is reduced as compared to both the naive and the Merkle trie implementations, and is proportional to the space occupied by a single account state.

Direct transfer between two accounts requires both participants to validate the same transaction, such that one's assets are decreased by an amount and the other's are increased by the same amount. The **message-passing** scheme, on the other hand, implements the `transfer` constraint by requiring the transaction to produce a *message*, which is a UTXO locked by a message contract. A message contains the amount being transferred, and specifies both the sender and the recipient state machine. A state machine must make a special transition in order to either produce or consume a message.

As a result of the EUTxO system's built-in capacity for concurrency, each message has a unique identifier (the output reference of its UTxO), as well as specifying a sender and a receiver, therefore we can ensure that messages produced or consumed by transitions are (1) unique, and (2) can only be produced and consumed by transitioning the state machines specified in them.

Message-passing is useful in addressing existing practical problems within the EUTxO ledger. Message-passing architecture requires two transactions to implement `transfer` instead of one. However, direct transfer requires some *off-chain coordination* to get the recipient to sign the transaction, which we conjecture to be an undesirable trade-off. Message-passing might be optimized to produce or consume multiple messages in a single transition.

Now, consider a contract which requires the transaction executing it to also perform another action, such as transferring funds from another contract. A transaction may execute two such contracts, but only perform the required transfer once, nevertheless satisfying the constraints of both contracts being executed. This situation, which we call the *double-satisfaction problem*, is difficult to recognize during static analysis, but message-passing offers a simple solution. It enables a contract to impose a constraint limiting the transaction to containing no other contracts, without otherwise affecting its validation. Isolating contracts in separate transactions in this way solves the problem.

Message-passing allows concurrent and parallel state updates, as well as a modular way to formally verify a contract by only reasoning about individual threads along with the messages they exchange. Communicating automata have already been studied in the context of account-based blockchain ledgers, as exemplified by the Scilla programming language [3], and the message-passing approach to concurrency has already proven to be valuable in systems such as Erlang [1] and Akka [4]. These precedents suggest that message-passing is a natural fit for a wide range of EUTXO-based smart contracts.

3 Discussion and future work

We intend to deploy, test, and benchmark the four implementations we have discussed. These results will allow us to compare the transaction memory use in each version, and determine the crossover point at which the Merkle trie implementation uses less time and space than the naive implementation.

We plan to investigate what properties can be proven about our four implementations using static analysis and proof assistants. We have used Liquid Haskell to verify properties of the naive implementation. Checking the property that each account must contain a non-negative quantity of assets led to finding (and subsequently fixing) a bug in our specification and the implementation. We will continue to investigate other properties, such as the claim that the total amount of assets in the account simulation system can only be changed by a deposit or withdrawal operation. We anticipate that message passing may enable more modular reasoning about our systems.

We conjecture that the minimal accounts example we have presented is representative of the multi-threaded architecture we propose, and that a broader class of contracts can be designed according to this scheme. Our next step is to specify and implement a limit-order decentralized exchange (DEX) as an extension to our current model, wherein accounts additionally include assets that are offered up for exchange.

References

1 Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, 2003.

2 Manuel M. T. Chakravarty et al. Native custom tokens in the extended UTXO model. In *ISoLA 2020, Rhodes, Greece, Proceedings, Part III*, volume 12478 of *Lecture Notes in Computer Science*, pages 89–111. Springer, 2020. URL: `https://iohk.io/en/research/library/papers/native-custom-tokens-in-the-extended-utxo-model/`, `doi:10.1007/978-3-030-61467-6\_7`.

3 Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. Safer smart contract programming with Scilla. *Proc. ACM Program. Lang.*, 3(OOPSLA):185:1–185:30, 2019. `doi:10.1145/3360611`.

4 Derek Wyatt. *Akka concurrency*. Artima Incorporation, 2013.