Model-based testing with TLA$^+$ and APALACHE [*]

Andrey Kuprianov[1] and Igor Konnov[1]

[1] Informal Systems, Vienna, Austria
Email: {andrey,igor}@informal.systems

Abstract

In this talk we present our approach to model-based testing with TLA$^+$ and the model checker APALACHE. We have created the infrastructure that facilitates model-based testing for the Tendermint Light Client protocol. The system engineers find the approach much more appealing and apply it more willingly than hand-written integration tests.

1 Overview

Our model-based testing (MBT) finds its rightful place in the verification-driven development process (VDD) that we are developing at Informal Systems. Specifically, we are designing several blockchain protocols for Tendermint and Cosmos ecosystem [3] and implementing them in Rust. In VDD, we start with a semi-formal protocol specification in English, then formally specify the protocol in TLA$^+$ and check it with APALACHE [1, 4]. The TLA$^+$ specification is used as a concise reference model for clarifying the ambiguities between the English specification and the implementation. MBT is an automated approach to integration testing of the implementation. The tests are expressed as simple assertions about the computation history in pure TLA$^+$.

Listing 1: Example of a test goal in TLA$^+$

```
Test2NoSuccess ≜
  ∧ state = "finishedSuccess"
  ∧ ∃s1, s2 \in DOMAIN history:
     s1 ≠ s2 ∧ history[s1].verdict ≠ "SUCCESS" ∧ history[s2].verdict ≠ "SUCCESS"
```

Listing 1 shows an example of a test for the Light client protocol [2], which establishes a trust in a blockchain block retrieved from an untrusted source. The test specifies a behavior, in which the protocol finishes with “success”, despite passing through two distinct intermediate states that failed block verification. Our MBT framework generates a complete integration test for the Light Client protocol: e.g., in a Tendermint network with 4 validators these 6 lines of TLA+ are translated to more than 500 lines of the integration test in JSON with all necessary details: private and public keys, signatures, timestamps, etc.

In our talk, we focus on three aspects: (1) ease of use, (2) test harness, and (3) MBT as the link between the specifications and the implementation.

[*]Supported by Interchain Foundation (Switzerland).

Ease of use. Our preliminary experience on implementing and applying MBT to the Light Client protocol shows that this approach is accessible both to the verification engineers and system engineers. While our tests are quite abstract, they capture the behavior under test. In sharp contrast to the manually written integration tests, our tests leave many details underspecified. These missing details are instantiated differently for every run of the model-based test by the model checker and the test harness. Alternatively, the tool can enumerate all concrete scenarios that fall under the abstract test.

The conceptual simplicity of MBT is amplified by the rich syntax and semantics of TLA$^+$. We concisely model the complex aspects of several Tendermint protocols, their correctness properties, and test goals. Several developers who had no previous experience with TLA$^+$ were able to follow the TLA$^+$ specifications that were written by the researchers. The specifications helped the developers in understanding non-trivial assumptions about the protocol and its environment.

Test harness. Not surprisingly, to apply MBT, one has to develop additional infrastructure. However, we found that large parts of that infrastructure are reusable. In the Light Client project, we have implemented the following components for MBT:

- Apalache support for importing and exporting specifications in the JSON format (*reusable*).
- JSON Artifact Translator that maps a TLA$^+$ behavior (a counterexample) to an integration test in Rust (*reusable*).
- Test generator that maps specification states to the implementation states (*reusable* within the products that share the same blockchain data structures).
- Test driver that orchestrates the above infrastructure components and runs them within the standard continuous integration framework. This is done with the standard command `cargo test` in Rust.

Implementation of the above infrastructure took us about three person-months of work, which is quite a reasonable cost taking into account the reusability of most of the components.

Link between the specifications and the implementation. Our initial approach to VDD process left the gap between the specifications and the implementation. Indeed, after some time the artifacts at different levels of abstraction start to diverge, namely, the English specification, the TLA$^+$ specification, and the implementation. In this talk, we will show examples of how MBT helped us to eliminate this divergence by running simple model-based tests. As a result, MBT plays the role of the missing link between the specifications and the implementation.

References

[1] Apalache model checker. Informal Systems, 2020. `https://github.com/informalsystems/apalache`.

[2] Specification of the tendermint light client. Informal Systems, 2020. `https://github.com/informalsystems/tendermint-rs/tree/master/docs/spec/lightclient`.

[3] Ethan Buchman and Jae Kwon. Cosmos whitepaper: a network of distributed ledgers, 2016. `https://cosmos.network/resources/whitepaper`.

[4] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. TLA+ model checking made symbolic. *Proc. ACM Program. Lang.*, 3(OOPSLA):123:1–123:30, 2019.