Hillel Wayne Follow
I once kicked a pigeon.
Mar 14, 2017 · 10 min read

Formal Methods in Practice: Using TLA+ at eSpark Learning

How do we harden a system against race conditions? When it uses a complex cache? When it requires a global state? We could blanket our system in tests, hope we've covered every possible edge case, and pray that nothing new happens in production.

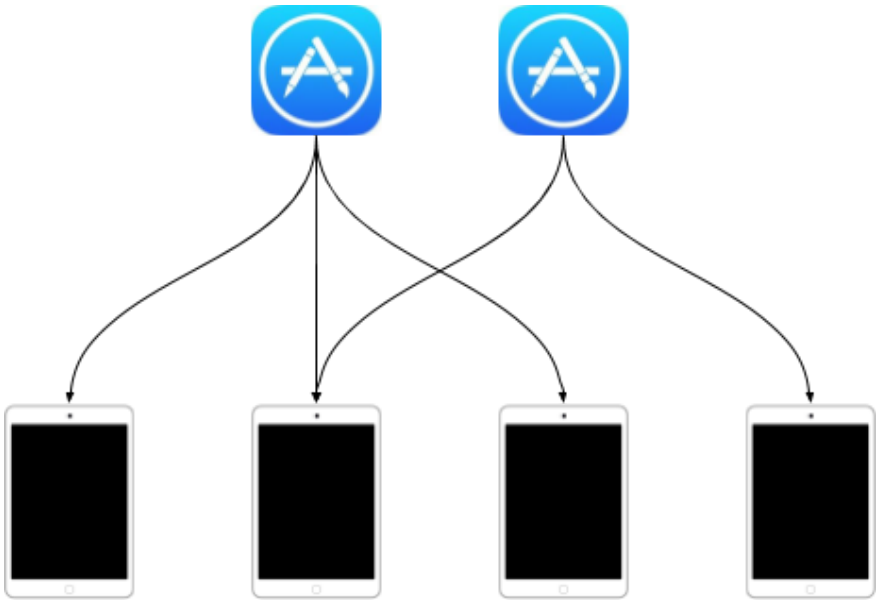
Or we could use formal methods.

We could write the abstract design in a machine-checkable *specification language*, determine the properties we want out of the design, and **prove** it matches our requirements. Some cases may need actual mathematical proof, but it is far more accessible to give the design a range of inputs and have it exhaustively explore all states and timelines for any sign of a mistake. It's not perfect, but almost everybody agrees that it's damn effective … in theory. In practice people say it's too difficult to use for anything less critical than MRIs or spacecraft. Nobody gives a real-life example of how an average startup used it.

So let's give an example of how an average startup used it.

Background

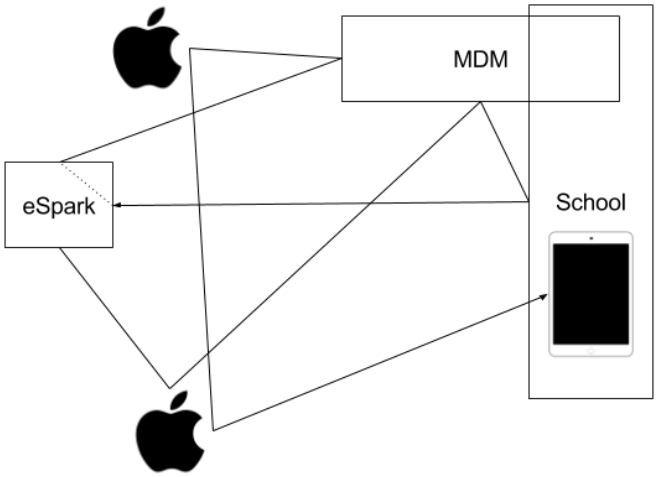
Since 2012 eSpark has offered individualized learning to elementary school students using iPads. We give students a brief quiz, determine their strengths and weaknesses, and give them a lesson to help them improve. At the core of this is a massive list of curated iPad apps, researched and selected by an in-house team of former teachers, that we add to each lesson. It's an effective product and I'm proud to be part of it.



A very rough overview of our product

The apps, while critical to the effectiveness of the product, are also the most complicated component. Each student needs a different set of apps, which change as they complete lessons. We have to be able to track and adjust them in near-real time, so that students aren't left without a next lesson. In extreme cases a student might need six apps in the morning and six completely different apps in the afternoon. We also have to be mindful of school budgets, so we have a limited number of the paid apps and shuttle them between students. Finally, since we need to do this at scale, the entire system has to run automatically with no teacher or admin intervention.

Besides all that, there's one more challenge: the Mobile Device Managers. These systems control all of the iPads in the school. If we want to add apps, remove apps, or even get the list of currently installed apps, we have to go through them. The MDMs are provided by third parties, each with its own API, control flow, and quirks. They also may be hosted by the schools themselves, which means we have to deal with performance and stability issues, too.



This is actually an oversimplification.

The Problem

One of our MDM partners determines which apps go to which devices using *app scopes*, allowing administrators to create imperative configurations such as “this app should be on these devices”. In order to determine which apps needed to be installed we have to check the device data. The delay between a device entering app scope and receiving the app could lead us to believe a device was missing an app when it was already in scope. In that case, we’d potentially try to add it to scope again.

The partner requested that we 1) send as few scope adjustments as possible (adding and removing devices from an app scope in bulk), and 2) not make any redundant changes in the bulk command. In other words, if we asked to add “Bluster” to five devices, none of those five could already be in the Bluster scope.

Oh, also: communicating with MDMs is I/O heavy so all API calls–updating app scope, getting device data and so on–has to happen asynchronously in an independent process. And device synchronization happens at different rates so we need to keep a running pool of devices for each app. And there needs to be a manual override for sales demos. And this system would directly impact over two million dollars in revenue. This is not the kind of system we can just fix in production.

So we decided to use formal methods.

. . .

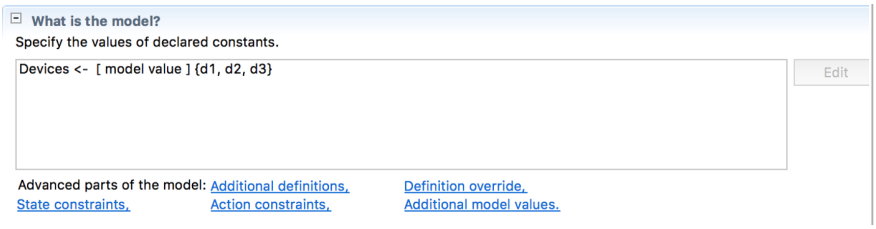
Our tool of choice is TLA+. First, we specify the system as a giant abstract state machine and list all of the possible transitions it can make. Then, we specify *invariants* of the system, properties that **every** state must have. Finally, we run the model checker, called **TLC**, on the spec. TLC exhaustively checks all possible states and raises an error if any state violates the invariant. Here's our initial design:

```
EXTENDS Integers, TLC, Sequences
CONSTANTS Devices

(* --algorithm BatchInstall
variables
  AppScope \in [Devices -> {0, 1}];
  Installs \in [Devices -> BOOLEAN];
  batch_pool = {};

define
  PoolNotEmpty == batch_pool # {}
end define;

procedure ChangeAppScope()
begin
  Add:
    AppScope := [d \in Devices |->
        IF d \in batch_pool THEN AppScope[d] + 1
        ELSE AppScope[d]
     ];
  Clean:
    batch_pool := {};
  return;
end procedure;

fair process SyncDevice \in Devices
```

This should look mostly familiar as a programming language, but there are a few subtle differences. First, `AppScope \in [Devices -> {0, 1}]` tells TLC that any number of the devices may already be in scope as part of the initial state. Real world expectations rarely start with a blank slate: the school may have preloaded some apps on devices, or we might be doing a later batch cycle. For reasons that will become apparent, we represent this as "each device is in scope either 0 times or 1 time." We could also use `[Devices -> BOOLEAN]`, where `BOOLEAN` is the set `{TRUE, FALSE}`.

Second, there is one TimeLoop process (representing periodic installs) and one SyncDevice process per device. In what order will they run? For each starting state, there are multiple *behaviors*, or timelines. TLC can choose in what order processes start, how they interleave while running, and even if they can crash mid-process. `Sync(d1) -> Sync(d2) -> Add -> Clean` is just as valid as `Sync(d2) -> Add -> Sync(d1) -> Clean`, and TLC will check both.

Finally, `Devices` is a *constant*. In addition to meaning we cannot change it at runtime, constants have an additional property in TLA+: we can define their value right before running checking a model. To test this with one device, we declare Devices to be the set `{d1}`. If we wanted to test with three devices, we could just as easily write `{d1, d2, d3}`. This allows us to make our model as large and as complicated as we need it to test our invariants.



What it looks like in the IDE

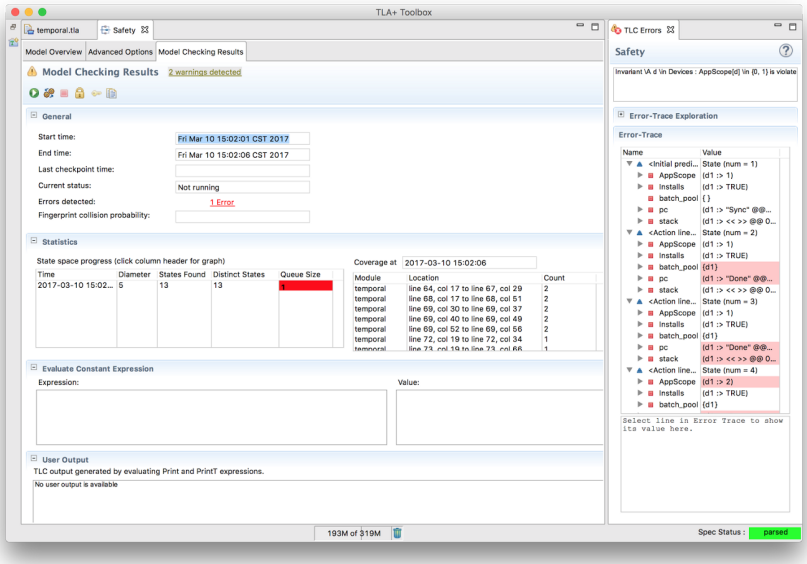
Otherwise, the spec is pretty straightforward. SyncDevice may add a device to the batch pool. At some point, TimeLoop runs, the devices in the batch pool are added to the AppScope, and then the batch pool empties.

Proving Safety

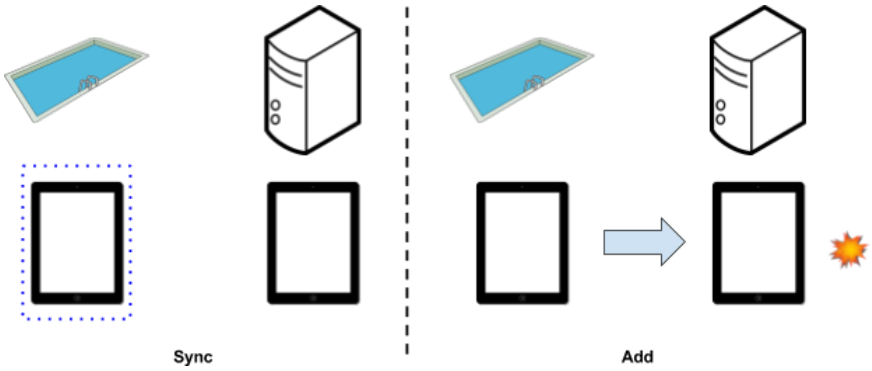
Invariants are statements that should always be true at every state in the system. When we check for *safety*, we're checking that it's impossible to reach a state where any invariant is violated. In our case, we can represent our invariant as follows:

```
\A d \in Devices:
  AppScope[d] \in {0, 1}
```

"For all devices, it must be scoped either 0 times or 1 time". If it is scoped 2 times, we added a device that was already in scope. When we test this with one device, we get the expected error:



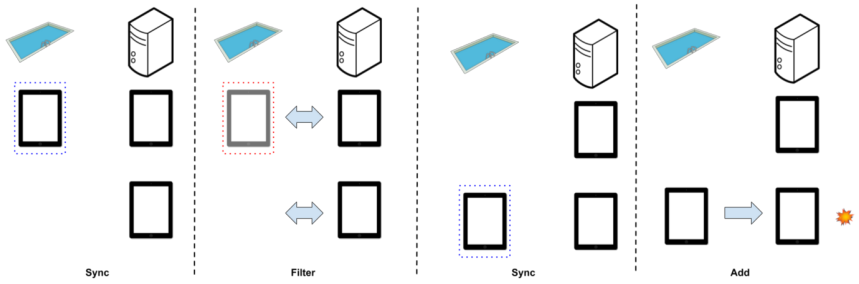
TLC provides an error trace. Future traces will be in links.



We don't have any logic to detect devices already in scope. A simple fix would be to pull the scope, then remove anything already in scope from the pool. If we do that, it works.

```
Filter:
    batch_pool := batch_pool \intersect {d \in Devices:
AppScope[d] = 0};
```

But what about for two devices? That would be difficult to check with unit tests. But with TLA+, it's trivial. We just add a second device to the model and watch it fail.



Error Trace

This failure involves a race condition. One device syncs, which means we can start the installation. As soon as we filter the pool, the **other** device syncs, putting it back in the pool in time for the add step. We ended up fixing this by caching the pool during the installation process and filtering and sending the cache. That way even if the pool updates after the process starts, we don't use the new data in our call.

```
procedure ChangeAppScope()
variables cache;
begin
  Cache:
    cache := batch_pool;
  Filter:
    cache := cache \intersect {d \in Devices: AppScope[d] =
0};
  Add:
    AppScope := [d \in Devices |->
        IF d \in cache THEN AppScope[d] + 1
        ELSE AppScope[d]
     ];
  Clean:
    batch_pool := {};
  return;
end procedure
```

Maintaining Safety

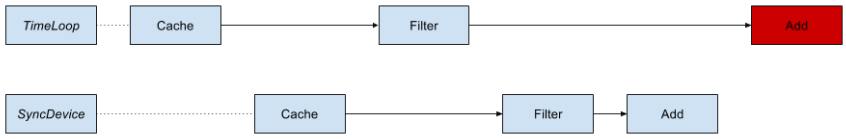
This system is incomplete: some apps are more important than others, and some apps are needed by more people than others. The best compromise to this is to add in "pool thresholds". If a device sync puts the pool size above the threshold, we should immediately start the installation process. That means that if 30 students need the same app, they get it in the next few minutes, instead of whenever the TimeLoop period ends.

We can specify that with an `either` statement. Whenever TLC hits the statement, it creates a separate timeline for each branch. Again, we

don't care how we trigger the threshold sync, just that it could potentially happen.

```
if PoolNotEmpty then
  either call ChangeAppScope();
  or     skip;
  end either;
end if;
```

When we add this, though, TLC finds a new error in the system.



Error Trace

Our system was built on the assumption that only one installation process can run at one time. When two can run, they can potentially make the same threshold cache. Then both filter the cache at the same time, leaving both installing the same device because it wasn't in scope. The first will succeed normally, but by the second process, we'll now be doing a double install.

There's a few different ways to fix this, but the simplest for us was putting a mutex on the installation process. While multiple sources could start the process, only one can run at a given time, which cleaned up the bug for us.

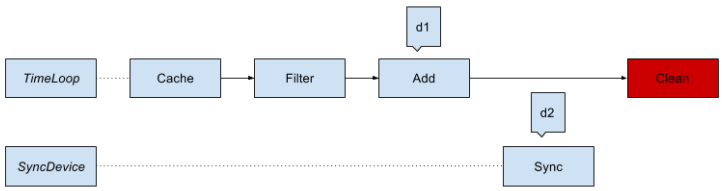
```
variables lock = FALSE;
\* ...
GetLock:
    await ~lock;
    lock := TRUE;
\* ...
Clean:
    lock := FALSE;
```

Proving Liveness

So far we've proved the system doesn't do bad things. However, we haven't yet proved it does what we want. Specifically, we want to ensure that if we want to install the app on a device, it eventually the device is in the app scope. Safety is proving bad things **don't** happen. *Liveness* is proving that good things **do** happen. This is what our liveness property looks like:

```
\A d \in Devices:
  Installs[d] ~> AppScope[d] = 1
```

That just says that if we want to install on a device, at some point in the future it's actually in scope. And, once again, TLC finds an error.



Error Trace

We want to install an app on two devices. The first device syncs and starts an install process. After we cache the pool, the second device syncs. Then, after the process finishes, we empty the pool, **including the second device**. The fix for this is pretty simple: instead of emptying the entire pool, just remove the cached devices from it. This means the entire install process is isolated from the outside world and works as we expect.

```
Clean:
  batch_pool := batch_pool \ cache;
  lock := FALSE;
```

You can see the complete spec here.

Results

Resounding success. We finished the system a week ahead of schedule and, while we did find some bugs, they were all implementation issues

we easily tracked down. That said, we did hit a few pain points with TLA+:

- It's very easy to get overconfident with it. As mentioned, we did have some bugs in production. One was a performance bottleneck we needed to fix. Another came from a requirement we forgot to model.
- TLA+ is slow. Since it exhaustively searches the state space, it's very easy to balloon a runtime to hours or days. With some optimization we got our specs below a minute each, but that did take some effort.
- TLA+ has a high learning curve. The entire concept of formal methods is new to a lot of people, plus the language itself has its warts and gotchas.

If you're interested in using TLA+, the best place to start is my Learn TLA+ beginner's guide, which is packed with exercises, real-world examples, and practical tips. It is not affiliated with eSpark and is purely a personal labour of love.

We wanted to see if formal methods were useful for practical problems and the answer is *absolutely*. The two days of TLA+ modeling more than paid for itself in delivering a complicated system faster and with fewer bugs. It takes time to learn, but the reward is worth it.

At eSpark Learning we're building the next generation of tools to help students succeed in school and in life. We believe in humility, compassion, and work-life balance. We're hiring!

Hillel is a senior software engineer at eSpark Learning. He's interested in software testing, civic tech, and generally being a contrarian. Find him on twitter as @hillelogram or at hillelwayne.com.