[Short Paper] Checking Laws of the Blockchain With Property-Based Testing

Alexander Chepurnoy and Mayank

No Institute Given

Abstract. Inspired by enormous success of Bitcoin, which reference implementation was released after a whitepaper [?] in 2009, many alternative implementations of a Bitcoin protocol, and also alternative blockchain protocols were released. However, implementations contain errors, and cost of an error in case of a cryptocurrency could be extremely high.
In order to tackle the problem we propose a suite of abstract properties tests checking laws of potentially every blockchain system. When developing a new system, a developer needs to instantiate generators for random objects which the tests are using, getting ready test suite checking the laws over many random cases in result. We provide examples of some laws.

1 Introduction

A blockchain-based cryptocurrency is defined by a set of protocols, such as consensus protocol, , agreement on tokens emission rules, . A node, or a *client* is software implementation of the protocols. Even more, usually not all the details of all the protocols are specified, so there is some reference client implementation which defines behavior for other implementations. the notable exception here is Ethereum, which Yellow Paper [?] is trying to define all the details of a client implementation. In Bitcoin, the reference implementation Bitcoin Core is considered to be standard, so an alternative implementation must reproduce its behavior, even bugs. (Alex notes : cite)

An error in a client implementation would be utterly costly and hard to fix. (Alex notes : examples) On other hand, there is an increasing demand in developing more blockchain protocols and clients.

Plenty of modular open-source frameworks were proposed for speeding up development of new blockchains: Sawtooth [?] and Fabric [?] by Hyperledger, Exonum [?] by Bitfury Group, Scorex [?] by IOHK etc. We have chosen Scorex Alex notes : why?.

1.1 Our Contribution

In this paper we report on design and implementation of a suite of abstract property tests which are implemented for Scorex framework and checking laws

of a blockchain client. A developer of a concrete blockchain system just needs to implement generators of random test inputs (such as blocks and transactions), and then the testing system will extensively check properties against multiple input objects. We have implemented XX Alex notes : concrete number property tests. We integrated the tests into a prototype implementation of the TwinsCoin cryptocurrency, which has two types of blocks. Alex notes : enhance

1.2 Property-based testing

In this section, we will begin by giving a formal definition of a property followed by a discussion on property-based testing in contrast to the conventional testing methodologies.
Within the scope of a data domain $\mathbb{D}$, a property can be seen as a collective abstract behaviour which has to be followed by every valid member of that data domain. In other words, a property can be understood as a predicate P over a variable X ($P : X \rightarrow \{true, false\}$) such that:

$$\forall X \in \mathbb{D}, P(X) = true$$

Let us consider an example of a property P over the domain of all possible strings $\mathbb{S}$.

$$\forall X \in \{s_1 :: s_2 | \#s_1 :: s_2 > \#s_1 \wedge s_1, s_2 \in \mathbb{S}\}, P(X) = true$$

where :: denotes a regular string append operation and $\#s$ denotes the length of string s. But if we look closely, we observe that if $s_1, s_2 \in \{\phi\}$ then the property P does not hold true anymore contrary to the fact that both component strings are valid members of S. Hence, P is not a valid property over the domain S.
In contrast to conventional testing methods where it might suffice to test the behaviour of some boundary points at the discrete neighbourhoods, property-based testing emphasises on defining universal (within the domain) properties and then testing their validity against randomly sampled data points. There are some popular libraries available for property testing including QuickCheck for Haskell, JUnit-QuickCheck for Java, theft for C, and ScalaTest and ScalaCheck (majorly for generator-driven property testing) for Scala-based programs.

Alex notes : improve the bridge Walking inside the walls of bitcoin and other cryptocurrencies, we, in this work, highlight some of these properties which should hold true for any implementation of a valid cryptocurrency.

1.3 Scorex Framework

The idea of a modular design for a cryptocurrency was first proposed by Goodman in Tezos whitepaper [?]. The whitepaper [1] proposes to split a cryptocurrency design into the three protocols: network, transaction and consensus. In many cases, however, these layers are tightly coupled and it is hard to describe

[1] Section 2 of the whitepaper

them separately. For example, in a proof-of-stake cryptocurrency a balance sheet structure, which is heavily influenced by a transaction structure, is used in a consensus protocol. To split a cryptocurrency design in more clear way, Scorex 2.0 has finer granularity. In particular, in order to support hybrid blockchains as well as more complicated linking structures than a chain (such as SPECTRE[?]), Scorex 2.0 does not have a notion of the blockchain as a core abstraction. Instead, it provides an abstract interface to a *history* which contains *persistent modifiers*. The history is a part of a *node view*, which is a quadruple of ⟨*history*, *minimal state*, *vault*, *memory pool*⟩. The minimal state is a data structure and a corresponding interface providing an ability to check a validity of an arbitrary persistent modifier for the current moment of time with the same result for all the nodes in the network having the same history. The minimal state is to be obtained deterministically from an initial pre-historical state and the history. The vault holds node-specific information, for example, a node user's wallet. The memory pool holds unconfirmed transactions being propagated across the networks by nodes before got into blocks. Alex notes : write what are we doing with it

1.4 Related Work

Verification and testing of software systems is an integral part of a software development lifecycle. Immediately after the implementation of the software, and before its deployment, it has to be verified and tested extensively enough to ensure that all the functional requirements have been properly met. A lot of methods have been developed over the course of time for verification and testing of software. Formal verification, for example, is one of the most popular methods of program verification. It is used to validate the correctness of a software module by modeling its behaviour based on a set of formal methods, which are mathematical models specifying the intended functional behaviour. Though there are a lot of formal methods used for formal verification, and are selected based on the functional outlook of the software program, the most common methods are the ones based on finite state machines, Petri Nets, process algebra and timed automata. Prior to formal verification, a mathematical model employing a formal method is decided upon which is then followed by a specification phase where the behaviour of the system is modeled. Finally the actual program is verified against this behaviour specification which is called the verification phase. In the late 90's, some optimisations were proposed to the otherwise conventional and inefficient exhaustive search methods [http://spinroot.com/spin/Doc/forte94a.pdf].

Since a software program is developed at module or class level and is integrated with other modules or classes along the development cycle, testing is done at both, unit level as well as integration level, before the software is deployed. Unit tests target individual modules, methods or classes and have a small coverage compared to integration tests which aim towards checking the behaviour of modules when combined together. The two main approaches to unit testing are black box testing and white box testing. The former one focuses on designing test instances without looking inside the code or design, and in other words black

box testing only focuses on the functionality, while the latter approach is more inclined towards testing code coverage i.e. writing test instances which employ the different paths inside the code. Though initially white box testing was considered as a method suitable for unit testing alone, recently it has emerged as a popular method for integration testing as well. Integration testing is usually done by one or a combination of big-bang approach, top-down approach, bottom-up approach and the sandwich approach. The final full scale testing that a software undergoes is called the system testing which includes a lot of tests including security test, compatibility test, exception handling, scalability tests and some performance tests.

Opposed to the conventional unit testing methods which are closed i.e. they do not take any input parameters, Parameterized unit tests (add the main paper here) are generalized tests which have an encapsulated collection of test methods whose invocation and behaviour is controlled by a set of input parameters giving more flexibility and automation to unit testing as a whole.

In this manuscript, we focus on a different method of program testing and verification, called property testing. A property testing algorithm is concerned with making approximate decisions as to whether an entity, which is modelled mathematically as a function, satisfies a property globally or is far from having the property, using only a small number of queries (compared to the function's domain).

1.5 Organization of the Paper

2 Scorex Architecture

Scorex is a framework for implementing blockchain clients. A client is a node in a peer-to-peer network. The node has a local view of the network state. The goal of the whole peer-to-peer system [2] is to synchronize on a critical part of local views. Scorex is breaking node's local view into the following four parts:

- *history* contains log of persistent modifiers. For example, in a Bitcoin-like blockchain the log is about a sequence of blocks. For an arbitrary persistent modifier, a history implementation can define whether the modifier is valid against it or not. If the modifier is valid w.r.t history, we call it the *syntactically* valid modifier. Alex notes : explain/example
- *minimal state* is a structure enough to check semantics of an arbitrary persistent modifier with a constraint that the checking has to be deterministic in nature. If a modifier is valid w.r.t minimal state, we call it the *semantically* valid modifier.
- *vault* contains user-specific information. A wallet or additional indexes to get richer information from the blockchain are perfect examples of vault implementation.

[2] concretely, honest nodes of it. We skip a notion of adversarial behavior further.

– *memory pool* contains temporary objects to be packed into persistent modifiers. An unconfirmed transaction is a clear example of an object living in the pool.

The history and the minimal state are the parts of local views to be synchronized across the network. For this, nodes are running some consensus protocol to form a proper history, and the history should be resulted in a valid minimal state which is got by application of persistent modifiers contained in history to a prehistorical state.

The whole node view quadruple is to be updated atomically by applying whether a persistent node view modifier or an unconfirmed transaction. Scorex provides guarantees of atomicity and consistency for the application while a developer of a concrete system needs to provide implementations for the abstract parts of the quadruple as well as a family of persistent modifiers.

A central component which is holding the quadruple <*history, minimal state, vault, memory pool*> and processing its updates atomically is called *node view holder*. The holder is implemented as an actor Alex notes : link, so it is processing all the messages it is getting in sequence even if being requested from multiple threads. If the holder is getting a transaction, it updates the vault and the memory pool with it. Otherwise, if the holder is getting a persistent modifier, it is first updating the history by appending the modifier to it. If appending is successful (thus the modifier is syntactically valid), the modifier is to be applied to the minimal state. Alex notes : finish, write about forks

As an example, we consider a cryptocurrency Twinscoin, which consensus protocol is a combination of Proof-of-Work and Proof-of-Stake. Scorex has a full-fledged Twinscoin implementation as an example of its usage. In Twinscoin, there are two kinds of persistent modifiers, a Proof-of-Work block, and a Proof-of-Stake block.

3 Blockchain properties

In this section we provide our approach to generalized testing of abstract blockchain-like system design. For extensive testing we are testing history, minimal state and memory pool separately, and also node view holder which updates the quadruple <*history, minimal state, vault, memory pool*> in an atomic way.

3.1 Generators

In our test suites we are using generators for objects we are using to check laws of a blockchain. A developer of a concrete system needs to implement these generators. We provide interfaces for generators of following types of objects:

– Syntactically valid, respectively, invalid modifier, which is valid, respectively invalid, based on a node's local view of history.
– Semantically valid, respectively, invalid modifier, which is valid, respectively invalid, based on a node's local view of minimal state.

– Totally, so both semantically and syntactically, valid modifier. Respectively, a sequence of totally valid modifiers
– Transaction

For example, in our TwinsCoin implementation, we provide a concrete implementation to these interfaces. To generate syntactically valid modifiers, we generate a PoW block if a previous pair of $<PoW\ block,\ PoS\ block>$ is complete, otherwise we generate a new PoS block. It can be noted that, in TwinsCoin, transactions are only added to PoS blocks. A minimal state in this implementation deterministically determines the validity of an arbitrary transaction. To generate semantically valid modifiers we generate a PoS block based on transaction outputs, or boxes. A totally valid modifier generator has a matching PoS block among a syntactically valid and a semantically valid modifier. A generated transaction takes as input unused boxes and returns as output a set of new boxes.

3.2 Forking

Alex notes : forking

4 Examples of properties tests

We are providing some examples of property tests.

5 Conclusion

A Tests Implemented

Alex notes : check the lists:

Implemented test scenarios - Valid persistent modifier should be successfully applied to history and available by id after that. - Valid box should be successfully applied to state, it's available by id after that. - State should be able to generate changes from valid block and apply them. - Wallet should contain secrets for all it's public propositions. - State changes application and rollback should lead to the same state and the component changes should also be rolled back. - Transactions successfully added to memory pool should be available by id. - Transactions once added to a block should be removed from the local copy of mempool. - Mempool should be able to store a lot of transactions and filtering of valid and invalid transactions should be fast. - Minimal state should be able to add and remove boxes based on received transaction's validity. - Modifier (to change state) application should lead to new minimal state whose elements' intersection with previous ones is not complete (at least some new boxes are introduced and some previous ones removed). - Application of the same modifier twice should be unsuccessful. - Application of invalid modifier (inconsistent with the previous ones) should be unsuccessful. - Application of a valid modifier after rollback should be successful. - Invalid modifiers should not be able to be added to history. - Once an invalid modifier is appended to history, then history should not contain it and neither should it be available in history by it's id. - History should contain valid modifier and report if a modifier if semantically valid after successfully appending it to history. - BlockchainSanity test that combines all this test.

Coming test scenarios: - Block application and rollback leads to the same history (rollback is not defined for history yet) - NodeView apply block to both state and history or don't apply to any of them - It is not possible to apply transaction to a state twice - Tests for invalid transactions/blocks/etc

Alex notes : generators list