Verifying Software Traces Against a Formal Specification with TLA$^+$ and TLC

A question that often comes up with regards to TLA$^+$, especially among beginners, is how to verify that the implemented software conforms to the TLA$^+$ specification. We'd like to mechanically verify that our code is a refinement of a high-level specification.

While this is possible – at least in the lab – by compiling the code into TLA$^+$ and checking refinement,[1] it greatly suffers from scalability limitations and can only work for very small programs. This scalability problem is not unique to TLA$^+$ (or to TLC); formally verifying low-level code against a high-level specification (often called end-to-end verification, although sometimes the term refers only to cases when even the machine code is verified) is extremely difficult regardless of tool or technique used, and has only been accomplished for small programs, and even then at great cost.

Experienced TLA$^+$ users would recognize that this is hardly a big issue. Not only is end-to-end verification too cost prohibitive to be all but completely infeasible for all but small, niche software, it is hardly ever a requirement. In practice, we use formal verification to help us write more correct software, and the costliest bugs occur at the algorithm or system design level, and not at the code level, particularly as we can specify at low-enough a level to ensure that any translation errors would be easy to find and fix (possibly with the aid of code-level verification of simple, local properties).

Nevertheless, the more we can verify *affordably* the better, and, as it turns out, there is a relatively cheap way to verify that a running software system conforms with a specification, although not with the sound guarantees that would have been afforded by end-to-end verification. Instead of verifying that our code is a refinement of the specification – which amounts to checking that *all* possible behaviors of the program are allowed by the spec – we check that *some* observed behaviors are. We do that by capturing execution traces of the system in logs, and checking them against the high-level specification with the aid of a model-checker. This works even when the log is collated from multiple machines in a distributed system, each running a program written in a different programming language.

If we obtain a trace with a visible bug, this can help us pinpoint where and how the bug originates. But even the trace does not lead to an observable bug, the approach might detect disallowed behavior that indicates a bug. Faulty behaviors are not evenly distributed, hence the unsoundness of the approach, which only examines a sample of behaviors, but the larger the sample, the higher the chances of catching a bug early.

This approach is called "model-based trace-checking" (`https://arxiv.org/pdf/1111.2825.pdf`), and it is particularly easy with TLA$^+$ and TLC. All it takes – regardless of how complex your specification or how partial the log (although a richer log yields more confidence in the verification) – is adding a few lines to your spec and writing a TLC plugin to ingest the log files.

But to get there, we need to understand TLC's capabilities (and how to exploit them) and limitations (and how to work around them).

Note: In order to write this as a single document, I introduced a number of inner modules to avoid naming clashes. Real specifications will not require this extra complication.

---

[1] For example, see `https://cedric.cnam.fr/fichiers/art_3439.pdf` and `http://tla2014.loria.fr/slides/methni.pdf` for C, and `https://www.researchgate.net/publication/224262035_Java_goes_TLA` for Java.

MODULE *Trace*

Suppose we had the following high-level system specification:

EXTENDS *Naturals*, *Sequences*

MODULE *System*

VARIABLES x, y, z, *tickTock*
$vars \triangleq \langle x, y, z, tickTock \rangle$

$TypeOK \triangleq \land x \in Nat$
$\land y \in Nat$
$\land z \in Nat$
$\land tickTock \in \{\text{“tick”}, \text{“tock”}\}$

$Init \triangleq \land x \in 0 \,..\, 9$
$\land y \in 0 \,..\, 9$
$\land z = 0$
$\land tickTock = \text{“tick”}$

$Next \triangleq \lor \land tickTock = \text{“tick”}$
$\land tickTock' = \text{“tock”}$
$\land z' = x + y$
$\land$ UNCHANGED $\langle x, y \rangle$
$\lor \land tickTock = \text{“tock”}$
$\land tickTock' = \text{“tick”}$
$\land x' \in 0 \,..\, 9$
$\land y' \in 0 \,..\, 9$
$\land$ UNCHANGED z

$Safety \triangleq Init \land \Box[Next]_{vars}$ Just the safety part of the spec
$Spec \triangleq Safety \land \mathrm{WF}_{vars}(Next)$

Then, suppose we implement the specification and obtain an execution trace in a log file. We would like to know whether the trace is consistent with our specification.

MODULE *Trace*1

We could inline the trace in the specification and "read" it as follows, or we could write a TLC intrinsic in Java that would read the trace directly from a log file. Writing such a log analysis module for TLC is a nice weekend project.

Tuples are: $\langle x, y, z, tickTock \rangle$

$Trace \triangleq \langle\langle 1, 0, 0, \text{"tick"}\rangle, \langle 1, 0, 1, \text{"tock"}\rangle, \langle 1, 1, 1, \text{"tick"}\rangle, \langle 1, 1, 2, \text{"tock"}\rangle,$
$\langle 0, 3, 2, \text{"tick"}\rangle, \langle 0, 3, 3, \text{"tock"}\rangle, \langle 2, 2, 3, \text{"tick"}\rangle, \langle 2, 2, 4, \text{"tock"}\rangle,$
$\langle 3, 2, 4, \text{"tick"}\rangle, \langle 3, 2, 5, \text{"tock"}\rangle, \langle 2, 4, 5, \text{"tick"}\rangle, \langle 2, 4, 6, \text{"tock"}\rangle,$
$\langle 5, 2, 6, \text{"tick"}\rangle, \langle 5, 2, 7, \text{"tock"}\rangle, \langle 4, 4, 7, \text{"tick"}\rangle, \langle 4, 4, 8, \text{"tock"}\rangle,$
$\langle 2, 7, 8, \text{"tick"}\rangle, \langle 2, 7, 9, \text{"tock"}\rangle, \langle 6, 4, 9, \text{"tick"}\rangle, \langle 6, 4, 10, \text{"tock"}\rangle\rangle$

VARIABLES $x,\ y,\ z,\ tickTock$

$Model \triangleq$ INSTANCE $System$
$vars \triangleq \langle x, y, z, tickTock \rangle$ If we write $Model!vars$, TLC complains.

VARIABLE i the trace index

"Reading" a record is just $vars = Trace[i]$, but unfortunately TLC isn't happy with that, so:
$Read \triangleq$ LET $Rec \triangleq Trace[i]$ IN $x = Rec[1] \land y = Rec[2] \land z = Rec[3] \land tickTock = Rec[4]$

Unfortunately, TLC also isn't happy with just $Read'$ – which is equivalent to:
$ReadNext \triangleq$ LET $Rec \triangleq Trace[i']$IN $x' = Rec[1] \land y' = Rec[2] \land z' = Rec[3] \land tickTock' = Rec[4]$

$Init \triangleq i = 1 \land Read$
$Next \triangleq \lor i < Len(Trace) \land i' = i + 1 \land ReadNext$
$\lor$ UNCHANGED $\langle i, vars \rangle$ So that we don't get a deadlock error in TLC

$TraceBehavior \triangleq Init \land \Box[Next]_{\langle vars,\, i \rangle}$

Because we're dealing with a finite trace, we only care about safety properties, as liveness concerns only infinite behaviors.
THEOREM $TraceBehavior \Rightarrow Model!Safety$

To verify, we check the spec *TraceBehavior* in TLC , with $Model!Safety$ as a temporal property. As we're always wary of success, we modify the above trace to ensure that TLC finds an error.

Because I split this document into modules (for ease of writing this as a post), and because TLC doesn't support checking a specification inside an inner module so, to check, we add the following, outside of the inner module:

VARIABLES $x,\ y,\ z,\ tickTock,\ i$
INSTANCE $Trace1$

If we wish to use trace-checking not to analyze an error trace, but to gain confidence that our system implements our specification, it is important to check many traces. We can use the same technique to check multiple traces at once:

MODULE *Trace*2

Traces $\triangleq$ [*log*1 $\mapsto$
⟨⟨1, 0, 0, "tick"⟩, ⟨1, 0, 1, "tock"⟩, ⟨1, 1, 1, "tick"⟩, ⟨1, 1, 2, "tock"⟩,
⟨0, 3, 2, "tick"⟩, ⟨0, 3, 3, "tock"⟩, ⟨2, 2, 3, "tick"⟩, ⟨2, 2, 4, "tock"⟩,
⟨3, 2, 4, "tick"⟩, ⟨3, 2, 5, "tock"⟩, ⟨2, 4, 5, "tick"⟩, ⟨2, 4, 6, "tock"⟩⟩,
*log*2 $\mapsto$
⟨⟨5, 2, 0, "tick"⟩, ⟨5, 2, 7, "tock"⟩, ⟨4, 4, 7, "tick"⟩, ⟨4, 4, 8, "tock"⟩,
⟨2, 7, 8, "tick"⟩, ⟨2, 7, 9, "tock"⟩, ⟨6, 4, 9, "tick"⟩, ⟨6, 4, 10, "tock"⟩⟩,
*log*3 $\mapsto$
⟨⟨3, 4, 0, "tick"⟩, ⟨3, 4, 7, "tock"⟩, ⟨0, 9, 7, "tick"⟩, ⟨0, 9, 9, "tock"⟩,
⟨2, 2, 9, "tick"⟩, ⟨2, 2, 4, "tock"⟩, ⟨2, 6, 4, "tick"⟩, ⟨2, 6, 8, "tock"⟩⟩]

VARIABLES x, y, z, *tickTock*
Model $\triangleq$ INSTANCE *System*
vars $\triangleq$ $\langle x, y, z, tickTock\rangle$

VARIABLE *log*, the *log* file
i the trace index

Trace $\triangleq$ *Traces*[*log*]
Read $\triangleq$ LET *Rec* $\triangleq$ *Trace*[i] IN $x = Rec[1] \land y = Rec[2] \land z = Rec[3] \land tickTock = Rec[4]$
ReadNext $\triangleq$ LET *Rec* $\triangleq$ *Trace*[i'] IN $x' = Rec[1] \land y' = Rec[2] \land z' = Rec[3] \land tickTock' = Rec[4]$

Init $\triangleq$ *log* $\in$ DOMAIN *Traces* $\land$ $i = 1 \land$ *Read*
Next $\triangleq$ $\land \lor i < Len(Trace) \land i' = i + 1 \land$ *ReadNext*
$\lor$ UNCHANGED $\langle log, i, vars\rangle$
$\land$ UNCHANGED *log* Each trace follows a single log

TraceBehavior $\triangleq$ $Init \land \Box[Next]_{\langle log,\, i,\, vars\rangle}$

THEOREM *TraceBehavior* $\Rightarrow$ *Model*!*Safety*

VARIABLES x, y, z, *tickTock*, i, *log*
INSTANCE *Trace*2

While that is the best way to verify traces against a formal specification because it allows checking many traces at a time, it may be the case that we don't log all of the real system's internal state that corresponds to all the variables in our specification. Suppose our real system only logs the value of z:

$Trace \triangleq \langle 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\rangle$

VARIABLES i, z
$tvars \triangleq \langle z, i\rangle$

$InitTrace \triangleq i = 1 \land z = Trace[1]$
$NextRecord \triangleq$ IF $i < Len(Trace)$
THEN $i' = i + 1 \land z' = Trace[i']$
ELSE $i' = i \land z' = z$ We don't use UNCHANGED to prevent problems later, when we compose this definition.

Ideally, we would like to check the following proposition:

$TraceBehavior \Rightarrow \exists\, x, y, tickTock : Model!Safety$

But TLC cannot check such a theorem. Unlike the previous limitations, this is not a minor implementation detail. Checking a specification with temporal quantifiers may require a time-complexity exponential *in the number of states* . To check that proposition, we need to come up with a refinement mapping from *TraceBehavior*, which requires adding auxiliary variables to it:

MODULE $Trace3$

We need a variable that introduces stuttering into the trace behavior to allow for internal state changes in the model.
VARIABLE tt

$Init \triangleq InitTrace \land tt = 0$
$Next \triangleq \lor \land i < Len(Trace)$
$\land\, tt' = 1 - tt$
$\land \lor tt = 0 \land NextRecord$
$\lor\, tt = 1 \land$ UNCHANGED $tvars$
$\lor$ UNCHANGED $\langle tt, tvars\rangle$ So that we don't get a deadlock error in TLC

$TraceBehavior \triangleq Init \land \Box[Next]_{\langle tt,\, tvars\rangle}$

$Model \triangleq$ INSTANCE $System$ WITH $tickTock \leftarrow$ IF $tt = 0$ THEN "tick" ELSE "tock",
$x \leftarrow$ IF $tt = 0$ THEN z ELSE $z - 1$,
$y \leftarrow 1$

THEOREM $TraceBehavior \Rightarrow Model!Safety$ As before, this is what we check

VARIABLES tt
INSTANCE $Trace3$

But creating a refinement mapping is not only work that has to be tailored to every specific system specification rather than being completely automatic – it can be difficult. Not only did we have to introduce an auxiliary variable to introduce stuttering into the trace behavior, we had to compute legal values for the specification's internal variables x and y. This is not very hard in this case, but it could get tricky. Luckily, we can make the model-checker work for us.

$$TraceBehavior \triangleq InitTrace \land \Box[NextRecord]_{tvars}$$

We want to check a proposition of the sort $B \Rightarrow A$ (where A is $Model!Safety$ and B is $TraceBehavior$), but as we've seen, this is only possible with a manually written refinement mapping. However, in our case, B , the trace behavior, is not an arbitrary specification but a *single* behavior (well, up to stuttering and all unmentioned variables), and we can make use of that. But because we will be exploiting that feature, unlike the previous techniques, this one can only work on a single trace at a time. Because of that, it may be appropriate as a tool to help understand what has gone wrong in an error trace.

What we really need is to find out whether the trace is a *possible* behavior of the system's specifation. In other words, we want to verify that $Model!Safety \Rightarrow \neg TraceBehavior$ (*i.e.* that no behavior is the trace behavior) is *not* a theorem. But TLC is limited in what temporal properties it can check, and $\neg TraceBehavior$ is not one of them, and neither, I believe, is any other equivalent formula. But we can still do what we want; in fact, we can do it in a way that is better, as it (may) make the model checker run much, much faster, by not trying all behaviors.

We note that $B \Rightarrow A$ iff $B \land A \equiv B$. This does not help us in general because the model checker can only check implication, not equivalence. Checking $B \land A \equiv B$ is the same as checking both $B \land A \Rightarrow B$ (which is trivially true), and $B \Rightarrow B \land A$, and the latter is hard for the same reason I mentioned above, as it requires temporal quantification.

However, in our case, because B is a single behavior (sort-of), checking $B \land A \equiv B$ is the same as checking that $B \land A$ is not empty. To get there, some work still needs to be done, but it's easy, mechanical, and always the same.

First, if A and B are temporal formulas, TLC can't even check the specification $A \land B$ as it's not in the canonical (or "normal") form. This is easily resolved with some formal manipulation:

We notice that if $A \triangleq InitA \land \Box[NextA]_{varsA}$ and $B \triangleq InitB \land \Box[NextB]_{varsB}$ then $A \land B \equiv (InitA \land InitB) \land (\Box[NextA]_{varsA} \land \Box[NextB]_{varsB})$

$InitA$ and $InitB$ are fine, but $(\Box[NextA]_{varsA} \land \Box[NextB]_{varsB})$ is still not in canonical form. But we notice that (the calculation below includes steps that are ill-formed in TLA):

$$\begin{aligned}
& \Box[NextA]_{varsA} \land \Box[NextB]_{varsB} \\
\text{(by TLA)} \quad & \equiv \Box(NextA \lor \text{UNCHANGED } varsA) \land \Box(NextB \lor \text{UNCHANGED } varsB) \\
\text{(by PTL)} \quad & \equiv \Box((NextA \lor \text{UNCHANGED } varsA) \land (NextB \lor \text{UNCHANGED } varsB)) \\
\text{(by PL)} \quad & \equiv \Box(\lor NextA \land NextB \\
& \qquad \lor NextA \land \text{UNCHANGED } varsB \\
& \qquad \lor \text{UNCHANGED } varsA \land NextB \\
& \qquad \lor \text{UNCHANGED } varsA \land \text{UNCHANGED } varsB) \\
\text{(by TLA)} \quad & \equiv \Box[\lor NextA \land NextB \\
& \qquad \lor NextA \land \text{UNCHANGED } varsB \\
& \qquad \lor NextB \land \text{UNCHANGED } varsA]_{\langle varsA, varsB \rangle}
\end{aligned}$$

Now we could write a composition operator,

$$\begin{aligned}
Compose(NextA,\ varsA,\ NextB,\ varsB) \triangleq\ & \lor NextA \land NextB \\
& \lor NextA \land \text{UNCHANGED } varsB \\
& \lor \text{UNCHANGED } varsA \land NextB
\end{aligned}$$

and get:

$A \land B \equiv \mathit{InitA} \land \mathit{InitB} \land \mathit{Compose}(\mathit{NextA},\ \mathit{varsA},\ \mathit{NextB},\ \mathit{varsB})$

(I have used the exact same transformation when I wrote about specifying in the behavioral programming style: `https://pron.github.io/files/TicTacToe.pdf`)

Another complication is that in our case, $A \land B$ can never be empty, because it admits various stuttering behaviors. Instead, by being a bit clever, we'll ask TLC whether the composed specification contains our trace; this will be possible because it will not require checking a complex temporal formula.

MODULE *Trace4*

Unfortunately, due to TLC limitations, we can't write the operator as above, but defining it as follows, and later defining *UnchTrace* and *UnchSystem* is not too bad. When this technique is used in a real specification, the definition of *UnchSystem* is the only specification-specific line here.

$\mathit{Compose}(\mathit{NextA},\ \mathit{UnchangedA},\ \mathit{NextB},\ \mathit{UnchangedB}) \triangleq$
$\quad \lor\ \mathit{NextA} \quad \land\ \mathit{NextB}$
$\quad \lor\ \mathit{UnchangedA} \quad \land\ \mathit{NextB}$ If B is a trace this disjunct is FALSE, as $\mathit{UnchangedA} \Rightarrow \mathit{UnchangedB}$
$\quad \lor\ \mathit{NextA} \quad \land\ \mathit{UnchangedB}$

VARIABLES $x,\ y,\ \mathit{tickTock}$
$\mathit{vars} \triangleq \langle x,\ y,\ z,\ \mathit{tickTock} \rangle$

$\mathit{UnchTrace} \triangleq z' = z \land i' = i$ UNCHANGED *tvars* doesn't work, and neither does $\mathit{tvars}' = \mathit{tvars}$
$\mathit{UnchSystem} \triangleq x' = x \land y' = y \land z' = z \land \mathit{tickTock}' = \mathit{tickTock}$ ditto

$\mathit{Model} \triangleq$ INSTANCE *System*

$\mathit{ComposedSpec} \triangleq$ $\mathit{Model}!\mathit{Safety} \land \mathit{TraceBehavior} \equiv$
$\quad \land\ \mathit{Model}!\mathit{Init} \land \mathit{InitTrace}$
$\quad \land\ \Box[\mathit{Compose}(\mathit{Model}!\mathit{Next},\ \mathit{UnchSystem},\ \mathit{NextRecord},\ \mathit{UnchTrace})]_{\langle \mathit{vars},\ \mathit{tvars} \rangle}$

$\mathit{TraceFinished} \triangleq i \geq \mathit{Len}(\mathit{Trace})$ Our secret weapon is this definition, which is TRUE when the trace has finished.

Finally, to check if *ComposedSpec* contains the trace behavior, all that's required is to check that the following is *not* a theorem. This is done by letting TLC check *ComposedSpec*, and adding $\neg \mathit{TraceFinished}$ as an invariant, essentially asserting that the trace *never* finishes, and challenging TLC to prove us wrong. We also need to turn deadlock checking off.

$\mathit{Check} \triangleq \mathit{ComposedSpec} \Rightarrow \Box(\neg \mathit{TraceFinished})$

If (and only if) our trace conforms to the spec, TLC will report a violation of the invariant, along with a trace that contains the inner states TLC has computed for us. However, if the trace does not conform, there is no trace emitted that can help us pinpoint the issue. What we can do is change the definition of *TraceFinished* to become TRUE after a short prefix of the trace, and so find which state is in violation, *e.g.*, to say $i \geq 7$ rather than $i \geq \mathit{Len}(\mathit{Trace})$.

VARIABLES $x,\ y,\ \mathit{tickTock}$
INSTANCE *Trace4*