A Note on Formal Specification for Programmers

2020-11-20

Abstract

Broadly speaking there are three levels of programming complexity: single-threaded applications, concurrent processes, and distributed systems. The latter category is complicated by unreliable messaging which further complicates composing individual thread state into a coherent system to achieve resilient availability. Those aims frustrate the scaling potential that machine clusters might otherwise deliver. Formal methods in specification may help alleviate these challenges by proving a design satisfies its functional and non-functional requirements before labor resources are expended to possibly uncertain ends. However, the predominately imperative orientation of programmers may actually serve to make specification harder. In this first technical note I examine selected issues here working through a simplistic KV (key-value) store with emphasis on state and state reachability in `TLA+`, a formal system with good acceptance in industry. Please note this work is based on several important clarifications provided by Stephan Merz, and Calvin Loncaric in the Google TLA usenet group.

1 Introduction

A software boss calls an impromptu meeting where he explains that, because sales is onboarding a new strategic customer, the team will be redirected to complete an estimated six quarters of work over fifteen Agile epics or feature sets. While this is real work, the team is not unduly stressed. After all their business lead will deliver a written description of the epics, and help decompose that work into multiple Agile stories for execution over two-week sprints. Being well versed in Agile processes, the programming team is quite able to turn those plans into code. The programmers also know that, as time elapses, some features may be discarded while others may grow more complicated, or require a redo as the team's collective judgement warrants. Agile is about risk reduction, and better team communication. It cannot eliminate all risk.

Now consider another software boss across the street in a similar meeting. Except that in this case, the boss says he will require a formal model be co-developed with the code. In some circles this may be met with appreciation. But in other circles this causes serious concern. Formal methods is a whole other work discipline that doesn't automatically fall-out of C/C++/GO know-how. And besides will it really help? The programmers here know what the guys across the street know: features change in scope, size, and shape which carries over to code. That translation isn't always straightforward even in single threaded cases. Industrial programming uses third party libraries which are not formally verified and probably can't be without obliterating available resources. Even more worrying is that the specification may be wrong, or fail to be carried intact into code undoing indeed even mocking the formal work.

So, what's wisdom here? Formal methods are like Agile but in another subject matter arena: specification. Formal methods ask us to name then define safety and liveness properties, which are important sub-categories of functional behavior. Safety properties say what the system must do. Formal methods try to find counter-examples which violate safety. Liveness concerns itself with proving or disproving through counter-examples that the system is

not deadlocked, and that threads are making progress towards their post conditions.

The appropriateness of formal methods turns then on a judgement of the overall complexity of the system with single threaded and distributed systems at opposite ends. Formal modeling is indicated when the architecture of concurrent communicating agents or systems (CCS) is unclear. With formal modeling, abstract potential solutions can be attempted on paper before the company spends real dollars in time and labor implementing it.

Formal methods come into two modalities with two writing tools. In the model checking variety, the formal toolset will start from a defined initial state then systematically explore all reachable states. At each step it will test the safety, liveness conditions for violation. `TLA+, SPIN` are two well known systems here. Depending on state size, the tool may find all possible states, or explore a subset of them limited by memory or the time it's allowed to work. The tool will quantify the states explored in graph terms: total nodes found, edges per mode, and depth in the graph reached.

The second variety is proof oriented. In these systems the tool will help one prove the safety or liveness requirements are satisfied through induction, by definition, leveraging assumptions, or *modus ponens* things we'd normally see in math class or set theory. `Verdi, COQ, Isabelle` are good examples.

Both modalities provide the programmer with two writing tools: writing state and state change in a mathematically precise language on types, and operations on elements in the types. For example, safety requirements often present as predicate like assertions (functions returning true or false) on current state. The second mode is temporal (time) formulae which say what the system must always from some point forward or eventually do in the future.

Formal methods are about project risk reduction: getting something to beta or production without fear the system will be continually beset with bad state, timeouts, or bad recovery after crashing. It can be about a company's street reputation: can we engineer complicated systems that work two days in a row? Work under duress or local failure?

Moving from a programmer mentality telling the CPU do this, then do that, then run this loop N times is certainly helpful in modeling. Indeed, a classical imperative program is kind of a constructive proof (when bug free) that takes a system or function call from its preconditions to its post conditions. So when programmers first see a formal tool's notions of conditions, types, loops, invariants, or state exploration, we find them accessible.

Nevertheless, it's still easy to get lost in modeling. Modeling is more like SQL: we say what we want done without saying how it gets done. It's hard to peek behind the curtain. Unexpected outcomes — including, say, no reported errors — may leave one anxious wondering *how* the error happened or indeed whether the tool checked what we hoped it did. This confusion has the nasty tendency to turn over on itself making the specification unclear in its effects. A programmer may find himself stuck: if one can't understand how the specification is applied, one can't know where he lands nor can one tweak the specification to a certain, better end.

Another challenge is avoiding one's programming muscle memory whereby we write imperative code in the formal language. This tendency has two ill-effects. First, it necessarily locks in or bounds reachable states or values through the implementation. In other words it tends to implicitly and silently prelude buggy behavior that the real system might have. Second, it can be too complex. For example, passing messages in queues across a network reliably turns on macro level things like no overflow, no loss of message, exactly once message processing, and no thread waits forever to send or receive messages. It does not turn on micro level things like memory layout, serialization format of messages, or whether Berkeley sockets or `HTTP GET/PUT` is used. Modelers may reasonably assume subject to unit and integration testing, professionals can take macro to micro once the harder problems are solved.

In this note we explore these issues for a hash based key-value store by specifying a model with attention to states in all their forms. Through exposition I attempt to `SQL EXPLAIN` what `TLA+` does. In later notes I'll expand this work to include temporal formulae, safety, and liveness. A KV store is both intuitively accessible while sufficiently complicated to

engage and disposition how best to use a formal tool. I will work in TLA+ PlusCal syntax, which is model based.

2 Organization

I start by giving instructions to set up a TLA+ tool environment. Second, I give the KV store goals informally. Third, I give a minimal TLA+ model then run it. Then, I review model results towards interpreting what might be concluded. Next, working in contrast mode, I compare imperative Python code to TLA+ extracting distinguishing conceptual and performance differences. At its conclusion one should have stronger sense of state, reachable states, and end-state. Also important is clearly seeing the difference between program behavior (imperative programming code paths) and specification state for safety and liveness checking. All code can be found here.

A non-goal is reviewing all of TLA+'s commands or modes. I limit the exposition to TLA+ commands and steps necessary to advance the KV store. However, TLA+ has extensive documentation on usage, examples, and on its technical foundations. See cite here. While there's a IDE called the *toolbox* cite download here which combines TLA+'s various modes, we'll use the command line which is functionally equivalent. That'll keep focus on modeling not mouse clicks, and screen shots.

3 Preliminaries

To get your environment ready, GIT clone TLA+. TLA+ is JAVA and uses ant to build it. While it's work to install JAVA, ant TLA+ is far easier to build than a similarly complex C/C++ program. Run the following commands or suitable variations thereto. Because of formatting, I've written backslash (\) to indicate line continuation. In your real commands, those lines would be on a single line omitting '\':

```
$ cd $HOME
$ mkdir TLAPlus
$ cd ./TLAPlus
$ git clone https://github.com/tlaplus\
/tlaplus.git
$ cd ./tlaplus/tlatools
$ cd ./org.lamport.tlatools
$ ant -f customBuild.xml
$ ls -ald $PWD/dist/tla2tools.jar
```

Once the unit tests start running I pressed CTRL-C since completion is not strictly required here.

TLA+ allows modeling in two syntaxes: native TLA+ and PlusCal. The latter is a very limited veneer over C. PlusCal is translated to native TLA+ with a helper function. I use PlusCal here. To complete the setup we'll need two helper scripts which run the model checker and PlusCal. You'll need to know the fully qualified path of the parent directory containing tla2tools.jar reading as PREFIX below:

```
$ cd $HOME
$ mkdir -p bin
$ cd bin
$ cat > ./runtla
#!/bin/sh

DEFAULT_JAVA_OPTS="-XX:+IgnoreUnrecognizedVMOptions\
-XX:+UseParallelGC"
PREFIX="/Users/smiller/TLAPlus/tlaplus/./tlatools/\
org.lamport.tlatools/dist"

if [ -z "$JAVA_OPTS" ]; then
  JAVA_OPTS="$DEFAULT_JAVA_OPTS"
fi

exec java $JAVA_OPTS -cp ${PREFIX}/tla2tools.jar\
tlc2.TLC "$@"
<CTRL-D>

$ cat > ./runpcal
#!/bin/sh

PREFIX="/Users/smiller/TLAPlus/tlaplus/./tlatools/\
org.lamport.tlatools/dist"

exec java -cp ${PREFIX}/tla2tools.jar\
pcal.trans "$@"
<CTRL-D>

$ chmod 550 runtla runpcal
```

Add `$HOME/bin` to your `PATH` and confirm everything is OK by asking for the command line options:

```
$ runtla -h
$ runpcal -h
```

Note that for reasons unclear to me, but probably color and terminal nonsense, `TLA`'s help line is partially obscured. Highlight it to see missing characters. In another directory download the `TLA+` model and Python code:

```
$ cd $HOME
$ mkdir -p note1
$ cd note1
$ git clone https://github.com/\
rodgarrison/tla_note1.git
$ cd tla_note1
$ ls
Makefile hash.cfg hash.tla
```

The `TLA+` the division of work is that `.tla` files define the model, possibly including other `.tla` files through the `EXTENDS` keyword, while `.cfg` files contain constants used in the model referenced in `.tla` files. `Makefile` is an optional hand-added convenience to execute `TLA+`. The default target `all` converts the PlusCal code in `hash.tla` then runs the model checker on the result. The optional fragment `-dump dot,colorize $(FILE).dot` tells `TLA` to produce an AT&T compatible `.dot` file representing the states it found and transitions between them. The makefile `dot` target converts the `.dot` file into a PDF and display it. Obviously you must have the `dot` utility pre-installed. Modify the second command in that target to taste; it's currently setup for `MacOS`.

It's important to know that the conversion of a `.tla` file in PlusCal syntax is converted to native `TLA+` syntax by writing the output into the same file! You don't get a second file out. Here the PlusCal lexer/parser is very unforgiving. You must adhere to several rules,

- The first line must include the filename without its extension surrounded by at least four dashes both sides
- Followed by EXTEND commands, CONSTANT commands. Symbols may span lines comma separated and are unterminated by a semi-colon
- Followed by a TLA comment of the form `(*--algorithm <name>` where `name` is a meaningful single string without spaces
- Followed by PlusCal code
- Terminated by a closing comment `end algorithm;*)`

Everything below `end algorithm;*)` is provided by the PlusCal translation and should not be touched.

Finally, it's helpful to define an alias to run `TLA's` command line REPL (read-eval-print-loop) so you can test `TLA+` commands. Unfortunately it's not as powerful as a Ruby or Python REPL where it can deal with variable assignments. It only works over constants and operations on constants. However, it beats writing a throw away specification whose side effect is test and print:

```
$ alias tlarepl='java -cp /Users/smiller/\
TLAPlus/tlaplus/./tlatools/org.lamport.\
tlatools/dist/tla2tools.jar tlc2.REPL'

$ tlarepl
Welcome to the TLA+ REPL!
TLC2 Version 2.15 of Day Month 20??
Enter a constant-level TLA+ expression.
(tla+) [ x \in 1..2 |-> 2*x]
<<2, 4>>
(tla+) <CTRL-D>
```

4 Modeling a Simple Hash

4.1 Hash Model: Pass1 - Orientation

I now model a hashing function which relates a key to a value. Modeling means choosing safety and liveness concerns that constitute first order risk. Other issues are deferred. In this first note, the hash will have one bucket per key and no hashing function. In fact, I'll leverage a built-in `TLA+` type called a record or structure that natively relates keys to values. The hash

will be of finite size. Keys and values will be integers in a limited range. This approach eliminates need for a equality value operator since hash collisions can't happen. The model allows for an *infinite number* of hash operations in arbitrary order. From there I'll model a finite number of calls, followed by arbitrary integers. Distributed operations and/or replication of KV state will await future notes.

`TLA+` comes with a type called variously *record* or *structure* of the form $[a_1|->b_1, a_2|->b_2, \ldots]$ where the a_n are like keys, and the b_n are like values. The infix arrow is written pipe character, minus character, then greater-than character without intervening spaces. More formally what appears between brackets is a function whose domain is the a_n mapped to elements in the range indicated by the b_n e.g. taking a_1 in the domain or pre-image to b_1 in the image or range. The type of a_n, b_n is arbitrary usually integers, strings, booleans, sequences/tuples, or sets. This will hold our hash. Once the hash is assigned to a variable, one writes `hash[key]:=value` for `PUT` operations, `hash[key]` read, and `hash[key]:=Nil` delete. `TLA` records do not support dynamic addition or removal in the domain/range. To handle the case where a key doesn't exist in the map we fallback to a unique constant `Nil` so that the `TLA` hash $[10|->100, 11|->Nil]$ means the same thing as Python {`10->100`}.

Let's relate these bits to file contents in the `tla_note1/pass1` subdirectory. The file `hash.cfg` names our constants explicitly defining allowable keys to be either 10 or 11 while allowable values are 100, 101. There was no reason to make their intersection empty other than to make them standout. The file also defines `Nil`, and contains a mandatory `SPECIFICATION Spec` fragment. `Spec` is what the tool will verify and is defined in `hash.tla`. `Spec` is the model capturing all the checks we want verified as computed by PlusCal after lex/parse.

The file `hash.tla` contains the model opening with boiler plate `EXTENDS` commands with intent similar to C/C++ `include` or Python `import`. The second line names constants we'll use in the specification; again the actual values for these are in the `.cfg` file. For the constant `ClientOps` $\in 0, 1, 2$, zero will mean `GET`, one `PUT`, and two `DELETE`. This can be made nicer, say, by using symbolic constants, but it's not important now. There is a single global variable called `Hash` initialized to [hashKey\in HashKey|− > Nil]. This means each element of the constant `HashKey` is mapped to a value `Nil` giving $[10|->\text{Nil}, 11|->\text{Nil}]$.

The rest of `hash.tla` contains one process (thread in the UNIX `pthreads` sense) called *Worker*. It is a `fair` process, which I'll return to later. The process runs a loop calling `PerformOp` which updates the hash relying on good old `if-then-else`. `skip` line 16 is equivalent to Python `pass`. Lines 13, 21 are single line comments introduced by $\backslash *$.

Line 16 reading with ... is an alternative phrasing of the Cartesian product HashKey × HashValue × ClientOps eventually producing twelve, distinct 3-tuples. It creates the tuples assigning the first element of each 3-tuple to `k`, the second element to `v`, and the third element to `op` which are passed into `PerformOp`.

Programmers will recognize the quasi-universal function or procedure form with formal arguments, and actual arguments at the call site. Like dynamic languages, the types of the arguments are set as a side effect of their initial assignment, and subsequent reassignment. Processes and procedures may declare additional local variables, but that's not shown here.

Procedures don't return values. In `TLA+` returning data must happen through global variables often with the keyword `self`. `self` behaves like Python's `self` except that it for choosing one's own PID. As such global variables may be indexed by `self` to pass or return data cleanly stratified by a process identifier.

Procedures must contain and hit a `return` statement otherwise `TLA+` concludes deadlock. Deadlock detection is enabled by default in the toolbox, but is off by default on the command line. Since deadlock detection can be off procedures missing `return` commands will lead to confusion: your specification will behave unexpectedly.

Procedure and process code is delimited by *labels*. These are single line alpha-numeric strings delimited by a colon *ex.* `expanded_loop_start:`. PlusCal conversion requires labels in certain locations, for example, just before the start of while loop. It'll report errors if missing. And in a few cases, PlusCal disallows labels within code blocks. `await` is one notable

case.

Code between labels is performed as an atomic step within the process that's running or invoking it. Consequently you decide how much or how little code is run to effect state change. More labels means a more granular step-wise change flow creating more states for `TLA+` to check. Beyond atomicity, more labels can — depending on what is shared gloabally — leak information to other processes of procedures. This may be leveraged to the good, or it may cause confusion.

More precisely, `TLA+` treats the label names like CPU `CS:IP` locations. Execution jumps in discrete steps label to label. Moreover, whenever a state step has been evaluated reaching the end of its labeled code block, `TLA+` finds all labels that are reachable from state state in all processes. Those labeled code blocks are queued for evaluation next. Failure to find next steps is called `deadlock` except when the process is sitting on `end process`.

There are more facets to this model, and this description omits numerous important details. That'll come later. Right now it's time to run the model.

4.2 Hash Model: Pass1 - Running It

While sitting in the subdirectory `tla_note1/pass1`, run `make`. About a second later it will report *No error has been found* plus some statistics (elided):

- 1 distinct initial state
- 217 states generated, 117 distinct states found
- 0 states left in the queue
- Depth of complete state graph search is 6
- Average outdegree is 1, minimum 0, maximum 12, 95th percentile 12

There are no errors because the specification does not yet contain safety or liveness checks. Since `TLA+` terminated with 0 states in the queue, there is no unfinished work. All reachable states were found and checked. `TLA+` also computes broad indicators on the state graph size giving its depth and branching factor. There is only 1 initial state. And there's at most 6 edges or steps between the initial state, and any other reachable state in the system. On average there is only one edge connecting the last checked state to its successor(s) *i.e.* most labels only have 1 successor label or step `TLA+` can check next. But in the worse case it's 12, a direct result of the `with` statement line 32.

Now, while this may appear quite reasonable, there is much to `TLA`'s work that is not indicated. To make this explicit, let's run down a few aisles and see what can found. First, uncomment line 13 so `print` is enabled. Then rerun `make`. `TLA` executes `print` 108 times with different combinations of keys, values, and operations. 108 is neither the Cartesian product 12 nor the 218 or 117 states `TLA+` claims exists in this model. Even more concerning is my earlier claim this model "allows for an *infinite number* of hash operations in arbitrary order." So are any of these numbers even in the right order of ten? And what was the value of `Hash` after any of these putative KV operation sequences?

If you have `dot` installed, run `make dot` to more directly engage these questions.

4.3 Hash Model: Pass2 - Python Detour

If we assign the label 1 to the first 3-tuple of the Cartesian product HashKey × HashValue × ClientOps, 2 to the second, and so on we can estimate the number of KV operations that could be executed here. Run the Python script `pass2/count_call_sequences` and find this total to be 1,302,061,345. This is a straightforward consequence of power sets and permutations:

1. Make the power set P of $[1, 2, 3, \ldots, 12]$
2. Total=0
3. For each set $s \in P$: Total = Total + factorial(cardinality(s))

Note that the number of permutations in a set of cardinality N is $N!$ This estimates the count of all possible sequences of KV calls to `PerformOp` which, among others, include,

- Each one sequence of the 12 KV operations
- Every possible sequence of 2 operations from among the 12 possible operations
- Every possible sequence of 3 operations from among the 12 possible operations
- Etc. up to all permutations on 12 KV operations

You can visualize these call sequences by running `pass2/see_call_sequences` but note that it'll take hours to complete.

4.4 Hash Model: Taking Stock

We are now in a position to disposition the several discrepancies between what `TLA+` actually did versus what we hoped `TLA+` did. In truth, `TLA+` did do the right thing. Note that while `TLA+` completes its work in about a second including PlusCal translation, and making the `.dot` file reporting 217 states (117 distinct), the Python analysis says there was over 1 billion call sequences. The Python code is slow: it'll take hours to emit all possible call sequences, and even longer to actually do it. A C++ version, clearly, will be faster but there'd still remain orders of 10 difference in its speed and `TLA+`'s. And careful readers might wonder how `TLA+` ever stopped at all: the while loop line 30 in `pass1/hash.tla` is unconditionally true.

While some implementation of this hash model might be deployed as a service running KV operations *ad infinitum*, there are in fact only 1,302,061,345 unique sequences of calls that could be run on an initially empty hash. If we suppose those operations are proved correct, it shouldn't be hard to show further KV operations are also correct. Now, if we extend the Python counting code to include those additional operations beyond the first 1,302,061,345 use cases, the total number of KV operations would become countably infinite. But none of this advances our understanding in any important way.

Our model imposes a finite number of keys, values, and operations running on an initially empty hash. And yet the total number of KV operations that might be run in production is countably infinite. It is a fact that it *is not required to generate* and run all 1,302,061,345 call sequences the imperative way to verify the model. This is not what `TLA+` does.

This is not to assign blame. After all, the final production code should work if clients run 1, 1 billion, or 1 trillion KV operations on whatever state is in the hash map. Moreover, programmers are keenly aware the correctness of a KV operation o_n depends on whether the hash state is correct from the previous $o_1, o_2, \ldots, o_{n-1}$ operations. Some how that concern needs to be added to the list too. Still, the confusion is that the imperative mind focuses on system behavior, which arises by concerning oneself with generating all possible code execution paths.

The modeling mind set, however, focuses on different concerns ultimately answering programmer concerns. `TLA+` tracks *all* variables and their values in *all* processes at *all* steps. This notion begins to make precise what a state really is inside `TLA+`. Next, `TLA+` carefully tracks which of those states are distinct. Of critical consequence is that while the program behavior allows for an unbounded number of KV operations in an arbitrary order, there are in fact only 217 states in the modeling sense of which 117 are unique. Therefore, `TLA+` need only concern itself with checking safety and liveness requirements on 117 states. This is why it finishes so quickly.

To make this explicit readers are strongly encouraged to run the `dot` target and view the resulting PDF. In top middle locate a shaded circle showing the initial state. Carefully notice it lists the `Hash` global variable, all variables local to `PerformOp` with their initial values. Also note the entry pc: << worker_begin >>. This identifies the `CS:IP` or starting label where model checking will begin. `stack` is used for internal `TLA+` purposes and counter-examples on invariance violation. That — that! — is the initial state in the `TLA+` sense.

Now, each outgoing arrow from the initial state takes you to a valid, possible successor state through one of the possible KV operations. The destination circle, again, records state at the termination of that step. There are only 117 nodes in this graph corresponding to `TLA+`'s report of 117 distinct states. On

the hand all nodes have outgoing edges. Therefore it is impossible to find a path of finite size through this graph *i.e.* the program behavior is unbounded.

This goes someway to explain how `TLA+` escapes the while loop line 30 of `pass1/hash.tla`. It's a combination of two factors. From a standpoint of semantics the PlusCal code `while (TRUE)` is misleading. Interpreted like a conventional imperative while loop, `TLA+` really should not "escape" it. However, the PlusCal code is converted to native `TLA+` which is evaluated. The semantics change here: `TLA+` is looking for successor states. As such, once `TLA+` sees there can be no more distinct states generated out of that loop, it "exits it" because it was never per-se in an imperative style loop from the perspective of finding reachable states. Note that later we'll see how the while loop comes with termination issues, however.

In summary, the graph is both finite and unbounded. It is finite because there is only 117 distinct states. It is unbounded because there are no terminal nodes. That is, all nodes have outgoing edges so it's impossible to find a finite path from the initial state to a node lacking outgoing edges. The modeling focus is on nodes, while the imperative mind set is on paths through the graph by following edges.

It's not true that `TLA+` "ran all possible sequences of KV operations." `TLA+` didn't run the hash "code" the imperative way which is why the print statement was only hit about 100 times. From this observation and technical limitations on `print` communicated by Merz/Loncaric, it is generally not possible to elicit from `print` what `TLA+` is "doing." `print` is not the poor man's `SQL EXPLAIN` in modeling.

Note that `.dot` files are not the only way to visualize model state or behavior. Readers are referred to github for more information. `_TEPosition`, `_TETrace` appear to be powerful tools, for example.

5 Generating Model States

Before changing focus to a more `TLA+` tooling mode, there remain two unexplored issues relative to model states. First, one should have passing knowledge on how model states are generated. It isn't hard: it's based on depth first graph generation of successor states. A toy, Python example is provided for hashing. Second, the dueling perspectives of imperative *v.* modeling isn't completely resolved. This is the more complicated problem both caught up in, and resolved by breadth first graphs. The model state graph generation runs like this:

1. Let G be a graph of nodes, edges
2. Let visited, a dictionary, be empty
3. Let queue, an array, equal initial state
4. While queue not empty do
   (a) Let s = Head(queue)
   (b) Let queue = Tail(queue)
   (c) If s in visited, jump to while above
   (d) If s fails an invariant, produce counterexample. Stop.
   (e) Put s into visited
   (f) Add s to G. Add an edge from s to s (stuttering)
   (g) Let succ, array, be successor states from s
   (h) For t in succ add an outgoing edge from s to t
   (i) Append succ to queue

See 14.3.1 in cite for detailed guidance. A potential concern is that once `TLA+` marks a state visited, will it check invariance on a successor s_n state against all of the *previous states* in all their permutations it previously saw? Did it see all possible permutations previously? Where? `(4c)` appears to be the source for a negative outcome here. Now, it will. But why?

For example, suppose `TLA+` marks the initial state I_0, and the first successor state s_1 visited. Roughly speaking `TLA` modeled I_0, s_1 concluding no violations so far. There is an edge between nodes $I_0 \rightarrow s_1$ in G. The nodes hold state visually communicated by a `.dot` file. By now we know any of the 12 KV operations is always possible from a behavioral (imperative) standpoint. That is, no matter where we are in the AT&T `.dot` graph, every node always has at least one outgoing edge by running one of arbitrary

12 KV operations on it. So suppose `TLA+` determines a reachable state from s_1 is s_2. Suppose s_2 is the state we get by running by running `Put(11,101)`. Three questions arise,

- Will the edge $I_0 \rightarrow s_2$ ever appear in G or has this somehow been precluded by `(4c)` since I_0 is marked visited? It's not enough that `Put(11,101)` runs on s_1 making a state s_2. The model and its evaluation must insure the state resulting from `Put(11,101)` is run on every valid prefix of earlier operations.
- What is `Put(11,101)` applied against exactly? We seemingly have two choices I_0, s_1 each getting us to the same resulting state. However, that might be dumb luck.
- What again does `TLA+` care about? I thought we just got done saying nodes. Now we're back to edges?

Two final points are needed. First, `TLA+` won't have done its job if the graph fails to have the right number of nodes, or the right number of incoming and outgoing edges to the right nodes. For performance and conceptual reasons I earlier gave emphasis to node count for modeling, but pathways through edges for programming. The truth is `TLA+` can't run and hide from either.

Second, successor operations are always applied against s in step `(4g)`). The problem, however, is `(4g)` doesn't make it abundantly clear how this works. The key insight is to recall s contains all variables, and their values from all processes. `(4g)` may then be read like this:

- Let succ=[]
- Let $L = \{l | l$ valid, enabled label from $s\}$
- For each $l \in L$
  - Let t be a copy of s
  - Update t by applying work at label l on it
  - Append t to succ
- Append succ to queue

The Python program `pass3/dfs-python` generates nodes in G but not edges. Adding edges would not be difficult from which a simple `.dot` file might result. Because the Python code is not label or process aware, it only makes 62 nodes rather than `TLA's` 117. Lines 42-44 are the Python equivalent to trying every KV operation in all orders. Lines 45-47 correspond to `(4g)` above. The hash global variable, and `PerformOP` local variables are kept in a very simple `State` object. `State` has a class method `serialize` which returns the object state as a string. That string is used to check for existence in `visited`.

Indeed, on the first pass through the while loop in `(4)`, succ will contain the result of all 12 possible KV operations. The exploded version of `(4g)` will copy $s = I_0$ then eventually apply `Put(11,101)` from one of the 12 operations making node s_2 with a $I_0 \rightarrow s_2$ edge. This will eventually generate all successor states from I_0 whence the process repeats. From there proof by induction will demonstrate all orderings are found.

The genius of depth first generation is that it *generates all operations in all orders*, but only requires memory proportional to the number of distinct states resulting from those operations. So even though this model has an unbounded behavior with combinatorial ordering of operations, the node count is finite. On the other hand, the nodes are connected with edges. In this model no node is terminal: it'll always have outgoing edges representing operations on state faithfully recording the program's infinite behavior.

6 Extending Model in TLA+

6.1 Finite Behavior

Thus far the model's behavior is infinite: KV operations are run without bound subject only to specific key, values. `pass4/hash.tla` gives a model in which just 12 KV operations *i.e.* HashKey × HashValue × ClientOps are run in order. The resulting `.dot` model is significantly different. Node count is still finite but some nodes are terminal because, at some point, no more operations are possible. Consequently the graph's topology is a straight line. `pass5/hash.tla`

also models finite behavior but in which TLA+ chooses arbitrary key, value, operations subject only to an upper limit on attempts. By modifying the limit line 31, you'll see the number of reachable end sates increase or decrease. The state graph topology remains straight, however.

In pass6/hash.tla I relax the hash key, values to a less arbitrary integer not a value from a finite set. In fact it's not possible to ask TLA+ to use an arbitrary integer or natural number. TLA+ can choose arbitrarily, however, only from an enumerable set or bounded range. Since constants are not applicable here, I define HashRange to be a bounded interval from which TLA+ chooses. This isn't markedly different from pass1/hash.tla; it's mildly more configurable. But it makes clear that even small increments in range drive up reachable state counts exponentially. A range of [1..2] gives 117 distinct states, [1..3] 1732 states, and [1..6] 12,823,741 states taking four minutes to find.

One can extract two practical conclusions: infinite behavior models are comprehensive, and may even take fewer lines to model. Prefer those. Second, obsessing about value ranges isn't smart most of the time. What is smart is allowing TLA+ to choose arbitrary values for you within a bounded range, or finite set. Model evaluation will happen faster, and you are more likely to see errors since you have not excluded value or proscribed ordering. Readers should see CHOOSE, EITHER, and WITH for more options on the same theme.

6.2 Coverage Checks

A TLA+ program is punctuated by a set of labels, L. But periodically one of the labels won't be hit. That is, a label $l \in L$ might never become an enabled, valid successor. And that's probably a specification error. TLA+ will help find those cases as shown in pass7/hash.tla. It manufactures a trivial omission to calling SpecialOp through the condition if k=1000. Since k was limited to either 10 or 11 this condition will never be true.

TLA+ is instructed to check for label coverage by specifying -coverage 1 in Makefile's all target. The integer argument to coverage makes TLA+ emit coverage data every N minutes where $N = 1$ here. Upon running the all target you'll see additional output in the console (elided, formatted):

```
<perform_op_begin line 73, col 1 to line 73,
 col 22 of module hash>: 8:108
line 73, col 30 to line 73, col 58 of module hash: 225
line 73, col 30 to line 73, col 37 of module hash: 117
<special_op_begin line 89, col 1 to line 89,
 col 22 of module hash>: 0:0
line 89, col 30 to line 89, col 58 of module hash: 117
line 91, col 30 to line 91, col 77 of module hash: 0
line 92, col 30 to line 92, col 86 of module hash: 0
```

The number following the trailing colon counts how many times that line in the PlusCal translated model was hit. Lines reading ": 0" are not hit corresponding to SpecialOp's text. Note these lines refer to translated code. Therefore, you'll need to interpret the TLA+ native code to assess what PlusCal code it derives.

6.3 Termination

PlusCal can optionally check for process termination by adding the switch -termination to its command line. The translation will add a Done label at end process with a temporal formulae that it must eventually become a visited state. This is demonstrated two ways in pass8/term1.tla, pass8/term2.tla. The first model is a copy of pass5/hash.tla while term2.tla is a copy of pass1/hash.tla but with termination enabled. Run make term1 or make term2 to evaluate the models.

pass8/term1.tla is error free with respect to termination because the while loop eventually becomes false. When false, the Done label becomes an enabled successor. However, pass8/term2.tla fails because, with respect to termination, the while loop does not terminate. From a model state generation perspective, as note earlier, TLA+ realizes all reachable states have been visited moving to other concerns like termination whereby it termination is not possible.

When a safety, liveness, or other temporal formula fails TLA+ gives a counter-example (to the invariant) read top to bottom from state 1 (initial)

to the last state N. The trace is a kind of stack trace for models showing how the initial state was transformed by labeled code blocks into bad state. In `pass8/term2.tla` we find (elided):

```
State 1: <Initial predicate>
/\ pc = <<"worker_begin">>
State 2: <worker_begin line 72, col 23
         to line 86, col 37 of module term2>
/\ pc = <<"perform_op_begin">>
State 3: <perform_op_begin line 57, col 27
         to line 68, col 80 of module term2>
/\ pc = <<"worker_begin">>
State 4: <worker_begin line 72, col 23
         to line 86, col 37 of module term2>
/\ pc = <<"perform_op_begin">>
Back to state 1: <perform_op_begin line 57,
    col 27 to line 68, col 80 of
    module term2>
```

This isn't always easy to interpret but be assured the information is there. The location information on lines beginning with "State N:" is the source node state location. The data following this line gives the state (at destination node) after doing the work indicated by label `pc=`. Note the pc labels alternate `worker_begin`, `perform_op_begin`. One interprets this as meaning `TLA+` found an infinite cycle in the model *viz* `.dot` which prevents the process from ever visiting `Done`.

Here's a variation on termination. In the subdirectory `tla_note1/pass9` I model a producer process which appends integers to the end of a list, and a consumer process which dequeues the head of the list. Run `make term1`. After a few seconds, press CTRL-C. It'll never stop running because it'll never stop finding more states. Now run `make term2`, which runs the same model as `term1` but with termination checks. It will end with an error since this model won't terminate.

Now inspection of `pass9/term.tla` might suggest a variation on `pass1/hash.tla` granted with two processes in that `TLA+` ought to be able to find all reachable states. Indeed, we've seen infinite behavior on a finite number of states before, and `TLA+` did terminate there. However, unlike the hash example, the length of `Queue` is unbounded. In the hash example, there were only a bounded number of number of keys, and values therefore only a bounded number of reachable states. Here the producer process can always append numbers faster than the consumer can dequeue them. That means `TLA+` has to check the empty queue, queue of length 1, then 2, and so on on all permutations of contents in the queue. In such a case (assuming fairness; see below) the depth first graph generation of states will always have a non-empty queue.

6.4 Deadlock

Quoting Lamport cite

> An algorithm is deadlocked if it has not terminated, but it can take no further step. A process has terminated if it has reached the end of its code, so control is at the implicit Done label that ends its body.

Termination is about never eventually visiting the `Done` label, which was co-located within the code at `end process`. On the other hand, deadlock is about no successor states except in the case of termination. Deadlock then is a more expansive concept than, say, merely buggy lock code. In fact, writing lock code in `TLA+` is less about deadlock per se than it is about temporal formulae that prove no two threads are in the critical section, and threads wanting the lock eventually get it. Locking is a fairly solved problem anyway.

Instead, this section's example is the dining philosopher problem. There's N philosophers who dine together around a circular table. Each person has a plate in front of them, and forks to their left and right. Three modes of behavior are possible. If they are `hungry`, they grab forks to their left, and right if not used and go into the *eating* mode. Once they are done eating, they revert to the *thinking* mode. And when they are done thinking, they go into the *hungry* mode again. The challenge is to model the three modes without deadlock. Deadlock happens when a philosopher cannot transition into another behavior.

`pass10/deadlock.tla` gives a model with $N = 3$ taking the description straight into code. Now while

it's obvious that there will be contention around forks — adjacent philosophers always share a fork between them when sitting in a circle — it seems like every person will make progress. There has to be some time at which two forks are available. Running the model shows that deadlock occurs after checking some 1800 distinct states with a counter-example that consists of 13 steps away from the initial state. In particular, the trace shows this 13th state:

```
State 13: <P line 107, col 12 to line 114,
          col 28 of module deadlock>
/\ state = (0 :> "Hungry" @@
            1 :> "Hungry" @@
            2 :> "Hungry")
/\ pc = (0 :> "E" @@ 1 :> "E" @@ 2 :> "E")
/\ fork = (0 :> 1 @@ 1 :> 2 @@ 2 :> 0)
```

All philosophers are in hungry mode, and each philosopher has one fork each *i.e.* diner 0 has fork 1, diner 1 fork 2, and diner 2 fork 0. In addition they are all on label E where they are waiting to get their second fork. But no fork is available. States 11 and 12 show two of the three philosopher's getti

Turning deadlock detection off shows 1959 distinct states with a depth of 19. Thus the deadlock scenario doesn't happen immediately. Deadlock detection is enabled by default in both the IDE and on the command line. Confusingly, the command line switch `-deadlock` disables deadlock detection. See `TLA's` usage line by running `runtla -h` for more information.

6.5 Backdoor To The Command Line

You may prefer the IDE in `TLA`'s toolbox to the command line. But even command line users will find plenty to like. The IDE organizes and writes the multitude of configuration options into files you can copy and revise later. If you can't figure out the command line directly, make your model work in the IDE then leverage those files. The IDE logs command lines it runs in `$HOME/.tlaplus/.metadata/.log`. Find the command line switches there.

Recall the `.cfg` files hold the constant values. And since, as a matter of practice, it's easier to validate models by placing different checks into different configuration sets, the IDE supports multiple configurations (what it unfortunately calls models) per `.tla` file. In the IDE toolbox sense, different configurations refers both to constants whose number or values might change, plus all the invariants, temporal formulae, filters, and liveness checks enabled in that model *e.g.* termination or deadlock. For this reason the toolbox collects all those variance points into a per subdirectory structure. Its approach is to make a new file set `MC.tla, MC.cfg` that include the user's model code. The toolbox runs `MC.tla` which is why you can't use that name yourself in the IDE.

Assume `example.tla` and two configurations or models `Model_1, Model_2`. The directory format follows:

- file `example.tla` *i.e.* your model file
- subdirectory `example.toolbox`
  - subdirectory `Model_1`
    - file `MC.cfg`
    - file `MC.tla`
    - file `example.tla`
  - subdirectory `Model_2`
    - file `MC.cfg`
    - file `MC.tla`
    - file `example.tla`

Each per configuration `example.tla` file is a copy of the original `example.tla` modified to include the invariants, temporal formulae, filters, and liveness checks for that configuration. The toolbox automatically makes it. Running the model means running `MC.tla` inside the configuration directory. `MC.tla` always includes `example.tla` in the same directory. I've omitted temporary files, for example, those containing console output and print results. The toolbox also writes the results of each model run into its own snapshot directory not shown here. They're transient output files.

Note that the IDE allows you to define constant values which are not actually valid with the `CONSTANT` command. To work around this, the toolbox makes

a GUID in the form `constant_<N>` where N is a large, random integer. It then defines a constant with the modeler's real name referring to the name of the GUID in `MC.cfg`. It then assigns the actual values to the GUID in `MC.tla` as a global variable. The intuition is that those global variables are hidden and unknown to the programmer's `.tla` file and hence effectively constant.

7 Future Work

This note describes how `TLA+` models states through the dueling perspectives of program behavior, and state determination itself. Behavior is about following nodes via their edges, while states arise by carefully tracking what possible new state results after running all possible enabled actions (labels) in all their permutations. Behavior can be infinite while states are finite. `TLA+` uses depth first graph traversal to combine these facets which, for example, an AT&T `.dot` artifact can be produced.

This treatment, however, is woefully incomplete. For example, deadlock has strong connections to fairness which went unexamined. Notable tooling options were not discussed either. To contend with large state models, `TLA+` provides certain bounding constraints, and filtering modes that narrow `TLA`'s work to a subset of the full state space. PlusCal, while adequate for a lot of work, cannot do everything native `TLA+` can. Therefore programmers should be versed in writing native models, or at least know how to work in the last bits into native `TLA+` combining it with what PlusCal produces. Finally, the KV store examined here is trivial failing to admit uncontrived safety or liveness checks.

The KV store here is part of a larger effort to formally model Stanford's `RAM`Cloud, a distributed KV system with durable writes cite. Future notes therefore will eventually address all of the following,

- Fairness
- Safety, liveness checks
- Temporale formulae
- All within the context of a true, complex distributed KV store

8 TLA+ Resources