More TLA+ Examples

Richard Tang

January 15, 2025

Contents

Chapter 1

Introduction

TODO: personal industry experience

TODO: TLA+ was developed ahead of its time, and software complexity finally caught up

TODO: concurrent programming use case

TODO: distributed system use case

TODO: exhaustively check the entire space

With diminishing return on vertical scaling, the industry now invest heavily towards horizontal scaling. This shifts complexity from hardware to software with solution such as distributed algorithms. However, distributed algorithms are hard to get right. The system is now a cluster of independently operating entities and need to somehow collectively offer the correct system behaviour. To make matter worse, human cognition is inherently singled threaded. We are not good at reasoning about parallel execution. The usual anti-pattern is to keep bandaiding the solution until bug reports stop comming in - but how does anyone know if the solution is actually *correct by design*? To solve this problem, we then must rely on tools to do the reasoning for us, entering 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).

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.

2.2 Digital Clock

TLA+ specifies the system using *propositional logic*. In this example, we will specify a digital clock that has hour and minute. The clock increments one minute at a time, and wraps around at midnight (ie. 23:59 transitions to 00:00). 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:

$vars \triangleq \langle hour, minute \rangle$
$Spec \triangleq$
$\quad \wedge Init$
$\quad \wedge \Box[Next]_{vars}$

$\Box[Next]_{vars}$ 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.

2.2.1 Safety

Safety property describes invariant that must hold true in every state of system. A common invariant is *type safety* checks, defined below:

2.2.2 Liveness

2.3 TLC Model Checker

The TLA+ can be verified using TLC model checker.

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

$vars \triangleq \langle b \rangle$

$TypeOK \triangleq$

$\quad \land b \in \{0, 1\}$

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

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

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+.

4.1 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.

4.2 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⟩
```

4.3 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:

$$\sim ((pc[100] = "w\_cs") \wedge (pc[101] = "r\_cs") \wedge rptr = wptr) \tag{4.1}$$

All unused index should be set to 0:

$$\forall kk \in unused : buffer[kk] = 0 \tag{4.2}$$

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

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

4.4 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.

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

4.5 Configuration

Chapter 5

SPMC Lockless Queue

Chapter 6

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/

6.1 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:

$$propagation\_time = \log_k N * gossip\_interval \tag{6.1}$$

With the total number of messages exchanged:

$$messages\_exchanged = \log_k N * k \tag{6.2}$$

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

6.2 Spec

6.2.1 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:

$$Next \triangleq \lor \exists\, n \in Node : Increment(n) \\ \lor \exists\, n,\, o \in Node : Gossip(n,\, o)$$

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.

Finally, the actual spec definition:

$$\begin{aligned} Spec \triangleq\ & \land Init \\ & \land \Box[Next]_{counter} \end{aligned}$$

The second conjunction formula is a **Box-Action Formula**, where *Next* is an action and *counter* is a state function. The formula is true iff every successive pair of steps in behaviour is a $[Next]_{counter}$. The spec defines a temporal theorem that is *always true*.

6.2.2 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:

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

$SetMin(s)$ 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:

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

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

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

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

$Fairness \triangleq \forall\, n,\, o \in Node : \mathrm{WF}_{vars}(Gossip(n,\, o))$

$Spec \triangleq$

$\wedge\, Init$

$\wedge\, \Box[Next]_{vars}$

$\wedge\, Fairness$

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

Chapter 7

Raft Consensus Protocol

Chapter 8

OS Scheduler

Chapter 9

Miscellaneous