# [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. We provide examples of some laws.

## 1 Introduction

## 2 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 is one of the most popular methods of program verification. Formal verification is used to validate the correctness of a software module by modeling its behavior based on a set of formal methods, which are mathematical models specifying the intended functional behavior. 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 behavior of the system is modeled. Finally the actual program is verified against this behavior 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]. Properties like safety and liveness are often modeled and verified using finite state

machines. For example, in order to check the security mechanism inside a software, states can be divided into secure and non-secure states and then an analysis around the set of reachable states can prove the software security.

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 behavior 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 sandwich approach. Big-bang is the most naive approach where all the modules are combined at once and then tested, followed by top-down approach, where firstly the main routine is tested and then other subroutines are added with each level. In bottom-up approach the most independent subroutines are first tested and then they are integrated gradually with each level, which is followed by the sandwich approach which is a combination of top-down and bottom-up 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 behavior 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).

### 2.1 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. It is trivial to understand that in order to automate the process of program testing at scale, having properties defined for all the different components of the program largely optimises the whole process.
Since it might not be the case everytime that the random data points needed for property testing are indeed the primitive data types, so in such cases generators are defined, which are simple functions that usually generate some output based on input parameters, to facilitate data generation for testing. A simple example of a generator $G$ can be:

$$G(X) \rightarrow Y, \text{ where } X = (x_1, x_2, \ldots, x_{n+1}) \text{ and } Y = \{x | x\%2 = 0\}$$

Generators can automate the process of random data point generation on virtually any data type, which can then be verified against the respective properties to yield the data points, if any, which falsify the predicate.
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.

### 2.2 Scorex core concepts

The idea of a modular design for a cryptocurrency was first proposed by Goodman in Tezos whitepaper [**?**]. The whitepaper (Section 2 of it) proposes to break a cryptocurrency design into 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 structure, is used in a consensus protocol. To split the layers clearly, 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 transaction 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 inital 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.

The whole node view quadruple is to be changed 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 coin developer needs to provide implementations for the abstract parts of the quadruple as well as a family of persistent modifiers.

## 3 Scorex Architecture

Scorex is breaking node view into for following parts.

- history contains log of persistent modifiers
- minimal state is a structure enough to
- vault is user-specific information
- memory pool contains temporary objects to be packed into persistent modifiers

## 4 Blockchain properties

## 5 Examples of properties tests

## 6 Conclusion