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

Alexander Chepurnoy and Mayank

No Institute Given

Abstract. Inspired by the enormous success of the reference implementation of Bitcoin [?] in 2009, many alternative implementations of the Bitcoin protocol and of alternative blockchain protocols were released. However, implementations contain errors, and the cost of an error in case of a cryptocurrency could be extremely high.
We propose to tackle the problem with a suite of abstract property tests that check whether a blockchain system satisfies some laws which all of blockchain and blockchain-like systems do satisfy. To test a new blockchain system, the developers need to instantiate generators of random objects to be used by the tests. The test suite then checks the satisfaction of the laws over many random cases. We provide examples of some laws.

1 Introduction

A blockchain-based cryptocurrency is defined by a set of protocols, such as the consensus protocol, agreement on tokens emission rules, deterministic transactions processing protocol, and so on. 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 code acts as a protocols definition 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, and even its 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 for the development of more blockchain protocols and clients.

1.1 Our Contribution

In this paper we report on the design and implementation of a suite of abstract property tests which are implemented in the Scorex framework to ease checking whether a blockchain client satisfies specified laws. 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 have integrated the tests into a prototype implementation of the TwinsCoin Alex notes : cite cryptocurrency, which has two types of blocks. Alex notes : enhance

1.2 Property-based testing

In this section, we give 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 the 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 $\mathbb{S}$. Hence, P is not a valid property over the domain $\mathbb{S}$.
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, ScalaTest and ScalaCheck (majorly for generator-driven property testing) for Scala-based programs. Mayank notes : add references for quickcheck and others 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 them separately. For example, in a proof-of-stake cryptocurrency a balance sheet structure, which is heavily influenced by a transaction format, is used in a consensus protocol.

[1] Section 2 of the whitepaper

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 2.0 [?] by IOHK etc. We have chosen Scorex 2.0, as it 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 a 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*⟩. A persistent modifier triggers updates to the *node view*, more precisely the *history* and *minimal state* components of it. The minimal state is a data structure and a corresponding interface providing the ability to check the 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 their inclusion into blocks. Such a design, described in details in Section Alex notes : ref, gives us a possibility to develop an abstract testing framework where it is possible to state contracts for the quadruple entities without knowing details of their implementations.

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 a popular method of program verification (though testing still remains more prevalent). 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 this, 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 [?] Mayank notes : add reference

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 unit level, integration level and system level, before the software is deployed.

End-to-End testing is also performed, usually after system testing (sometimes it is seen as a type of system testing as well), to validate correct flow spanning different components of the software in real world use cases. 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. Following this, 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 [?] Mayank notes : add 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 Structure of the Paper

The paper is organized as follows. In Section 2 we give details on architecture of the Scorex framework. In Section 3 we describe our approach to property-based testing of blockchain system properties. Alex notes : finish

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 splits a node's local view into the following four parts:

[2] concretely, its honest nodes. We skip the notion of adversarial behavior for now.

- *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 a *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 a *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.
- *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 parts of local views to be synchronized across the network. For this, nodes run a consensus protocol to form a proper history, and the history should result in a valid minimal state when persistent modifiers contained in history are applied to a prehistorical state.

The whole node view quadruple is to be updated atomically by applying either 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 holds the quadruple <*history, minimal state, vault, memory pool*> and processes its updates atomically is called a *node view holder*. The holder is implemented as an actor Alex notes : link, so that all received messages are processed in sequence, even if received from multiple threads. If the holder gets a transaction, it updates the vault and the memory pool with it. Otherwise, if the holder gets a persistent modifier, it first updates the history by appending the modifier to it. If appending is successful (i.e. if the modifier is syntactically valid), the modifier is then applied to the minimal state. Alex notes : finish, write about forks

As an example, we consider the cryptocurrency Twinscoin, which has a hybrid Proof-of-Work and Proof-of-Stake consensus protocol. Scorex has a full-fledged Twinscoin implementation as an example of its usage. There are two kinds of persistent modifiers in Twinscoin: 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 an abstract blockchain-like system design. For extensive testing, we test 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.

History is a log of *persistent modifiers*. A modifier is persistent in the sense that it has a unique identifier, and it is always possible to check if the modifier was ever appended to the history (by presenting its identifier). There are no requirements on a modifier structure, besides the requirements to have a unique identifier and at least one parent (referenced by its identifier). If a modifier could be appended to a history, we call it *syntactically* valid. In addition to syntax of the blockchain, there are some stateful semantics, and minimal state takes care of them. That is, all nodes agree on some pre-historical state S_0 and then by applying the same sequence of persistent modifiers $m_1, \ldots, m_k$ in a deterministic way, all the nodes get the same state $S_k = apply(\ldots apply(apply(S_0, m_1), m_2), m_k)$ if all the $m_1, \ldots, m_k$ are *semantically* valid; otherwise a node gets an error on the first application of a semantically invalid persistent modifier. From our mere abstract point of view, the goal of obtaining the state S_k is to check whether a new modifier m_{k+1} will be correct or not. Thus the minimal state has very few mandatory functions to implement, such as $apply(\dot{})$. For a user running a node, the goal to run it is usually to get valuable user-specific information. For that, the vault component is used which has the only function to update itself by scanning a persistent modifier or a *transaction*. A transaction, unlike a persistent modifier, has no mandatory reference to its parents; also we consider that a transaction is not to be applied to the history and minimal state. To store transactions (temporarily, before them being included into persistent modifiers), the memory pool is needed.

3.1 Generators

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

- Syntactically valid (respectively, invalid) modifier, which is valid (respectively, invalid) based on the node's local view of history.
- Semantically valid (respectively, invalid) modifier, which is valid (respectively, invalid), based on the 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 concrete implementations for these generators. To generate syntactically valid modifiers, we generate a Proof-of-Work (PoW) block if a previous pair of *<PoW block, PoS block>* is complete, otherwise we generate a new Proof-of-Stake (PoS) block. It can be noted that, in TwinsCoin, transactions are only recorded in PoS blocks. A minimal state in our TwinsCoin implementation, similarly to Bitcoin, is defined as a set of unspent transaction outputs, and deterministically determines the validity of an arbitrary transaction, or a sequence of transactions. To generate semantically valid modifiers, we generate a PoS block including transactions based on unspent transaction outputs. A totally valid modifier generator has a matching PoS block among a syntactically valid and a semantically valid modifier. Generated transactions take as input unspent outputs and return as output a set of new randomly generated outputs.

3.2 Implicit properties

We implicitly define some properties. For example, the existence of a generator for a totally valid modifier for any given correct history and valid minimal state assumes that it is always possible to make a progress. Alex notes : continue, what else properties do we implicitly assume?

3.3 Simple properties

We can define a lot of trivial properties. For example, if a modifier has been appended to a history, it is always possible to get it at some later point of time, for any implementation of the history interface and any syntactically valid modifier. In opposite, if modifier is syntactically invalid, it should be not possible to get it by requesting from history. Similarly, if a persistent modifier is semantically valid against a minimal state, the former could be applied to the latter. Even more, after application we can, for example, to roll the application back and successfully apply the persistent modifier again,

3.4 Forking

Processing forks could be a complicated issue, making testing inevitable here. We proceed by describing the way in which we implement forking in Scorex. When a persistent modifier is appended to the history, it returns (if the modifier is syntactically valid) *progress info* structure which contains a sequence of persistent modifiers to apply as well as a possible identifier of a modifier to rollback to the state before application of this sequence of persistent modifiers. For efficiency reasons, the minimal state is usually limited in maximal depth for a rollback, so the rollback could fail (this situation is probably unresolvable in a satisfactory way without a human intervention).

We have implemented forking tests for the node view holder. The tests are checking that shorter sequence of totally valid persistent modifiers is not resulting in a fork, a longer sequence leads to a fork. Alex notes : add

4 Conclusion