Formal investigation of the Extended UTxO model

Laying the foundations for the formal verification of smart contracts

ORESTIS MELKONIAN, Utrecht University, The Netherlands

This report serves as the proposal of my MSc thesis, supervised by Wouter Swierstra from Utrecht University and Manuel Chakravarty from IOHK.

1 INTRODUCTION

Although blockchain technology has opened a whole array of interesting new applications (e.g. secure multi-party computation[Andrychowicz et al. 2014], fair protocol design fair[Bentov and Kumaresan 2014], zero-knowledge proof systems[Goldreich et al. 1991]), reasoning about the behaviour of such systems is an exceptionally hard task. This is partly due to their concurrent nature, but also the fiscal nature of the majority of the applications, which require a much higher degree of rigorousness compared to conventional IT applications.

The advent of smart contracts (programs that run on the blockchain itself) gave rise to another source of vulnerabilities. One primary example of such a vulnerability caused by the use of smart contracts is the DAO attack[1], where a security flaw on the model of Ethereum's scripting language led to the exploitation of a venture capital fund worth 150 million dollars at the time. The solution was to create a hard fork of the Ethereum blockchain, clearly going against the decentralized spirit of cryptocurrencies. Since these (possibly Turing-complete) programs often deal with transactions of significant funds, it is of utmost importance that one can reason and ideally provide formal proofs about their behaviour in a concurrent/distributed setting.

Research Question. The aim of this thesis is to provide a mechanized formal model of an abstract distributed ledger equipped with smart contracts, in which one can begin to formally investigate the expressiveness of the extended UTxo model. Moreover, we hope to lay down firm grounds, onto which one can further conduct a formal comparison with account-based models used in Ethereum. Put concisely, the research question posed is:

> *How much expressiveness do we gain by extending the UTxO model?*
> *Is it as expressive as the account-based model used in Ethereum?*

Overview. Section 2 reviews some basic definitions related to blockchain technology and introduces important literature, which will be the main subject of study throughout the development of our reasoning framework. Section 3 describes the technology we will use to formally reason about the problem at hand and some key design decisions we set upfront. Section 4 presents the progress made thus far in terms of (mechanized) formal verification, as well as problems we have encountered and also expect along the way. Section 5 discusses next steps for the remainder of the thesis, as well as a rough estimate on when these milestones will be completed.

2 BACKGROUND

2.1 Distributed Ledger Technology: Blockchain

Cryptocurrencies rely on distributed ledgers, where there is no central authority managing the accounts and keeping track of the history of transactions.

[1] https://en.wikipedia.org/wiki/The_DAO_(organization)

Author's address: Orestis Melkonian, Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, melkon.or@gmail.com.

One particular instance of distributed ledgers are blockchain systems, where (unrelated) transaction are bundled together in blocks, which are linearly connected with hashes and distributed to all participants/peers. The blockchain system, along with a consensus protocol deciding on which competing fork of the chain is to be included, maintains an immutable distributed ledger (i.e. the history of transactions).

Validity of the transactions is tightly coupled with a consensus protocol, which makes sure peers in the network only validate well-behaved/truthful transactions and are, moreover, properly incentivized to do so.

The absence of a single central authority that has control over all assets of the participants allows for shared control of the evolution of data (in this case transactions) and generally leads to more robust and fair management of assets.

While cryptocurrencies are the major application of blockchain systems, one could easily use them for any kind of valuable assets, or even as general distributed databases.

2.2 Smart Contracts

Most blockchain systems come equipped with a scripting language, where one can write *smart contracts* that dictate how a transaction operates. A smart contract could, for instance, pose restrictions on who can redeem the output funds of a transaction.

One could view smart contracts as a replacement of legal frameworks, providing the means to conduct contractual relationships completely algorithmically.

While previous work on writing financial contracts [Jones et al. 2000] suggests it is fairly straightforward to write such programs embedded in a general-purpose language (in this case Haskell) and to reason about them with *equational reasoning*, it is restricted in the centralized setting and, therefore, does not suffice for our needs.

Things become much more complicated when we move to the distributed/blockchain setting, as can be evidenced by the current attempts being made to overcome this [Bhargavan et al. 2016; Sergey et al. 2018; Setzer 2018]. Hence, there is a growing need for methods and tools that will enable tractable and precise reasoning about such systems.

Numerous scripting languages have appeared recently [Seijas et al. 2016], spanning a wide spectrum of expressiveness and complexity. While language design can impose restrictions on what can a language express, most of these restriction are inherited from the accounting model that the underlying system adhere to.

In the next section, we will discuss the two main forms of accounting models:

(1) **UTxO-based**: stateless models based on *unspent transaction outputs*
(2) **Account-based**: stateful models that explicitly model interaction between *user accounts*

2.3 UTxO-based: Bitcoin

The primary example of a UTxO-based blockchain is Bitcoin [Nakamoto 2008]. Its blockchain is a linear sequence of *blocks* of transactions, starting from the initial *genesis* block. Essentially, the blockchain acts as public log of all transactions that have taken place, where each transaction refers to outputs of previous transactions, except for the initial *coinbase* transaction of each block. Coinbase transactions have no inputs, create new currency and reward the miner of that block with a fixed amount. Bitcoin also provides a cryptographic protocol to make sure no adversary can tamper with the transactional history, e.g. by making the creation of new blocks computationally hard and invalidating the ”truthful” chain statistically impossible.

A crucial aspect of Bitcoin’s design is that there are no explicit addresses included in the transactions. Rather, transaction outputs are actually program scripts, which allow someone to claim the

funds by giving the proper inputs. Thus, although there are no explicit user accounts in transactions, the effective available funds of a user are all the *unspent transaction outputs* (UTxO) that he can claim (e.g. by providing a digital signature).

2.3.1 Script. In order to state such scripts in the outputs of a transaction, Bitcoin provides a low-level, Forth-like, stack-based scripting language, called Script. Script is intentionally not Turing-complete (e.g. it does not provide looping structures), in order to have more predictable behaviour. Moreover, only a very restricted set of "template" programs are considered standard, i.e. allowed to be relayed from node to node.

P2PKH. The most frequent example of a 'standard' program in Script is the *pay-to-pubkey-hash* (P2PKH) type of scripts. Given a hash of public key `<pub#>`, a P2PKH output carries the following script:

```
OP_DUP OP_HASH <pub#> OP_EQ OP_CHECKSIG
```

where `OP_DUP` duplicates the top element of the stack, `OP_HASH` replaces the top element with its hash, `OP_EQ` checks that the top two elements are equal, `OP_CHECKSIG` verifies that the top two elements are a valid pair of a digital signature of the transaction data and a public key hash.

The full script will be run when the output is claimed (i.e. used as input in a future transaction) and consists of the P2PKH script, preceded by the digital signature of the transaction by its owner and a hash of his public key. Given a digital signature `<sig>` and a public key hash `<pub>`, a transaction is valid when the execution of the script below evaluates to `True`.

```
<sig> <pub> OP_DUP OP_HASH <pub#> OP_EQ OP_CHECKSIG
```

To clarify, assume a scenario where Alice want to pay Bob ₿ 10. Bob provides Alice with the cryptographic hash of his public key (`<pub#>`) and Alice can submit a transaction of ₿ 10 with the following output script:

```
OP_DUP OP_HASH <pub#> OP_EQ OP_CHECKSIG
```

After that, Bob can submit another transaction that uses this output by providing the digital signature of the transaction `<sig>` (signed with his private key) and his public key `<pub>`. It is easy to see that the resulting script evaluates to `True`.

P2SH. A more complicated script type is *pay-to-script-hash* (P2SH), where output scripts simply authenticate against a hash of a *redeemer* script `<red#>`:

```
OP_HASH <red#> OP_EQ
```

A redeemer script `<red>` resides in an input which uses the corresponding output. The following two conditions must hold for the transaction to go through:

(1) ⟦`<red>`⟧ = `True`
(2) ⟦`<red> OP_HASH <red#> OP_EQ`⟧ = `True`

Therefore, in this case the script residing in the output are simpler, but inputs can also contain arbitrary redeemer scripts (as long as they are of a standard "template").

In this thesis, we will model scripts in a much more general, mathematical sense, so we will eschew from any further investigation of properties particular to Script.

2.3.2 The BitML Calculus. Although Bitcoin is the most widely used blockchain to date, many aspects of it are poorly documented. In general, there is a scarcity of formal models, most of which are either introductory or exploratory.

One of the most involved and mature previous work on formalizing the operation of Bitcoin is the Bitcoin Modelling Language (BitML) [Bartoletti and Zunino 2018]. First, an idealistic *process calculus* that models Bitcoin contracts is introduced, along with a detailed small-step reduction semantics that models how contracts interact and its non-determinism accounts for the various outcomes.

The semantics consist of transitions between *configurations*, abstracting away all the cryptographic machinery and implementation details of Bitcoin. Consequently, such operational semantics allow one to reason about the concurrent behaviour of the contracts in a *symbolic* setting.

The authors then provide a compiler from BitML contracts to 'standard' Bitcoin transactions, proven correct via a correspondence between the symbolic model and the computational model operating on the Bitcoin blockchain. We will return for a more formal treatment of BitML in Section 4.2.

2.3.3 Extended UTxO. In this work, we will consider the version of the UTxO model used by IOHK's Cardano[2] blockchain. In contrast to Bitcoin's *proof-of-work* consensus protocol [Nakamoto 2008], Cardano's *Ouroboros* protocol [Kiayias et al. 2017] is *proof-of-stake*. This, however, does not concern our study of the abstract accounting model, thus we refrain from formally modelling and comparing different consensus techniques.

The actual extension we care about is the inclusion of *data scripts* in transaction outputs, which essentially provide the validation script in the corresponding input with additional information of an arbitrary type.

This extension of the UTxO model has already been implemented[3], but only informally documented[4]. The reason to extend the UTxO model with data scripts is to bring more expressive power to UTxO-based blockchains, hoping that it is on par with Ethereum's account-based scripting model (see Section 2.4).

However, there is no formal argument to support this claim, and it is the goal of this thesis to provide the first formal investigation of the expressiveness introduced by this extension.

2.4 Account-based: Ethereum

On the other side of the spectrum, lies the second biggest cryptocurrency today, Ethereum [**?**]. In contrast to UTxO-based systems, Ethereum has a built-in notion of user addresses and operates on a stateful accounting model. It goes even further to distinguish *human accounts* (controlled by a public-private key pair) from *contract accounts* (controlled by some EVM code).

This added expressiveness is also reflected in the quasi-Turing-complete low-level stack-based bytecode language in which contract code is written, namely the *Ethereum Virtual Machine* (EVM). EVM is mostly designed as a target, to which other high-level user-friendly languages will compile to.

Solidity. The most widely adopted language that targets the EVM is *Solidity*, whose high-level object-oriented design makes writing common contract use-cases (e.g. crowdfunding campaigns, auctions) rather straightforward.

One of Solidity's most distinguishing features is the concept of a contract's *gas*; a limit to the amount of computational steps a contract can perform. At the time of the creation of a transaction, its owner specifies a certain amount of gas the contract can consume and pays a transaction fee proportional to it. In case of complete depletion, all global state changes are reverted. This is a necessary ingredient for smart contract languages that provide arbitrary looping behaviour, since non-termination of the validation phase is certainly undesirable.

[2] www.cardano.org
[3] https://github.com/input-output-hk/plutus/tree/master/wallet-api/src/Ledger
[4] https://github.com/input-output-hk/plutus/blob/master/docs/extended-utxo/README.md

Formal investigation of the Extended UTxO model

If time permits, we will initially provide a formal justification of Solidity and proceed to formally compare the extended UTxO model against it. Since Solidity is a fully-fledged programming language with lots of features (e.g. static typing, inheritance, libraries, user-defined types), it makes sense to restrict our formal study to a compact subset of Solidity that is easy to reason about. This is the approach also taken in Featherweight Java [Igarashi et al. 2001]; a subset of Java that omits complex features such as reflection, in favour of easier behavioural reasoning and a more formal investigation of its semantics. In the same vein, we will try to introduce a lightweight version of Solidity, which we will refer to as *Featherweight Solidity*.

3 METHODOLOGY

3.1 Scope

... no cryptographic/implementation details ... ... focus on the big picture ...

3.2 Proof Mechanization

Through mechanization ... ... vs informal mathematics ...

It is exactly this side effect, that will allow us to discover edge cases and increase the confidence of the model under investigation.

As our proof development vehicle, we choose Agda...

3.3 Agda

... stay on a highly abstract level ... postulate cryptographic operations etc ...

Limitation. ... proof automation ala Coq ...

3.4 The IOHK approach

... rigorous methodology ... ... industry co-existing with academia ... ... formal verification (Agda/Coq) -> prototype/reference implementation (Haskell) -> production codebase (Haskell) ...

3.5 Functional first

... UTxO vs Account ... ... functional vs imperative ... ... dataflow vs ?? ...

4 PRELIMINARY RESULTS

4.1 Formal Model I: Extended UTxO

4.1.1 Inherently-typed validity of transactions.

4.1.2 Scripts via Denotational Semantics.

4.1.3 Address space as module parameter.

4.1.4 Weakening Lemma.

4.1.5 Example.

4.2 Formal Model II: BitML Calculus

4.2.1 Contracts in BitML.

4.2.2 Small-step Semantics. ... mention paper bug in [C-Control] ...

4.2.3 Configurations modulo permutation.

4.2.4 Example.

4.3 Expected Problems

... up to permutation -> quotient types -> homotopy type theory ...

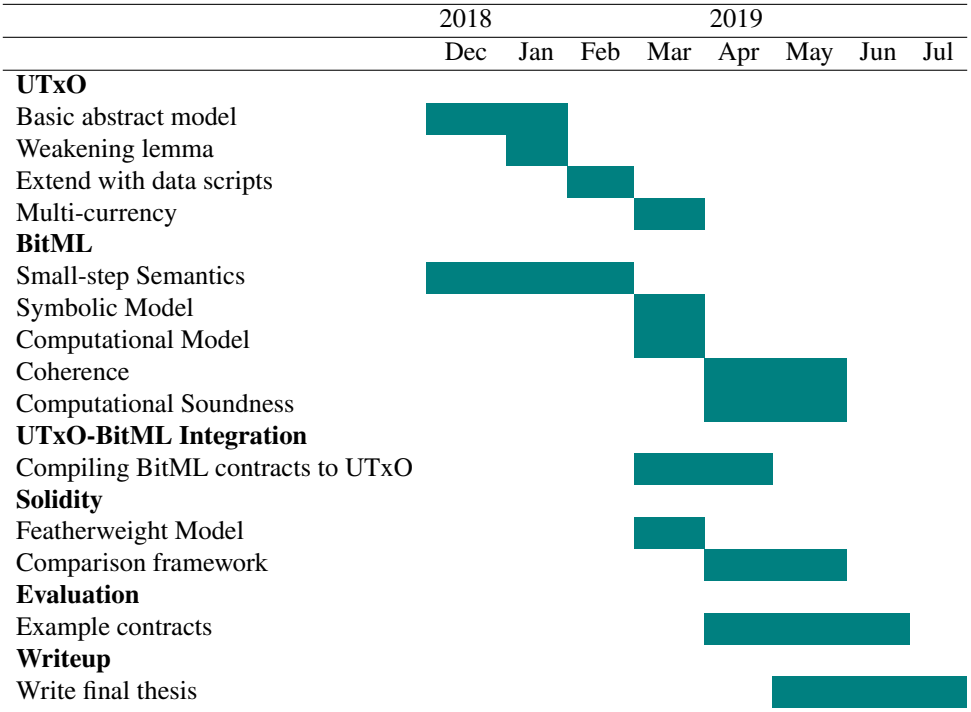

Fig. 1. My workplan.

5 PLANNING

5.1 Extended UTxO

... multi-currency ...

5.2 BitML Calculus

... symbolic runs ... computational runs ... coherence

5.3 UTxO-BitML Integration

5.4 Featherweight Solidity

5.5 Formal Comparison

... lots of examples ...

5.6 Proof Automation

5.7 Timetable

... mention things that are really out-of-scope ...

REFERENCES

2010. Script - Bitcoin Wiki. Retrieved 2/2019 from https://en.bitcoin.it/wiki/Script

2018. Formal verification of a Cardano wallet. Retrieved 2/2019 from https://cardanodocs.com/files/formal-specification-of-the-cardano-wallet.pdf

2019. The Extended UTxO Model. Retrieved 2/2019 from https://github.com/input-output-hk/plutus/blob/master/docs/extended-utxo/README.md

Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. 2014. Secure multiparty computations on bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 443–458.

Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. 1997. *The Coq proof assistant reference manual: Version 6.1*. Ph.D. Dissertation. Inria.

Massimo Bartoletti and Roberto Zunino. 2018. *BitML: a calculus for Bitcoin smart contracts*. Technical Report. Cryptology ePrint Archive, Report 2018/122.

Iddo Bentov and Ranjit Kumaresan. 2014. How to use bitcoin to design fair protocols. In *International Cryptology Conference*. Springer, 421–439.

Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cedric Fournet, A Gollamudi, G Gonthier, N Kobeissi, A Rastogi, T Sibut-Pinote, N Swamy, and S Zanella-Béguelin. 2016. Short paper: Formal verification of smart contracts. In *Proceedings of the 11th ACM Workshop on Programming Languages and Analysis for Security (PLAS), in conjunction with ACM CCS*. 91–96.

Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* (2014).

Oded Goldreich, Silvio Micali, and Avi Wigderson. 1991. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of the ACM (JACM)* 38, 3 (1991), 690–728.

Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23, 3 (2001), 396–450.

S Peyton Jones, Jean-Marc Eber, and Julian Seward. 2000. Composing contracts: an adventure in financial engineering. *ACM SIG-PLAN Notices* 35, 9 (2000), 280–292.

Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. 2017. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*. Springer, 357–388.

Per Martin-Löf and Giovanni Sambin. 1984. *Intuitionistic type theory*. Vol. 9. Bibliopolis Naples.

Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

Ulf Norell. 2008. Dependently typed programming in Agda. In *International School on Advanced Functional Programming*. Springer, 230–266.

Pablo Lamela Seijas, Simon J Thompson, and Darryl McAdams. 2016. Scripting smart contracts for distributed ledger technology. *IACR Cryptology ePrint Archive* 2016 (2016), 1156.

Ilya Sergey, Amrit Kumar, and Aquinas Hobor. 2018. Scilla: a smart contract intermediate-level language. *arXiv preprint arXiv:1801.00687* (2018).

Anton Setzer. 2018. Modelling Bitcoin in Agda. *arXiv preprint arXiv:1804.06398* (2018).

Joachim Zahnentferner. 2018. An Abstract Model of UTxO-based Cryptocurrencies with Scripts. *IACR Cryptology ePrint Archive* 2018 (2018), 469.

Joachim Zahnentferner and Input Output HK. 2018. *Chimeric ledgers: Translating and unifying utxo-based and account-based cryptocurrencies*. Technical Report. Cryptology ePrint Archive, Report 2018/262, 2018. https://epri nt. iacr. org ….