Kayfabe: Model-Based Program Testing with TLA+/TLC

Star Dorminey, Microsoft Azure WAN, August 2020

Abstract

While `TLA+/TLC` is an ideal system for designing and checking protocol specifications, showing that a program implements the specified protocol remains challenging. Formal methods for doing so exist on a continuum. Trace verification, as proposed by Ron Pressler [1], verifies an observed sequence of execution traces by as a refinement of the model under test. This monitors programs for specification compliance in real-time, but can't catch bugs ahead of time. At the other extreme, the correctness and liveness of programs such as `seL4` have been proven as refinements of their specifications [2], but this is extremely labor-intensive. Model-based testing systems that tour state spaces [3] offer an efficient compromise between these two extremes.

As a middle-of-the-road option, we propose having the program under test explore the entire state space of a finite `TLC` model, checking at each step that the program's state is a refinement of the model's state. We describe `Kayfabe`, a model-based testing system that implements this approach, and discuss its design, success and shortcomings in verifying an assortment of distributed protocol implementations in C#.

Kayfabe tests are composed of the following:

- `TLA+` specification and model parameters.
- A test harness consisting of:
    - An injective mapping from current program state -> `TLA+` model state.
    - An *action successor function*, which returns a list of predicted successor model states resulting from possible actions, along with callbacks for executing them.
    - A way of resetting the program.

When the test is executed, the program is stepped through every state (or optionally transition) in the model, and checked for compliance with the spec.

In the first phase, Kayfabe's *route solver* calculates an efficient inspection tour that covers that whole state space, minimizing states revisited. We show how this reduces to the Traveling Salesman Problem (TSP) via a graph transformation using single-source shortest paths, which currently severely restricts model size, but polynomial-time approximation algorithms should make this tractable for larger models. This phase produces a list of tours that collectively span the state space.

In the second phase, Kayfabe resets the program and begins the tour. At each step, we:

1. Check that current program state matches expected model state.
2. Run the *action successor function* to get predicted successor states, checking that those match the successors of the model.

3. Select the single successor state that matches the next state on the tour, and run it.
4. If we're at the end of this tour, reset the program and continue with the next tour, otherwise repeat.

We used Kayfabe to test `StateManagerLibrary` (SML), an `Apache Curator`[4]-like library of distributed systems primitives built on top of a `ZooKeeper`[5]-like database. Each SML recipe was designed with a `TLA+` specification, so we created Kayfabe harnesses for each and tested them independently. We observed the following benefits:

- Repeatable, deterministic testing of highly asynchronous/concurrent code.
- Assurance that the program "implements" the model, in a sense.
- Specification stays up to date, or the build breaks.
- Caught and fixed a number of bugs.
- The program to model state binding was elegant and easy to maintain.

However, the harness had a few limitations:

- Symbolic model values were awkward to bind to concrete values.
- Modeling nonce or GUID-like values requires tricky mapping: we only care about uniqueness, not value, and this is difficult to express and bind.
- The model state omits some internal program state, so side-effects that change internal values may make the program sensitive to tour order, and means that some sequences of actions that cause the program to violate the model may remain untested.
- Since the route solver is NP-hard, models are constrained to only a few hundred states to remain feasible.

The computational complexity of the route solver is the largest hurdle at present. Replacing the default `or-tools` TSP solver [6] with Christofides algorithm would reduce overall computational complexity to `O(n^2 * log(n))` [7]. Larger models may be accomodated either by allowing "*warps*" -- resetting the program to arbitrary nodes, which makes route-solving trivial (this can also be thought of as expanding the set of initial states), and/or by invoking TLC as a coroutine alongside the exploration phase, rather than using a deterministic route solver.

References

1. Verify Software Traces Against a Formal Specification with TLA+ and TLC. *Ron Pressler (2018)*
2. seL4: Formal Verification of an OS Kernel. *Klein, Elphinstone et. al. (2009)*
3. Model driven code checking. *Holzmann, Joshi & Groce (2008)*
4. Apache Curator.
5. Apache ZooKeeper.
6. Google OR-Tools.
7. OR-Tools's Implementation of Christofides Algorithm.