Learning TLA+ by Examples

Richard Tang

January 20, 2025

Contents

Chapter 1

Introduction

1.1 Catching Problems Early

Years ago, I worked on a propietary low power processor in an embedded system. The processor ran microcode featuring a custom instruction set. To enter (or exit) a low power state, a set of (possibly hundreds) instructions were executed. These instructions progressively puts the system in lower power state. For example: turn off IP A, then turn off IP B, then turn off the power island to the IPs, etc. To save cost and power, the low power processor had very limited debuggability support.

An experienced reader may start to notice some redflags.

When the system enters a low power state, it executes possibly hundreds of instruction. If the microcode attempts to access the memory interface when the power island has been shut off, the system would hang. Since the power island has been shut off, the JTAG port is also shut off, leaving the developer with *no way* of live debugging related problem. At this point the developer needs to siphon through (possibly hundreds) of instructions to catch invariant violation *manually*.

As one can imagine, maintaining the microcode was very expensive. Fortunately, the propietary low power processor only had a handful of instructions, I was able to reduce the maintenance cost by creating an emulator to verify the microcode prior to deploying it on target. The emulator tracks the system state on every command execution and confirm every step along the way none of the system invariants have been violated. This included stuff such as:

- Accessing memory interface after power off leads to a hang
- Accessing certain register in certain chip revision leads to a hang
- Verify IPs are shut off in the allowed order

The verification algorithm was implemented using a *depth-first search* algorithm, providing *100%* microcode coverage before deploy on target.

This was a very enlighting experience. To generalize, there are a class of problem where the solution has high *failure cost*. Solution such as lockless or waitfree data structure, distributed algorithms, OS scheduler are typically hard to reason about correctness, and tend to have high failure cost because related issues are difficult to reproduce and hard to debug. Good software engineering practice suggest we need to implement tests, but tests do not cover the entire solution space exhaustively.

How do we ensure the solution is correct by design?

1.2 The Generalized Problem

Fast forward to now: I stumbled across TLA+, a formalized solution of what I was looking for.

Leslie Lamport invented the TLA+ 1999, but TLA+ didn't appear to have caught on until the 2010's. My personal opinion is TLA+ was invented ahead of its time, and the problem complexity finally caught up in the past decade or so to allow TLA+ to visibly demonstrate its strength.

We are also at a point in the technology curve where vertical scaling is no longer practical, with CPU speed has plateau in the past decade or so. The industry is exploring horizontal scaling solution, such as hardware vendor focusing adding more CPU cores, or software vendors buying many low end hardware instead of a few high end hardware. This shifts the technology complexity from hardware to software, demanding concurrent software solution to maximize hardware resource utilization.

One slight problem: *human are bad at concurrent reasoning.*

Humans are fundamentally single-threaded machines. Reasoning things that execute in parallel is possible, but difficult. It is hard to enumerate all possible scenario in one's mind to ensure the design accommodates all the edge cases.

Consider a distributed system. The system is a cluster of independently operating entities and need to somehow collectively offer the correct system behaviour, while any one of the machines may receive instructions out of order, crash, recover, etc.

Consider a single producer multiple consumer lockless queue. The consumers may reserve an index in the queue in certain order, but may release them in different order. What if one reader is really slow, and another reader is super

fast and possibly lapse the slow reader?

Consider a OS scheduler with locks. Assume all the processes have the same priority - can a process starve the other processes by repeatedly acquire and release the lock?

The usual anti-pattern is to keep bandaiding the solution until bugs stop comming in - but how does anyone know if the solution is actually *correct by design*? To solve this problem, we must rely on tools to do the reasoning *for us*.

1.3 TLA+

TLA+ is a *system specification language*, with the intent to describe the system with implementation details removed. TLA+ allows designer to describe the system as a sequence of states. The designer can expresses transition condition from one state to another, describe invariants that must hold true in every state and liveness properties that the overall system should converge to. The key innovation of TLA+ is once the system is modeled as a finite state machine, the states can be *exhaustively* explored (via breath-first-search) to ensure certain properties are held through out the entire state space (either per state or a sequence of states).

1.4 Target Audience

TODO: describe target audience

1.5 Prerequisite

1.6 Book Layout

This book was motivated by the intent to solve problems. The book is designed to be example heavy with many chapter each represnting an problem that can be modelled using TLA+.

Examples are split into two categories: A set of examples written using native TLA+ syntax, and another set of examples written using PlusCal (C-like syntax). I believe they are useful under different use cases. The differences will be highlighted in their respective sections. All examples will follow a similar layout, covering the expected design process (eg. requirement, spec, safety and liveness properties).

Finally, there will be part that language reference portion that that discuss a few topics deserving extra attention. The intent is to be using this section of the book as a *reference*.

Chapter 2

TLA+ Primer

2.1 Design Intent

The key insight into TLA+ is modelling a system as a state machine. A blinking LED system can be described using a single variable with two states, LED being on or off. A simple digital clock can be represented by two variables, hour and minute and the number of possible states in a digital clock is $24 * 60 = 1440$. For example, 10:01 is the next state 10:00 can transition to. Extrapolating further, Asssume an arbitrarily system described by N variables, each variable having K possible values such arbitrary system can have up to N^K state.

For every specification, designer can specify *safety* proerty (or invariants) that must be true in *every* states. For example, in any state of the digital clock hour *must* be between 0 to 23, or formally described as $hour \in 0..23$. Similarly, $minute \in 0..59$. More generic invariant examples include: in any state, only one thread has exclusive access to a critical region, all variables in the system are within allowable value, the resource allocation manager never allocates more than available resources, etc.

Designer can also specify *liveness* property. These are properties that are satisfied by a *sequence of state*. One liveness property for the digital clock could be when the clock is $10:00$, it will eventually become $11:00$ ($10:00$ *leads to* $11:00$). More generic liveness property include: a distributed system eventually converges, the scheduler eventually schedules every tasks in the task queue, the resource allocation manager fairly allocates resources, etc.

TLC checks a TLA+ spec using *breath-first search* algorithm to explore *all* states in the state machine and ensure safety and liveness properties are upheld. TLA+ specifies the system using *propositional logic*.

2.2 Requirement

In this example, we will specify a *digital clock*. The digital clock has a few simple requirements:

- Two variables to represent state: hour and minute
- The clock increment one minute at a time
- The clock wraps around at midnight (ie. 23:59 transitions to 00:00)

2.3 Spec

The *Init* state of such system can be described as:

$$
\begin{aligned}
&\mathit{Init} \triangleq \\
&\quad \land \mathit{hour} = 0 \\
&\quad \land \mathit{minute} = 0
\end{aligned}
$$

$\triangleq$ is the *defines equal* symbol and $\land$ is the *logical and* symbol. The above TLA+ syntax can be read as *Init* state is defined as both hour and minute are both 0.

The spec also always include a *Next* definition, an *action formula* describing how the system transition from one state to another. Action formula contains *primed* variables what happens to the variable in its next state. The *Next* action for the digital clock can be defined as:

$$
\begin{aligned}
&\mathit{NextHour} \triangleq \\
&\quad \land \mathit{minute} = 59 \\
&\quad \land \mathit{hour}' = (\mathit{hour} + 1)\%24 \\
&\quad \land \mathit{minute}' = 0 \\
&\mathit{NextMinute} \triangleq \\
&\quad \land \mathit{minute} \neq 59 \\
&\quad \land \mathit{hour}' = \mathit{hour} \\
&\quad \land \mathit{minute}' = \mathit{minute} + 1 \\
&\mathit{Next} \triangleq \\
&\quad \lor \mathit{NextMinute} \\
&\quad \lor \mathit{NextHour}
\end{aligned}
$$

Here's a breakdown of what the spec does:

- *Next* can take *NextMinute* or *NextHour*
- *Next* takes *NextMinute* when *minute* is not 59, next hour is hour, next minute is minute + 1.

- *Next* takes *NextHour* when *minute* is 59, next hour is (hour + 1) modulus 24, next minute set to 0

Technically it's possible for *Next* to take both *NextMinute* and *NextHour*. This is not possible in this definition as *NextHour* and *NextMinute* are defined in a *mutually exlusively* fashion.

Finally, the spec itself is formally defined as:

$$
\begin{array}{l}
vars \triangleq \langle hour,\ minute \rangle \\
Spec \triangleq \\
\quad \land Init \\
\quad \land \Box[Next]_{vars}
\end{array}
$$

$\Box[Next]_{vars}$ deserves some special attention:

- *vars* is defined to be *all* variables in the spec. Different combination of these variables constitute the states of the system (eg. 23:59 and 00:00 are both states in the system).
- $\Box[Next]_{vars}$ is a *box-action formula*, where *Next* is an action and *vars* is a state function.
- $\Box$ operator asserts the formula is always true for every step in the behaviour.
- And steps in the behaviour is defined as $[Next]_{vars}$, where *Next* describe the action and *vars* capturing all variables representing the state.

2.4 Safety

Safety property describes invariant that must hold true in every state of system. A common invariant is *type safety* checks. In a digital clock, hour can only be in value between 0 to 23, and minute can only be value of 0 to 59:

$$
\begin{array}{l}
Type\_OK \triangleq \\
\quad \land hour \in 0 \,..\, 23 \\
\quad \land minute \in 0 \,..\, 59
\end{array}
$$

2.5 Liveness

Liveness property verifies certain behavioural across a sequence of state. One liveness property can be confirming the clock wraps around correctly at midnight (which involves mutliple states):

$$
Liveness \triangleq
$$

$$\land\ hour = 23 \land minute = 59 \leadsto hour = 0 \land minute = 0$$

$\leadsto$ is the *leads to* operator, suggesting something is eventually true. TLA+ provides a set of formulas that can be used to describe liveness property.

To verify liveness, we need to modify the spec slightly to enable *fairness* to prevent *stuttering*. In plain terms, fairness ensure *something* always happen in every step, allowing the states to transition. Without fairness the spec is allowed to *do nothing* as next step, this means liveness condition may fail because the spec permits the system to do nothing in perpetuity as next state. Fairness will be covered in more detailed in later chapter.

$$\begin{aligned}
&Spec \triangleq \\
&\quad \land\ Init \\
&\quad \land\ \Box[Next]_{vars} \\
&\quad \land\ \mathrm{WF}_{vars}(Next)
\end{aligned}$$

$WF_{vars}(Next)$ is the fairness qualifier.

2.6 Model Checker

The TLA+ spec can be verified using TLC model checker. The TLC model checker runs the spec and verifies all configured safety and liveness properties are satisfied during execution. To run TLC, we need two things:

- clock.tla - the spec itself
- clock.cfg - the corresponding configuration file

For reference, clock.tla spec is listed below:

MODULE *clock*

EXTENDS *Naturals*
VARIABLES *hour*, *minute*
$vars \triangleq \langle hour, minute \rangle$
$Type\_OK \triangleq$
$\quad \land\ hour \in 0 \,..\, 23$
$\quad \land\ minute \in 0 \,..\, 59$
$Liveness \triangleq$
$\quad \land\ hour = 23 \land minute = 59 \leadsto hour = 0 \land minute = 0$
$Init \triangleq$
$\quad \land\ hour = 0$
$\quad \land\ minute = 0$
$NextMinute \triangleq$
$\quad \land\ minute = 59$
$\quad \land\ hour' = (hour + 1)\%24$
$\quad \land\ minute' = 0$

$$
\begin{array}{l}
NextHour \triangleq \\
\quad \land\ minute \neq 59 \\
\quad \land\ hour' = hour \\
\quad \land\ minute' = minute + 1 \\
Next \triangleq \\
\quad \lor\ NextMinute \\
\quad \lor\ NextHour \\
Spec \triangleq \\
\quad \land\ Init \\
\quad \land\ \Box[Next]_{vars} \\
\quad \land\ \mathrm{WF}_{vars}(Next)
\end{array}
$$

The corresponding clock.cfg is listed below:

```
SPECIFICATION Spec
INVARIANTS Type_OK
PROPERTIES Liveness
```

Now run TLC and one should see something like this:

```
Model checking completed. No error has been found.
...
The depth of the complete state graph search is 1440.
```

Part I

Examples

Chapter 3

Blinking LED

Let's start with a trivial specification of a blinking LED. The intent of this example is to demonstrate the core functionalities of TLA+ specification language.

TODO: briefly talk about tla+ and model checker here.

3.1 Requirement

The LED is represented by a boolean variable that can be either 0 or 1.

... that's it.

3.2 Spec

The specification language may appear alienating as it is mathematically motivated based on propositional logic. Despite the (possibly) daunting syntax, designer only need to be familiar with a handful of key operators to start realizing value using TLA+. This chapter will attempt to describe the example in exhaustive detail to reduce the learning curve.

The following describe the core portion of the blinking LED spec.

MODULE *blinking*

VARIABLES b

$vars \triangleq \langle b \rangle$

$Init \triangleq$
$\quad \land b = 0$

$On \triangleq$
$\quad \land b = 0$
$\quad \land b' = 1$

$Off \triangleq$
$\quad \land b = 1$
$\quad \land b' = 0$

$$
\begin{array}{l}
Next \triangleq \\
\quad \lor Off \\
\quad \lor On \\
Spec \triangleq \\
\quad \land Init \\
\quad \land \Box[Next]_{vars}
\end{array}
$$

- $\triangleq$ is the *defines equal* operator
- $\land$ and $\lor$ are the AND and OR operator. The effect of these operator follow the natural definition in English:
  - $C \triangleq A \land B$: C is true iff A and B are true
  - $C \triangleq A \lor B$: C is true iff A or B is true
- The $'$ operator represents the next state. b' represent b's next state.
- *VARIABLES* keyword defines a list of variables for the spec. In this case the spec defines a variable b which can be either 0 or 1
- *vars* is typically defined as a shorthand to refer to *all* variables in the spec.

With the above definition, we can revisit the Action definitions: *Init* defines the initial system state, where b is set to 0.

Next requires more elaboration. TLA+ specifies the system as a collection of states with transitions between them. In a simplified sense, the state is described as a collection of ANDs (eg. system is in state C if both A and B are true), the ORs then describe the states the system can possibly be in (eg. system can be in state C OR D). Revisiting the example, the blinking LED has two states:

- $On \triangleq b = 0 \land b' = 1$: b switches on
- $Off \triangleq = 1 \land b' = 0$: b switches off

The system's *Next* state is defined to be one of these states:
$Next \triangleq On \lor Off$.

$\Box[Next]_{vars}$ is a **Box-Action Formula**, where *Next* is an action and *vars* is a state function. The formula is true iff every successive pair of steps in behaviour is a $[Next]_{vars}$. Finally *Spec* is conjunction between *Init* and $\Box[Next]_{vars}$. Note **all** TLA+ specification follows very similar template. There are situation we will need to provide *fairness* description - this will be covered later.

In short: this specification describes a two-state state machine where b toggles between 0 and 1.

Note that b can technically be *anything*. b can be 0, 1, -42, a dinosaur, etc. TLA+ specifies values of b which are valid in the system.

3.3 Safety

The spec so far only defines the possible states - but the *power* of TLA+ lies in its *properties* description. Safety properties are invariants that must hold true in *every* state. An invariant in the blinking LED example is:

$TypeOK \triangleq b \in \{0, 1\}$

This states the only valid value of b is 0 or 1. If b is ever set to anything else, the spec is invalid.

Some example safety properties include: Only a single thread have exclusive access to critical section, number of concurrent reads cannot exceed data available to be read, etc.

3.4 Liveness

While safety properties describe invariant that must be upheld in every state, *Liveness* describe properties of a sequence of states. In the blinking LED example, a liveness property can be the if b is 0, it eventually becomes 1, and vice versa. This is described below:

$$Liveness \triangleq \\ \quad \land b = 0 \leadsto b = 1 \\ \quad \land b = 1 \leadsto b = 0$$

It is the author's opinion liveness describes the *design essense* behind the spec. The key characteristic of a system is described by its *behaviour* across a series of states. Does a distribute algorithm eventually converge to a working state? Does a resource manager fairly allocate resources in all scenarios? Does a scheduler ensure all tasks are eventually scheduled? These are behaviours that are *cannot* be concluded by looking at a single state, but across a *sequence of state*. Liveness allows designer to express and verify these properties.

3.5 Model Checking

Since the blinking LED is trivially specified, the full specification is included below. For subsequent chapters only snippet will be included. Please refer to the accompanied material for full spec source.

TODO: install toolchain

TODO: commandline

TODO: using TLC

The following is the content of *blinking.tla*:

MODULE *blinking*

```
EXTENDS Naturals
VARIABLES b
vars ≜ ⟨b⟩
TypeOK ≜
  ∧ b ∈ {0, 1}
```

$$
\begin{array}{l}
\mathit{Liveness} \triangleq \\
\quad \land\ b = 0 \leadsto b = 1 \\
\quad \land\ b = 1 \leadsto b = 0 \\
\mathit{Init} \triangleq \\
\quad \land\ b = 0 \\
\mathit{Next} \triangleq \\
\quad \lor\ \land\ b = 0 \\
\qquad \land\ b' = 1 \\
\quad \lor\ \land\ b = 1 \\
\qquad \land\ b' = 0 \\
\mathit{Spec} \triangleq \\
\quad \land\ \mathit{Init} \\
\quad \land\ \Box[\mathit{Next}]_{\mathit{vars}} \\
\quad \land\ \mathrm{WF}_{\mathit{vars}}(\mathit{Next})
\end{array}
$$

The following is the content of *blinking.cfg*:

```
SPECIFICATION Spec
INVARIANTS TypeOK
PROPERTIES Liveness
```

3.6 Limitation

Since TLA+ exhaustively explores all possible state, a linear growth of variables leads to TLC (temporal logic checker) execution time grows *exponentially.*This means the specification must be scoped correctly to limit the state space.

Similarly, if you want to verify concurrent psuedo code implementation in PlusCal, you can likely at most verify 10s of lines of code.

Chapter 4

Simple Gossip Protocol

This section the author's notes on a simple gossip protocol by Andrew Hewler: https://ahelwer.ca/post/2023-11-01-tla-finite-monotonic/

4.1 Requirement

In a distributed system, a cluster of nodes collectively provide a serivce. A distributed database may have a collection of 10s to 100s of nodes working together to offer the service in a geo diverse fashion to be immnue to partial outage. The nodes often have requirements to know about each other. In the context of distributed database, a node may need to know the key range another of its peers. The cluster needs a way to communicate this information. One such mechanism is the gossip protocol.

Gossip protocols are used to communicate cluster information in a distributed fashion, (unsurprisingly) in a distributed system. Without gossip protocol, nodes in a cluster learns about its neighbours by contacting a centralized server. This introduces a single failure point in the system. As the name suggest, gossip protocol relies on nodes to gossip with each other. The nodes in the cluster periodically selects a set of neighbors to exchange what it knows about the cluster. The recency information is part of the gossip message itself, allowing the node and the peer its talking to quickly decide who has the latest information on a node, and converge to it. Assume a N node cluster and each interal a node selects k neighbours to gossip with, the total amount of gossip propagation time is described logrithmticly below:

$$propagation\_time = \log_k N * gossip\_interval \tag{4.1}$$

With the total number of messages exchanged:

$$messages\_exchanged = \log_k N * k \tag{4.2}$$

Now let's look at how a simple gossip protocol can be described by TLA+.

4.2 Spec

4.2.1 Base

In gossip protocol, every node needs to remember every other node's current state. In programming language this is typically described as counter[][]. The following is the equivalent in TLA+:

$Init \triangleq counter = [n \in Node \mapsto [o \in Node \mapsto 0]]$

This defines counter a collection of nodes, where each node also contains a collection of nodes initialized to 0.

The nodes can move to a new version:

$Increment(n) \triangleq counter' = [counter \text{ EXCEPT } ![n][n] = @ + 1]$

Note only the n's version is incremented. Communicating the update is done by the gossip action defined below:

```
Gossip(n, o) ≜
    LET Max(a, b) ≜ IF a > b THEN a ELSE b
    IN  counter' = [
          counter EXCEPT ![o] = [
              nn ∈ Node ↦
                  Max(counter[n][nn], counter[o][nn])
              ]
    ]
```

A few things to unpack here:

- n, o are the two nodes exchanging gossip. o is the node to be updated and n is the neighbor o gossips with.
- $LET..IN$ allows local definition under LET used under IN. In this case Max is a local macro defined to return maximum between a and b.
- $counter'$ (or refered to as counter *prime*) is what the variable will be in the next state. TLA+ doesn't provide a way to update a variable in a collection, so the convention is to assign a new array to the variable.
- $counter\ EXCEPT![o] = [...]$ return $counter$ with $counter[o]$ defined in the bracket.
- where [...] is a collection of nodes with with counter set to the max between the current node and neighbour.

Finally, the actual spec:

$$\begin{aligned} Next \triangleq\ & \lor \exists\, n \in Node : Increment(n) \\ & \lor \exists\, n,\, o \in Node : Gossip(n,\, o) \end{aligned}$$

Next supports two possible next steps describe using disjunctions. The first is bumping the version of a random node, the second is select a pair of nodes to gossip. Note the *existential qualifier* on both, which basically states there exists a node n in nodes, or there exists a pair of nodes n, o in nodes, respectively.

4.2.2 Finitized

There's a minor problem with the definition above. Gossip protocol, like many converging protocols, have a *monotonic increasing* requirement. On failures, the protocol bumps the version, which increases monotonically. Since TLA+ spec models the system as a graph, a monotonic increasing version number means the graph is *infinitely large*. To put the specification back into finite space, we can normalize the state:

$$
\begin{array}{l}
\mathit{GarbageCollect} \triangleq \\
\quad \text{LET } \mathit{SetMin}(s) \triangleq \text{CHOOSE } e \in s : \forall\, o \in s : e \leq o \text{IN} \\
\quad \text{LET } \mathit{Transpose} \triangleq \mathit{SetMin}(\{\mathit{counter}[n][o] : n,\ o \in \mathit{Node}\}) \text{IN} \\
\qquad \land\ \mathit{counter}' = [\\
\qquad\quad n \in \mathit{Node} \mapsto [\\
\qquad\qquad o \in \mathit{Node} \mapsto \mathit{counter}[n][o] - \mathit{Transpose} \\
\qquad\quad] \\
\qquad] \\
\qquad \land \text{UNCHANGED } \mathit{converge}
\end{array}
$$

SetMin(*s*) implements standard TLA+ semantics to retrieve the minimum element in the set. The definition can be read as *choose an e from S such that for every o in S, o is equal or bigger than e*. *Transpose* is then subsequently defined as the minimum value exist in counter. Finally, *counter′* is updated such that *every* elements substracts *Transpose*. The increment function is now updated to:

$$
\begin{array}{l}
\mathit{Increment}(n) \triangleq \\
\qquad \land\ \neg \mathit{converge} \\
\qquad \land\ \mathit{counter}[n][n] < \mathit{Divergence} \\
\qquad \land\ S\,!\,\mathit{Increment}(n) \\
\qquad \land \text{UNCHANGED } \mathit{converge}
\end{array}
$$

The conjunction $\mathit{counter}[n][n] < \mathit{Divergence}$ limits the maximum counter value. Finally, the Next action is updated to the follow:

$$
\begin{array}{l}
\mathit{Next} \triangleq \\
\qquad \lor\ \exists\, n \in \mathit{Node} : \mathit{Increment}(n) \\
\qquad \lor\ \exists\, n,\ o \in \mathit{Node} : \mathit{Gossip}(n,\ o) \\
\qquad \lor\ \mathit{Converge} \\
\qquad \lor\ \mathit{GarbageCollect}
\end{array}
$$

Note *GarbageCollect* is a now part of possible state transition. We will discuss *Converge* later, as it is related to liveness check. Lastly:

$Fairness \triangleq \forall\, n,\, o \in Node : \mathrm{WF}_{vars}(Gossip(n,\, o))$
$Spec \triangleq$
$\quad \wedge\, Init$
$\quad \wedge\, \Box[Next]_{vars}$
$\quad \wedge\, Fairness$

The *Fairness* formula ensures Gossip runs between every pair of n and o.

Chapter 5

Raft Consensus Protocol

Raft consensus protocol is a consensus algorithm that allows a cluster of independent nodes to work collectly to offer a service. One application of the raft consensus protocol is for database replication protocol. Assume replication factor of 3 and hard drive failure rate of 0.81% per year, the possibility of the total failure where the entire replication group goes down is $1 - 0.0081^3 = 99.9999\%$ uptime [3].

This section will provide a brief description of the protocol, enough to enable discussion of the TLA+ protocol. For a full description of the the Raft protocol, please refer to the sentinal paper [4].

The Raft protocol implements a few key tenent principles:

- A Raft cluster have N nodes, the cluster work collective as a *system* to offer some service
- Each node can be in one of three possible states: Follower, Candidate, Leader
- During normal operations, a cluster of N nodes have a single leader and N-1 followers.
- The leader handles all the client interactions. Requests sent to followers will be redirected to the leader.
- The leader regularly sends heartbeat to the follower, indicate its alive.
- The leader forwards all client requests to all of the followers. Once the majority nodes have processed the request, the request is now considered *committed to the system.*
- Once a request is *committed*, the protocol gaurantees the *system* will persist (possibly correct and re-replicate) all records across the cluster under *all* circumstance, including:

  - One or more server lost packet due to unfavourable network condition
  - One or more server crashed and recovered
  - A subset of the server were partitioned off for some period of time
  - ... etc

- If a follower cannot detect heartbeat from the leader, it will transition to become a candidate, vote for itself, and campaign for leader.
- A candidate collect the majority of the vote becomes the leader
- A newly elected leader will send a heartbeat to other nodes (irrespective to their state, leader, follower or candidate).

There are a lot of details omitted here, such as:

- How a node determine if it should vote for candidate
- How a node knows it's leadership status expired
- How a newly elected leader syncs its logs with the follower
- ... and more

Note all N nodes in the cluster operate *independently* following the above hueristics. Hopefully this highlights the complexity around verifying the the correctness of the protocol.

5.1 Spec

The Raft protocol inventor open sourced a full TLA+ spec for the protocol [5]. Since the focus of the book is learning TLA+, we will implement only a portion of the spec, namely the leader election. As usual, let's look at the init definition:

MODULE *raft*

$$
\begin{array}{ll}
InitHistoryVars \triangleq & \land\ elections = \{\} \\
& \land\ allLogs\ \ = \{\} \\
& \land\ voterLog = [i \in Server \mapsto [j \in \{\} \mapsto \langle\rangle]] \\
InitServerVars \triangleq & \land\ currentTerm = [i \in Server \mapsto 1] \\
& \land\ state\ \ \ \ \ \ \ = [i \in Server \mapsto Follower] \\
& \land\ votedFor\ \ \ \ = [i \in Server \mapsto Nil] \\
InitCandidateVars \triangleq & \land\ votesResponded = [i \in Server \mapsto \{\}] \\
& \land\ votesGranted\ \ = [i \in Server \mapsto \{\}]
\end{array}
$$

The values nextIndex[i][i] and matchIndex[i][i] are never read, since the leader does not send itself messages. It's still easier to include these in the functions.

$$
\begin{array}{ll}
InitLeaderVars \triangleq & \land\ nextIndex\ \ = [i \in Server \mapsto [j \in Server \mapsto 1]] \\
& \land\ matchIndex = [i \in Server \mapsto [j \in Server \mapsto 0]]
\end{array}
$$

$$
\begin{array}{lll}
InitLogVars & \triangleq & \land\ log \quad\quad\quad\quad\ = [i \in Server \mapsto \langle\rangle] \\
 & & \land\ commitIndex\ = [i \in Server \mapsto 0] \\
Init & \triangleq & \land\ messages = [m \in \{\} \mapsto 0] \\
 & & \land\ InitHistoryVars \\
 & & \land\ InitServerVars \\
 & & \land\ InitCandidateVars \\
 & & \land\ InitLeaderVars \\
 & & \land\ InitLogVars
\end{array}
$$

- Servers are defined as a set in this protocol, so *function mapping* are used to represent various server state information
- *log* is set to a *function mapping* for all the servers, and each server is initialized as an ordered tuple
- *commitIndex* is initialized to empty ordered tuple for every server
- *nextIndex* and *matchIndex* are initialized to a *function mappping of function mapping of integers*, to represent every server keeps a counter for all other servers.
- *voteResponded* and *voteGranted* are represented as a set for every server
- *currentTerm*, *state*, and *votedFor* are represented as a mapping function for every server.
- *message* is defined to be an empty set, but each key in the set has an associated count value (similar to C++ standard map with value tracking the key count).

The following is the state transition definition:

MODULE *raft*

$$
\begin{array}{lll}
Next & \triangleq & \land \lor \exists\, i \in Server : Restart(i) \\
 & & \quad \lor \exists\, i \in Server : Timeout(i) \\
 & & \quad \lor \exists\, i, j \in Server : RequestVote(i, j) \\
 & & \quad \lor \exists\, i \in Server : BecomeLeader(i) \\
 & & \quad \lor \exists\, i \in Server, v \in Value : ClientRequest(i, v) \\
 & & \quad \lor \exists\, i \in Server : AdvanceCommitIndex(i) \\
 & & \quad \lor \exists\, i, j \in Server : AppendEntries(i, j) \\
 & & \quad \lor \exists\, m \in \text{DOMAIN } messages : Receive(m) \\
 & & \quad \lor \exists\, m \in \text{DOMAIN } messages : DuplicateMessage(m) \\
 & & \quad \lor \exists\, m \in \text{DOMAIN } messages : DropMessage(m)
\end{array}
$$

The spec uses *there exists* qualifier to pick a server to perform an action. The possible actions are concatenated with $\lor$ to indicate any of them can take. Some of these actions are state specific (eg. *RequestVote* or *BecomeLeader*).

The state check are applied within the macro definition itself (eg. *RequestVote* checks server is in *Candidate* state). Let's take a closer look at *RequestVote*:

MODULE *raft*

```
Helper for Send and Reply. Given a message m and bag of messages, return a
new bag of messages with one more m in it.
WithMessage(m, msgs) ≜
    IF m ∈ DOMAIN msgs THEN
        [msgs EXCEPT ![m] = msgs[m] + 1]
     ELSE
        msgs @@ (m :> 1)

 Add a message to the bag of messages.
Send(m) ≜ messages' = WithMessage(m, messages)

RequestVote(i, j) ≜
    ∧ state[i] = Candidate
    ∧ j ∉ votesResponded[i]
    ∧ Send([mtype          ↦ RequestVoteRequest,
            mterm          ↦ currentTerm[i],
            mlastLogTerm   ↦ LastTerm(log[i]),
            mlastLogIndex  ↦ Len(log[i]),
            msource        ↦ i,
            mdest          ↦ j])
    ∧ UNCHANGED ⟨serverVars, candidateVars, leaderVars, logVars⟩
```

- i and j are reciever and sender, respectively
- Send request only if receiver is a candidate
- Send request only if j isn't part of i's votesResponded set
- A *message* is defined as a *function* with a collection of keys and values.
- *Send* calls *WithMessage*
- *WithMessage* increments the count (value) if the message (key) already exists
- *WithMessage* insert the message (key) with count of 1 (value)
  - @@ combines two functions
  - $m :> 1$ creates a function with message and a count of 1

In short, the *RequestVote* function inserts a message into a ddictionary and track the number of times it has been inserted. Let's take a look at how the message is processed:

MODULE *raft*

```
WithoutMessage(m, msgs) ≜
    IF m ∈ DOMAIN msgs THEN
        IF msgs[m] ≤ 1 THEN [i ∈ DOMAIN msgs \ {m} ↦ msgs[i]]
         ELSE  [msgs EXCEPT ![m] = msgs[m] − 1]
     ELSE
        msgs

Reply(response, request) ≜
    messages′ = WithoutMessage(request, WithMessage(response, messages))

HandleRequestVoteRequest(i, j, m) ≜
    LET logOk ≜  ∨ m.mlastLogTerm > LastTerm(log[i])
                 ∨ ∧ m.mlastLogTerm = LastTerm(log[i])
                   ∧ m.mlastLogIndex ≥ Len(log[i])
         grant ≜  ∧ m.mterm = currentTerm[i]
                  ∧ logOk
                  ∧ votedFor[i] ∈ {Nil, j}
    IN   ∧ m.mterm ≤ currentTerm[i]
         ∧ ∨ grant  ∧ votedFor′ = [votedFor EXCEPT ![i] = j]
           ∨ ¬grant ∧ UNCHANGED votedFor
         ∧ Reply([mtype          ↦ RequestVoteResponse,
                  mterm          ↦ currentTerm[i],
                  mvoteGranted ↦ grant,
                   mlog is used just for the 'elections' history variable for
                   the proof. It would not exist in a real implementation.
                  mlog           ↦ log[i],
                  msource        ↦ i,
                  mdest          ↦ j],
                  m)
         ∧ UNCHANGED ⟨state, currentTerm, candidateVars, leaderVars, logVars⟩

UpdateTerm(i, j, m) ≜
    ∧ m.mterm > currentTerm[i]
    ∧ currentTerm′  = [currentTerm EXCEPT ![i] = m.mterm]
    ∧ state′        = [state       EXCEPT ![i] = Follower]
    ∧ votedFor′     = [votedFor    EXCEPT ![i] = Nil]
       messages is unchanged so m can be processed further.
    ∧ UNCHANGED ⟨messages, candidateVars, leaderVars, logVars⟩

Receive(m) ≜
    LET i ≜ m.mdest
        j ≜ m.msource
    IN   Any RPC with a newer term causes the recipient to advance
         its term first. Responses with stale terms are ignored.
         ∨ UpdateTerm(i, j, m)
         ∨ ∧ m.mtype = RequestVoteRequest
           ∧ HandleRequestVoteRequest(i, j, m)
```

$$
\begin{array}{l}
\lor\ \land\ m.mtype = RequestVoteResponse \\
\quad \land\ \lor\ DropStaleResponse(i,\ j,\ m) \\
\qquad \lor\ HandleRequestVoteResponse(i,\ j,\ m) \\
\lor\ \land\ m.mtype = AppendEntriesRequest \\
\quad \land\ HandleAppendEntriesRequest(i,\ j,\ m) \\
\lor\ \land\ m.mtype = AppendEntriesResponse \\
\quad \land\ \lor\ DropStaleResponse(i,\ j,\ m) \\
\qquad \lor\ HandleAppendEntriesResponse(i,\ j,\ m)
\end{array}
$$

MODULE *raft*

- *Next* randomly picks a message received. This simulates network reordering effect
- Per Raft protocol definition: *UpdateTerm* checks if *message* has higher term. If so, the current server transitions back to *Follower* with grantVote cleared.
- *HandleRequestVoteRequest* defines two local variables:
  - *logOk* represents requester log's recency per protocol spec
  - *grant* represents if vote is granted per protocol spec. Note it's possible
- *Reply* adds the replay to messages and removes the request
- In *WithoutMessage*, return a new function if message count is one:
  - $msgs \backslash \{m\}$ is all msgs excluding m
  - For all i in this reduced msgs functions, set value to key i as msgs[i]. This effectively copies the entire msgs excluding key m

If receiver is stale (eg. lower term), it will **not** grant the vote. This wasn't explicitly stated in the protocol spec. At the same time, the receiver should eventually call *UpdateTerm* to get updated term.

Chapter 6

Simple Scheduler

6.1 Requirement

In this section we will define a spec for a simple task scheduler. The task sechdeuler has the following requirements:

- Supporting N execution context (ie. CPUs)
- Supporting T number of tasks
- Tasks have identical priority and are scheduled coopertively
- System has a single global lock
- Any task can attempt to acquire the lock, Any task attempting to acquire the lock are gauranteed to be scheduled.
- If multiple tasks attempt to grab the lock, the tasks will be scheduled in lock request order.

6.2 Spec

We will model scheduler using the following variables:

$Init \triangleq$
$\quad \wedge\ cpus = [i \in 0\,..\,N-1 \mapsto \text{“”}]$
$\quad \wedge\ ready\_q = S2T(Tasks)$
$\quad \wedge\ blocked\_q = \langle\rangle$
$\quad \wedge\ lock\_owner = \text{“”}$

A few things to note:

- The system has N executing context, represented as number of CPUs. When a task is running, $cpus[k]$ is set to *taskName*. When CPU is idle, $cpus[k]$ is set to an empty string.
- *ready_q* and *blocked_q* are initialized as *ordered tuple*, due to the cooperative scheduling requirement.
- $S2T$ is a macro that converts a set into a ordered tuple. This is to accommodate the fact it appears I cannot define tuple in .cfg file.
- Finally, the single system lock is represented as *lock_owner*.

A task can be in three possible state: Ready, Blocked and Running. The *Next* box-action fomula will define a Ready and Running action, and the implementation will include related lock contention handling.

MODULE *scheduler*

$$
\begin{array}{l}
MoveToReady(k) \triangleq \\
\quad \land\ cpus[k] \neq \text{“”} \\
\quad \land\ lock\_owner \neq cpus[k] \\
\quad \land\ ready\_q' = Append(ready\_q,\ cpus[k]) \\
\quad \land\ cpus' = [cpus \text{ EXCEPT } ![k] = \text{“”}] \\
\quad \land \text{ UNCHANGED } \langle lock\_owner,\ blocked\_q \rangle \\
Lock(k) \triangleq \\
\quad \text{lock is empty} \\
\quad \lor\ \land\ cpus[k] \neq \text{“”} \\
\quad\quad \land\ lock\_owner = \text{“”} \\
\quad\quad \land\ lock\_owner' = cpus[k] \\
\quad\quad \land \text{ UNCHANGED } \langle ready\_q,\ cpus,\ blocked\_q \rangle \\
\quad \text{someone else has the lock} \\
\quad \lor\ \land\ cpus[k] \neq \text{“”} \\
\quad\quad \land\ lock\_owner \neq \text{“”} \\
\quad\quad \land\ lock\_owner \neq cpus[k] \quad \text{cannot double lock} \\
\quad\quad \land\ blocked\_q' \ = Append(blocked\_q,\ cpus[k]) \\
\quad\quad \land\ cpus' = [cpus \text{ EXCEPT } ![k] = \text{“”}] \\
\quad\quad \land \text{ UNCHANGED } \langle ready\_q,\ lock\_owner \rangle \\
Unlock(k) \triangleq \\
\quad \land\ cpus[k] \neq \text{“”} \\
\quad \land\ lock\_owner = cpus[k] \\
\quad \land\ lock\_owner' = \text{“”} \\
\quad \land\ cpus' = [cpus \text{ EXCEPT } ![k] = \text{“”}] \\
\quad \land\ ready\_q' = ready\_q \circ blocked\_q \circ \langle cpus[k] \rangle \\
\quad \land\ blocked\_q' = \langle \rangle \\
\\
Running \triangleq \\
\quad \exists\, k \in \text{DOMAIN } cpus : \\
\quad\quad \land\ cpus[k] \neq \text{“”}
\end{array}
$$

$\land \lor MoveToReady(k)$
$\quad \lor Lock(k)$
$\quad \lor Unlock(k)$

6.3 Safety

6.4 Liveness

I believe this is the most important part of cooperative scheduler design. While the scheduler can't *force* a task to relinquish a lock (the scheduler doesn't dictate when the task is *done*), the scheduler can ensure scheduling fairness by scheduling the next lock requester intsead of the task that just relinquished the lock.

MODULE *scheduler*

$Liveness \triangleq$
$\quad \forall\, t \in Tasks :$
$\quad\quad$ LET
$\quad\quad\quad b \triangleq \{x \in \text{DOMAIN } blocked\_q : blocked\_q[x] = t\}$
$\quad\quad$ IN
$\quad\quad\quad \land\, b \neq \{\} \leadsto b = \{\}$

The formula defines set b to be either an empty set or a set of one task. Assume a set of {"p0", "p1", "p2}. Possible value of b include: {}, {"p0"}, {"p1"} and {"p2"}. The fomula then states an non empty set of b leads to an *empty set* of b. In other words:

If a task ever becomes blocked, it will eventually become unblocked.

However, when we actually run the model checker, we will find the liveness property is *violated*. The failure scenario is basically one task holding onto the lock in one CPU, while the scheduler repeatedly schedule/deschedule a separate task in another CPU. While this is perfectly allowed, the model checker detects a possible path for the the system to trap in a local state and fail the liveness property.

Perhaps not surprisingly, if you construct similar liveness property to verify a task is *eventually* always scheduled, it will also fail. The model checker will provide a counter case where a task is never scheduled because another task is repeatedly acquire/release the global lock.

We need *Strong Fairness* to solve this problem:

MODULE *scheduler*

$$
\begin{array}{l}
L \triangleq \\
\quad \forall\, t \in \mathit{Tasks} : \\
\qquad \forall\, n \in 0 \,..\, (N-1) : \\
\qquad\quad \mathrm{WF}_{\mathit{vars}}(\mathit{HoldingLock}(t) \land \mathit{Unlock}(n)) \\
\mathit{Spec} \triangleq \\
\quad \land\, \mathit{Init} \\
\quad \land\, \Box[\mathit{Next}]_{\mathit{vars}} \\
\quad \land\, \mathrm{WF}_{\mathit{vars}}(\mathit{Next}) \\
\quad \land\, L
\end{array}
$$

Fairness ensures that we are never stuck in a repeated state.

Chapter 7

Simple Elevator

https://surfingcomplexity.blog/2024/10/16/a-liveness-example-in-tla/

Chapter 8

Miscellaneous

Part II

Examples with PlusCal

Chapter 9

SPSC Lockfree Queue

Single producer single consumer (SPSC) *Lockfree* queue is a standard data exchange queue between a producer and a consumer. The SPSC lockfree queue promises data can exchange between producer and consumer in a *lockfree* fashion, suggesting all condition both producer and consumer can make progress.

Contrast to standard shared queues, a SPSC waitfree queue doesn't require the use of a *lock* (eg. mutex). The queue can be logically represented fairly simply as:

```
template <typename T, ssize_t N>
class cQueue<T> {
    ssize_t rptr = 0;
    ssize_t wptr = 0;
    std::array<T, N> buffer;
    /* TODO: API definition below... */
};
```

A real implementation need to account for memory ordering effects specific to the architecture. For example, ARM has weak memory ordering model where read/write may appear out of order between CPUs. In this chapter we will only assume *logical* execution where each command is issued sequentially (even perceived across CPUs) to focus the discussion on TLA+.

9.1 Requirement

As mentioned in earlier section, a SPSC queue is represented by an array, a pair of read write pointer. The implementation is (hopefully) descriptively trivial:

- Two executing context, reader and writer
- Writer advances wtpr after writes
- Reader advances rtpr after reads

- If rtpr equals wptr, queue is empty
- If (wtpr + 1) % N equals rptr, queue is full

A possible implementation may look like below (not accounting for memory ordering effects):

```
template <typename T, ssize_t N>
class cQueue {
    ssize_t rptr = 0;
    ssize_t wptr = 0;
    std::array<T, N> buffer;

public:
    bool read(T &v) {
        /* queue empty check */
        if (rptr == wptr) {
            return false;
        }
        /* data get */
        v = buffer[rptr];
        /* rtpr update */
        rptr = (rptr + 1) % N;
        return true;
    }

    bool write(const T &v) {
        /* queue full check */
        if ((wptr + 1) % N == rptr) {
            return false;
        }
        /* data write */
        buffer[wptr] = v;
        /* wptr update */
        wptr = (wptr + 1) % N;
        return true;
    }
};
```

Since reader and writer execute in different context, the instructions in read and write can interleave in *any* way imaginable:

- queue empty check can happen before or after queue full check
- data write happens immediately before data read
- ... so on and so forth

The key observations is that buffer[wtpr] is reserved by the producer. buffer[wtpr] is either unused or being written to. In either case the reader is not allowed to access it. Symmetric reasoning applies to rptr. This provides the *safety* to the design - but how do we verify this?

This is where TLA+ can help us formally verify the design.

9.2 Spec

TLA+ specification can be writen using its native formal specification language, or a C-like syntax called PlusCal (which transpiles down to itse native form). In this example, I chose to implement the specification using PlusCal, since the content to be verified is psuedo implementation. While it is possible specify SPSC in native TLA+, it is the author's opinion that it is more error prone in this case, each line is effective an individual state needs to be modeled.

The following is a snippet of the specification written in PlusCal, hopefully intuitive to read:

```
procedure reader(i)
variable
begin
r_chk_empty:      if rptr = wptr then
r_early_ret:          return;
                  end if ;
r_read_buf:       assert buffer[rptr] ≠ 0;
r_cs:             buffer[rptr] := 0;
r_upd_rtpr:       rptr := (rptr + 1)%N;
                  return;
end procedure ;

procedure writer(i)begin
w_chk_full:       if (wptr + 1)%N = rptr then
w_early_ret:          return;
                  end if ;
w_write_buf:      assert buffer[wptr] = 0;
w_cs:             buffer[wptr] := wptr + 1000;
w_upd_wptr:       wptr := (wptr + 1)%N;
                  return;
end procedure ;
```

Note each command starts with a *label*, such as r_chk_empty. All the actions associated with the label is assumed executed atomically. This is reflected in the generated TLA+ code:

```
r_chk_empty(self) ≜ ∧ pc[self] = "r_chk_empty"
                    ∧ IF rptr = wptr
                        THEN ∧ pc' = [pc EXCEPT ![self] = "r_early_ret"]
                        ELSE ∧ pc' = [pc EXCEPT ![self] = "r_read_buf"]
                    ∧ UNCHANGED ⟨rptr, wptr, buffer, stack, i_, i⟩
```

9.3 Safety

As mentioned before, safety properties need to hold true in every single state. Some safety requirement we can enforce, for example:

Reader and writer cannot access the same index at the same time:

$$\sim ((pc[100] = "w\_cs") \wedge (pc[101] = "r\_cs") \wedge rptr = wptr) \tag{9.1}$$

All unused index should be set to 0:

$$\forall kk \in unused : buffer[kk] = 0 \tag{9.2}$$

At any given moment, buffer[wtpr] may be unused or written. buffer[rptr] may be unused or read:

$\vee\ Cardinality(to\_be\_read) + 1 = Cardinality(reading)$
$\vee\ Cardinality(to\_be\_read) \quad = Cardinality(reading) + 1$
$\vee\ Cardinality(to\_be\_read) \quad = Cardinality(reading)$

9.4 Liveness

All indicies are eventually used:

$Liveness \triangleq$
$\forall\, k \in 0 \,..\, N-1 :$
$\Diamond(buffer[k] \neq 0)$

Unused index 0 becomes used, used index 0 becomes unused.

$Liveness2 \triangleq$
$\wedge\ (buffer[0] = 0) \rightsquigarrow buffer[0] = 1000$
$\wedge\ (buffer[0] = 1000) \rightsquigarrow buffer[0] = 0$

9.5 Configuration

Chapter 10

SPMC Lockless Queue

Part III

Language Reference

Chapter 11

Data Structure

Like other languages, TLA+ provides its data structure. I assume the readers are already familiar with common data structure, and this chapter will only focus on the TLA+ language semantics.

11.1 Set

This is the most common data structure used in TLA+ spec. The following is a few examples on how a set can be used:

$a \triangleq \{0, 1, 2\}$		
$b \triangleq \{2, 3, 4\}$		
$c \triangleq a \cup b$		{0, 1, 2, 3, 4}
$d \triangleq a \cap b$		{2}
$e \triangleq \exists x \in c : x > 3$		TRUE - because 4 in c is bigger than 3
$f \triangleq \exists x \in c : x > 5$		FALSE - nothing in c is bigger than 5
$g \triangleq \forall x \in c : x < 3$		FALSE - not all elements in c are smaller than 3
$h \triangleq \forall x \in c : x < 5$		TRUE - all elements in c are smaller than 3
$i \triangleq \{x \in c : x < 3\}$		{0, 1, 2} - all elementse less than 3
$j \triangleq Cardinality(c)$		5 - the number of elements in c
$k \triangleq c \setminus d$		{0, 1, 3, 4} - c substracts d

11.2 Tuple

$A \triangleq \langle 0, 1, 2 \rangle$		
$B \triangleq \langle 2, 3, 4 \rangle$		
$C \triangleq A \circ B$		tuple: 0, 1, 2, 2, 3, 4
$D \triangleq Len(C)$		6
$E \triangleq \forall x \in 1 \,..\, Len(C) : C[x] \neq 10$		TRUE - every C[x] is not 10
		First tuple element is at index 1 (not 0)
$F \triangleq \exists x \in 1 \,..\, Len(C) : C[x] = 2$		TRUE - there exists a C[x] that is 2

$G \triangleq \{x \in 1 \,..\, Len(C) : C[x] = 2\}$ {3, 4} - when index is 3 or 4, C[x] = 2

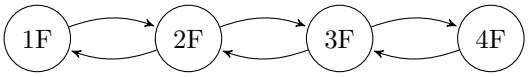
Chapter 12

Idiom

Chapter 13

Fairness and Liveness

For rigorous definition and proof, please refer to (TODO: citations). This chapter focus on the application aspect of liveness and fairness and define an elevator spec that goes up and down.



13.1 Liveness

Consider the following elevator *Spec*:

MODULE *elevator*

EXTENDS *Integers*
VARIABLES a
$vars \triangleq \langle a \rangle$
$TOP \triangleq 4$
$BOTTOM \triangleq 1$
$Init \triangleq$
$\quad \wedge a = BOTTOM$
$Up \triangleq$
$\quad \wedge a \neq TOP$
$\quad \wedge a' = a + 1$
$Down \triangleq$
$\quad \wedge a \neq BOTTOM$
$\quad \wedge a' = a - 1$
$Spec \triangleq$
$\quad \wedge Init$
$\quad \wedge \Box[Up \vee Down]_a$

The building has a set of floors and the elevator can go either up or down. The elevator keeps going up until it's the top floor, or keep going down until

it's the bottom floor. TLC will pass the *Spec* as is.

Let's introduce a liveness property. The elevator should always at least go to the second floor:

$Liveness \triangleq$
$\quad \wedge\ a = 1 \leadsto a = 2$

Running the *Spec* against TLC will report a violation:

```
Error: Temporal properties were violated.
Error: The following behavior constitutes a counter-example:
State 1: <Initial predicate>
a = 1
State 2: Stuttering
```

Since the *Spec* permits *suttering*, the state machine is allowed to perpetually stay on 1F and *never* go to 2F. This can be fixed by introduce fairness description.

13.2 Weak Fairness

Weak fairness is defined as:

$$\Diamond\Box(ENABLED\langle A\rangle_v) \Rightarrow \Box\Diamond\langle A\rangle_v \tag{13.1}$$

$ENABLED\langle A\rangle$ represents *conditions required* for action A. The above translates to: if conditions required for action A to occur is *eventually always* true, then action A will *always eventually* happen.

Without weak fairness defined, the elevator may *stutter* at floor 1 and never go to floor 2. Weak fairness states that if the conditions of an action is *eventually always* true (ie. elevator decides to stay on 1F but but *can* go up), the elevator *always eventually* go up.

$Spec \triangleq$
$\quad \wedge\ Init$
$\quad \wedge\ \Box[Down \vee Up]_a$
$\quad \wedge\ \mathrm{WF}_a(Down)$
$\quad \wedge\ \mathrm{WF}_a(Up)$

Running the spec against TLC passes again. What if we want to verify the elevator eventually always goes to the top, not just to 2F? Let's modify the Liveness property again:

$Liveness \triangleq$

$\land\ a = BOTTOM \leadsto a = TOP$

TLC now reports the following violation:

```
Error: Temporal properties were violated.
Error: The following behavior constitutes a counter-example:
State 1: <Initial predicate>
a = 1
State 2: <Up line 10, col 5 to line 11, col 17 of module elevator>
a = 2
Back to state 1: <Down line 13, col 5 to line 14, col 17 of module elevator>
```

TLC identified a case where the elevator is perpetually stuck going between 1F and 2F, but never go to 3F. Weak fairness is no longer enough, because the the elevator is not stuck on 2F repeatedly, but stuck going between 1F and 2F. This is where we need strong fairness.

13.3 Strong Fairness

Strong fairness is defined as:

$$\Box\Diamond(ENABLED\langle A\rangle_v) \Rightarrow \Box\Diamond\langle A\rangle_v \tag{13.2}$$

The difference between weak and strong fairness is the *eventually always* vs. *always eventually*.

In weak fairness, once the state machine is stuck in a state forever, the state machine always transition to a possible next state permitted by the spec (eg. if the elevator is stuck on 1F but can go to 2F, it will). With strong fairness, the elevator doesn't need to be stuck on 2F to go to 3F. If the elevator *always eventually* makes it to 2F, it *eventually always* go to 3F.

Intuitively we are tempted to enable strong fairness like so:

$Spec \triangleq$
$\land\ Init$
$\land\ \Box[Up \lor Down]_a$
$\land\ \mathrm{WF}_a(Down)$
$\land\ \mathrm{SF}_a(UP)$

However, TLC *still* reports the same violation. What's going on?

If we take a closer look at the enabling condition for *Up*, it only requires current floor to be not TOP. Up is *always eventually* called even if elevator goes between 1F and 2F indefinitely (because 1F is not TOP). What we really want is strong fairness on *Up* for *every floor*. So if elevator makes to 2F once, it will

eventaully go to 3F. If elevator makes to 3F once, it will eventaully go to 4F, etc. The following is the change required:

$$
\begin{array}{l}
Spec \triangleq \\
\quad \land\ Init \\
\quad \land\ \Box[Up \lor Down]_a \\
\quad \land\ \mathrm{WF}_a(Down) \\
\quad \land\ \forall f \in BOTTOM \mathinner{..} TOP - 1 : \\
\qquad \land\ \mathrm{WF}_a(Up \land f = a)
\end{array}
$$

Once again with this change TLC will pass.

Chapter 14

Reference

Bibliography

[1] Srikumar Subramanian *https://sriku.org/posts/fairness-in-tlaplus/*, 2015

[2] Richard M. Murray, Nok Wongpiromsarn *Linear Temporal Logic, Lecture 3*, 2012

[3] *https://www.backblaze.com/blog/cloud-storage-durability/*

[4] *https://raft.github.io/raft.pdf*

[5] *https://github.com/ongardie/raft.tla*