Checking Laws of the Blockchain with Property-Based Testing

Alexander Chepurnoy[1], **Mayank Rathee**[2]

[1]Ergo Platform and IOHK Research
Sestroretsk, Russia
[2]**Department of Computer Science and Engineering**
Indian Institute of Technology (Banaras Hindu University)
Varanasi, India

March 20 @ IWBOSE'18

Motivation behind our work

- Cryptocurrencies have been gaining popularity in the recent times.
- It is now, more than ever, crucial that errors in client implementations of cryptocurrencies are detected and corrected.

Motivation behind our work

- Cryptocurrencies have been gaining popularity in the recent times.
- It is now, more than ever, crucial that errors in client implementations of cryptocurrencies are detected and corrected.
- Bugs in reference client implementations can even propagate into alternate implementations. (Which are using the reference implementation as the definition of the protocols)

Motivation behind our work

- Cryptocurrencies have been gaining popularity in the recent times.
- It is now, more than ever, crucial that errors in client implementations of cryptocurrencies are detected and corrected.
- Bugs in reference client implementations can even propagate into alternate implementations. (Which are using the reference implementation as the definition of the protocols)
- There have been times when the famous Bitcoin network got affected by these bugs.
- The revelation of these bugs will only become more common with increasing demand and usage.

Our contribution

- A suite of generic property-tests designed to check if a blockchain client implementation is sane.
- Formally describing some essential properties which should be satisfied by any cryptocurrency and blockchain client implementation.

Software testing

- Conventionally, software testing requires writing some specific tests based on some examples.

Software testing

- Conventionally, software testing requires writing some specific tests based on some examples.
- Many bugs usually propagate through testing if the developer of code himself tries to write the tests.

Software testing

- Conventionally, software testing requires writing some specific tests based on some examples.
- Many bugs usually propagate through testing if the developer of code himself tries to write the tests.
- If someone else tries to write the tests then they can miss some behavioural aspects of the program.

Software testing

- Conventionally, software testing requires writing some specific tests based on some examples.
- Many bugs usually propagate through testing if the developer of code himself tries to write the tests.
- If someone else tries to write the tests then they can miss some behavioural aspects of the program.
- A test writer might take some assumptions regarding how the input is being processed by the function under testing, if he considers it as a black-box.

Software testing

- Conventionally, software testing requires writing some specific tests based on some examples.
- Many bugs usually propagate through testing if the developer of code himself tries to write the tests.
- If someone else tries to write the tests then they can miss some behavioural aspects of the program.
- A test writer might take some assumptions regarding how the input is being processed by the function under testing, if he considers it as a black-box.
- What happens to the tests which are heavily coupled with code of the program (which they test) when the program code changes?

Property-based testing

- More promising - model testing in a way that captures the idea of "how the program is intended to behave rather than how you think it might behave".

Property-based testing

- More promising - model testing in a way that captures the idea of "how the program is intended to behave rather than how you think it might behave".

Property

Consider a predicate $P : \mathbb{I} \rightarrow Boolean$
If $\forall X \in \mathbb{D} \subset \mathbb{I}$ we have that $P(X) = true$.
Then we say that predicate P defines a **property** over $\mathbb{D}$.

- In Property-based testing, properties are defined which should hold valid based on the intended behaviour of the program/module under testing.

Property-based testing II

- What happens during property-based testing?
  - During testing, the runner tries to find examples which falsify the specified properties.
  - If such a counter-example is found, then simpler versions, similar to those of the counter-example, which also falsify the predicate are searched and reported.

Property-based testing II

- What happens during property-based testing?
  - During testing, the runner tries to find examples which falsify the specified properties.
  - If such a counter-example is found, then simpler versions, similar to those of the counter-example, which also falsify the predicate are searched and reported.

Example

For a program which works only for inputs in $\mathbb{N}$, if the runner finds that "-5" falsifies the predicate, it can try a simpler example with similar arguments (like being in $\mathbb{Z}^-$) and can hence report that "-1" also falsifies.

Property-based testing III

- Since property-based testing treats the program code as a black-box, it does not suffer from **coupling** issues when the program code is modified.

Property-based testing III

- Since property-based testing treats the program code as a black-box, it does not suffer from **coupling** issues when the program code is modified.
- It is very convenient way of automating the task of testing. We just need to write the properties once and for all.
- With just a few lines of code, an extensive testing of software can be performed.

Property-based testing III

- Since property-based testing treats the program code as a black-box, it does not suffer from **coupling** issues when the program code is modified.
- It is very convenient way of automating the task of testing. We just need to write the properties once and for all.
- With just a few lines of code, an extensive testing of software can be performed.
- Checking **race conditions** with property-based testing is way easier than writing tests explicitly.

Popular libraries for property-based testing

- QuickCheck for Haskell-based programs.
- ScalaCheck for Scala-based programs, inspired by QuickCheck. **We used ScalaCheck to build our test suite**.
- ScalaTest for Scala-based programs.
- JUnit-QuickCheck for Java programs.
- theft of C programs.

A case study of Erlang QuickCheck

- Erlang Factory, in March'16, reported that they tested project FIFO, which is an open-source Cloud Management system, against some properties written with Erlang QuickCheck.

A case study of Erlang QuickCheck

- Erlang Factory, in March'16, reported that they tested project FIFO, which is an open-source Cloud Management system, against some properties written with Erlang QuickCheck.
- Project FIFO has 60,000 lines of code which was accompanied by just 460 lines of QuickCheck code.

A case study of Erlang QuickCheck

- Erlang Factory, in March'16, reported that they tested project FIFO, which is an open-source Cloud Management system, against some properties written with Erlang QuickCheck.
- Project FIFO has 60,000 lines of code which was accompanied by just 460 lines of QuickCheck code.
- 25 errors spanning timing errors, race conditions, type errors, logical errors, fault handling and **even a hardware error** among other software errors were found.

A case study of Erlang QuickCheck

- Erlang Factory, in March'16, reported that they tested project FIFO, which is an open-source Cloud Management system, against some properties written with Erlang QuickCheck.
- Project FIFO has 60,000 lines of code which was accompanied by just 460 lines of QuickCheck code.
- 25 errors spanning timing errors, race conditions, type errors, logical errors, fault handling and **even a hardware error** among other software errors were found.
- It is trivial to see that any decent test suite would have more lines of code compared to 460 lines here and would still report less errors than QuickCheck.

Property-based testing of generic systems

- Property-based testing for programs built over generic frameworks is even advantageous.
- Some generic frameworks can provide already implemented properties which should be satisfied by every application build on top of them.

Property-based testing of generic systems

- Property-based testing for programs built over generic frameworks is even advantageous.
- Some generic frameworks can provide already implemented properties which should be satisfied by every application build on top of them.

Generator

Routine which returns a sequence of values, one at a time, such that the caller can use them to generate data points from a non-primitive or user-defined data type

Property-based testing of generic systems

- Property-based testing for programs built over generic frameworks is even advantageous.
- Some generic frameworks can provide already implemented properties which should be satisfied by every application build on top of them.

Generator

Routine which returns a sequence of values, one at a time, such that the caller can use them to generate data points from a non-primitive or user-defined data type

- To check an application for sanity, the developer just needs to write generators to generate random data points.

Property-based testing of Blockchain systems I

- Blockchain systems built on top of a generic framework called Scorex can also gain from this. **We have deployed our tests over TwinsCoin cryptocurrency which is built over Scorex**.

Property-based testing of Blockchain systems I

- Blockchain systems built on top of a generic framework called Scorex can also gain from this. **We have deployed our tests over TwinsCoin cryptocurrency which is built over Scorex**.
- Building a generic testing suite specifically for just one cryptocurrency beats the whole purpose of the testing suite being "generic".

Property-based testing of Blockchain systems I

- Blockchain systems built on top of a generic framework called Scorex can also gain from this. **We have deployed our tests over TwinsCoin cryptocurrency which is built over Scorex**.
- Building a generic testing suite specifically for just one cryptocurrency beats the whole purpose of the testing suite being "generic".
- A lot of modular and open-source cryptocurrency frameworks have been proposed, with their only purpose - **speeding up the development time it usually takes to build a new blockchain system.**

Property-based testing of Blockchain systems II

- We have chosen Scorex framework by IOHK, which promises a **more intuitive partitioning** between the different component protocols like - network, transaction and consensus.

Property-based testing of Blockchain systems II

- We have chosen Scorex framework by IOHK, which promises a **more intuitive partitioning** between the different component protocols like - network, transaction and consensus.
- Along with this, Scorex also promises a very natural **support for more complex linking data structures**, other than the conventional blockchain like a Directed Acyclic Graph of blocks (SPECTRE by Sompolinsky, Lewenberg and Zohar 2016) or a Graph of cross-verifying transactions (Blockchain-Free Cryptocurrencies by Boyen, Carr and Haines 2016).

Scorex: Introduction

In Scorex, a client's local view of the system is divided into - **history, minimal state, vault, memory pool**.

Scorex: Introduction

In Scorex, a client's local view of the system is divided into - **history, minimal state, vault, memory pool**.

- *history* - It is an append-only registry for persistent modifiers (analogous to a block in Bitcoin), which must have a unique identifier to query their existence and must point to at least one parent, so that an ordering can be defined on history. **A persistent modifier may or may not contain transactions**.

Scorex: Introduction

In Scorex, a client's local view of the system is divided into - **history, minimal state, vault, memory pool**.

- *history* - It is an append-only registry for persistent modifiers (analogous to a block in Bitcoin), which must have a unique identifier to query their existence and must point to at least one parent, so that an ordering can be defined on history. **A persistent modifier may or may not contain transactions**.
- *minimal state* - It is a data structure which tells if a persistent modifier is valid at a particular point in time, such that all the nodes in the network having the same history, tell the same answer. If all nodes in the network agree on a historic state S_0, then if we apply the same sequence of modifiers, we get to the same state S_k. Once nodes have arrived at this state, they can tell if a modifier m_{k+1} is valid w.r.t to S_k or not. For example, the state S_k can dictate the current money that an account holds and can be used to see if a new modifier which encapsulates transactions is valid or not.

Scorex: Introduction

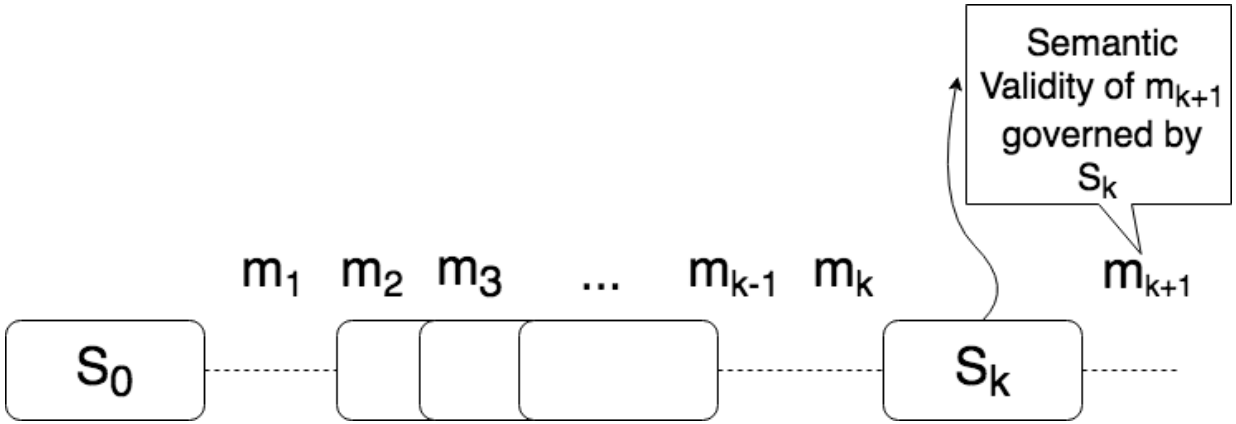

Figure: Minimal state progression

Scorex: Introduction

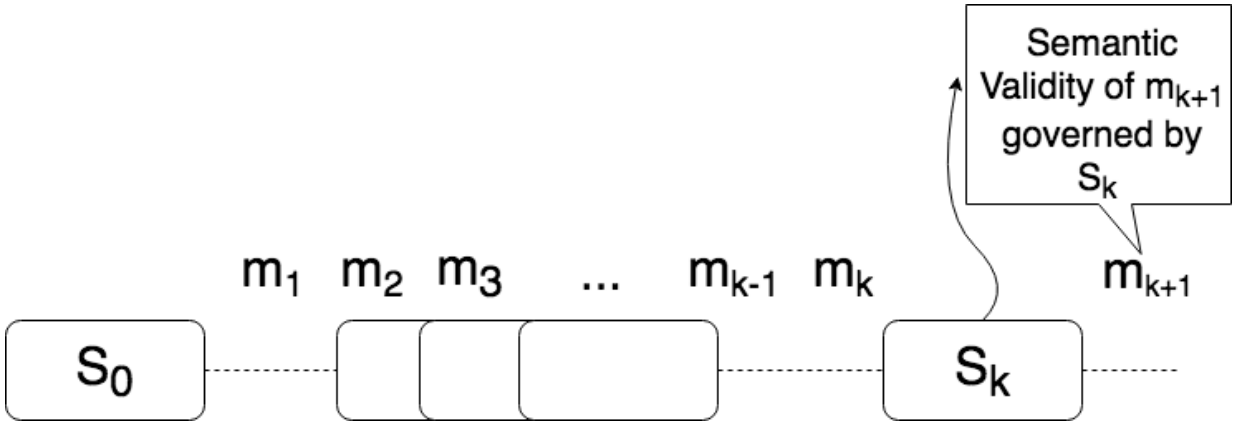

Figure: Minimal state progression

- *vault* - It contains some user/account specific information. It is maintained by the user running the node and is updated by scanning a persistent modifier, a transaction or at the time of a rollback.

Scorex: Introduction

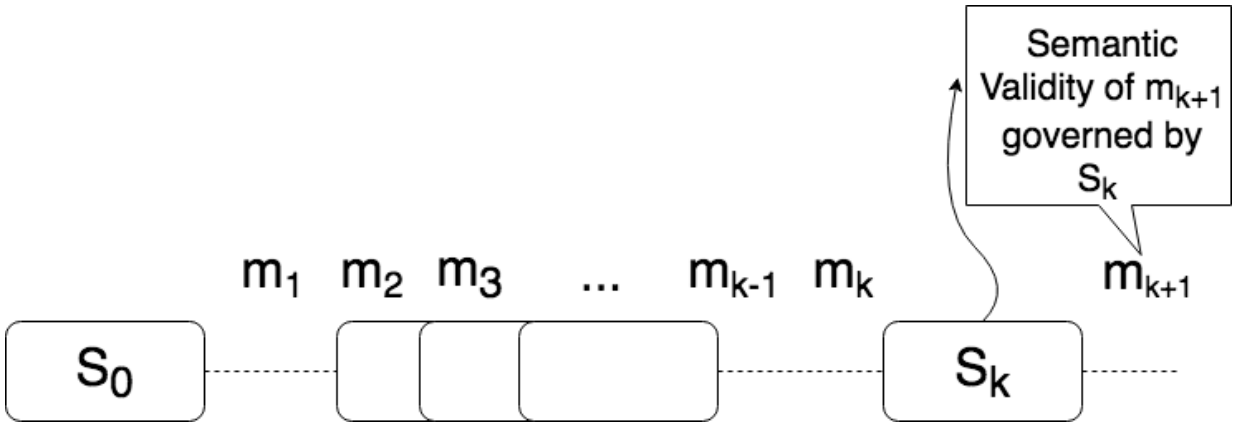

Figure: Minimal state progression

- *vault* - It contains some user/account specific information. It is maintained by the user running the node and is updated by scanning a persistent modifier, a transaction or at the time of a rollback.
- *memory pool* - It is used to store unconfirmed transactions which will be later added into a persistent modifier.

Node view quadruple

- A node view quadruple is - $<$*history*, *minimal state*, *vault*, *memory pool*$>$.

Node view quadruple

- A node view quadruple is - $<$*history, minimal state, vault, memory pool*$>$.
- A consensus protocol is run among the nodes so that they can form a proper, global and consistent history view.
- All updates to the node view quadruple should happen atomically to maintain consistency and harmony.

Persistent modifiers

In Scorex, valid persistent modifiers are divided into 2 types -

- *Syntactically valid* - which are valid according to the history. For example, in Bitcoin, a block is syntactically valid if its header is correct and also contains a correct proof-of-work. But, it can still have incorrect or faulty transactions which are taken care of by stateful semantic validity (done by minimal state).

Persistent modifiers

In Scorex, valid persistent modifiers are divided into 2 types -

- *Syntactically valid* - which are valid according to the history. For example, in Bitcoin, a block is syntactically valid if its header is correct and also contains a correct proof-of-work. But, it can still have incorrect or faulty transactions which are taken care of by stateful semantic validity (done by minimal state).
- *Semantically valid* - which are valid according to the current minimal state instance. For example, in Bitcoin, a block is semantically valid if it contains all correct transactions, i.e. the senders should have more money in their account than they intend to send and the senders and receivers should have valid and active accounts.

A persistent modifier is called *totally valid*, if it is both semantically valid and syntactically valid.

Property-based Testing of Blockchain Client

- We have implemented **59** property tests in total.

Property-based Testing of Blockchain Client

- We have implemented **59** property tests in total.
- Our tests require random object generators of - syntactically, semantically and totally valid and invalid modifiers, transactions, history instance, minimal state instance, vault instance and node view holder instance.

Property-based Testing of Blockchain Client

- We have implemented **59** property tests in total.
- Our tests require random object generators of - syntactically, semantically and totally valid and invalid modifiers, transactions, history instance, minimal state instance, vault instance and node view holder instance.
- These generators are then used by our tests to tell sanity of the blockchain client implementation on certain ground examples which we will see next.

Examples of Property tests

Of the 59 tests that we have implemented, we will go through a few of them.
The tests are grouped into 5 classes - *Memory pool tests*, *History tests*, *Minimal state tests*, *Node view holder tests and Forking tests*

Examples of Memory pool tests

- *Memory pool should be able to store enough number of transactions.*

Examples of Memory pool tests

- *Memory pool should be able to store enough number of transactions.*
- *Time to filter valid and invalid transactions from the memory pool should be very fast.* This test revealed that the implementation of TwinsCoin cryptocurrency, which we used to run our tests on because it is built on top of Scorex, was inefficient at filtering transactions from memory pool.

Examples of History tests

- *A syntactically valid modifier should be applicable to history and after that it should be available by its identifier when history is queried.*

Examples of History tests

- *A syntactically valid modifier should be applicable to history and after that it should be available by its identifier when history is queried.*
- *Modifier never appended to the history should not be available on request from the history interface.*

Examples of Minimal state tests

- *Application of a semantically valid modifier and then rollback should lead to the same state*. In this test, it is required to check that the state after the rollback is exactly the same as the one before the application of the modifier.

Examples of Minimal state tests

- *Application of a semantically valid modifier and then rollback should lead to the same state*. In this test, it is required to check that the state after the rollback is exactly the same as the one before the application of the modifier.
- *Application of a semantically valid modifier after a rollback should be successful*. This test ensures that after recovering from the rollback, the system can continue its operation normally.

Examples of Node view holder tests

- *A totally valid persistent modifier should successfully update the minimal state and the history instances*. In this test, once the modifier is applied to the node view holder, we check that history indeed contains the modifier and that the version of minimal state is equal to modifier's identifier (which is a unique value).

Examples of Forking tests

- *Application of a longer sequence of totally valid modifiers should replace the shorter sequence with both starting from a common ancestor in the history*. In this test, we first apply a shorter sequence of valid modifiers and then starting from a common block, we try to apply a longer sequence and see if the last block now is indeed the last block of the longer sequence.

Conclusion

- We have developed a suite of 59 such property tests which should hold true for almost all possible blockchain based clients.
- For any blockchain client built over Scorex framework, our testing suite can be easily used by supplying it with concrete implementations of some generators.
- Our list of property tests can also be used as a reference if someone wants to develop a testing suite targeted to a specific client implementation.