Using TLC to Check Inductive Invariance

Leslie Lamport

4 July 2018

1 Inductive Invariance

Suppose we have a specification with initial predicate *Init* and next-state predicate *Next*, so its specification is $Init \wedge \Box[Next]_{vars}$ for the tuple *vars* of all its variables.[1] Suppose we want to prove that a formula $\mathcal{I}$ is an invariant of the spec, which means that $\mathcal{I}$ is true on all reachable states of the spec. We prove that $\mathcal{I}$ is an invariant by finding a formula *Inv* that satisfies these conditions:

1. $Init \Rightarrow Inv$, which means that *Inv* is true in all initial states.

2. $Inv \wedge Next \Rightarrow Inv'$, which means if *Inv* is true on any state s, then it's true on any state reachable from s by a *Next* step.[2]

3. $Inv \Rightarrow \mathcal{I}$, which means that $\mathcal{I}$ is true in every state on which *Inv* is true.

The first two conditions imply, by induction, that *Inv* is true on every reachable state of the spec. A formula satisfying these conditions is called an *inductive invariant* of the spec. Condition 3 then implies that, if *Inv* is an inductive invariant of the spec, then $\mathcal{I}$ is true on every reachable state of the spec, so it's an invariant. (If $\mathcal{I}$ happens to be an inductive invariant, we could let *Inv* equal $\mathcal{I}$ so condition 3 is trivial; but this is seldom the case for the invariants we care about.)

Finding an inductive invariant to prove invariance is hard if you haven't done it many times. Even if you've done it many times, it's hard to get the inductive invariant right. It would be very helpful to people writing proofs to have a tool that can check if a formula is an inductive invariant

[1] Fairness conditions of the spec are irrelevant for invariance, so they can be ignored.

[2] I'm assuming that all the variables in *Inv* are in the tuple *vars*.

of a spec. For TLA$^+$, the obvious choice for such a tool is the TLC model checker. Unfortunately, so far, TLC has been of only limited use in checking inductive invariance. Let's see why.

If you've used TLC, then you know that it can check that a formula is an invariant of a spec. What you may not know is that it can also be used to check if a formula is an inductive invariant. In principle, it's simple. TLC can check condition 1 by checking that the spec satisfies the property *Inv*. (Add it to the *Properties* list in the What to check? section of the *Model Overview* page.) Condition 3 is usually ensured when defining *Inv*, for example by simply making it a conjunct of *Inv*. The tricky part is checking condition 2. The trick is that condition 2 is satisfied if and only if *Inv* is an invariant of the specification

$$(1) \quad ISpec \triangleq Inv \wedge \Box[Next]_{vars}$$

which has initial predicate *Inv* and next-state predicate *Next*. Simple, right? Well, not so simple.

TLC can't handle a specification with any old state predicate as its initial condition. You couldn't use most of the invariants you might want to check as the initial predicate of a spec in TLC. So why should we expect TLC to be able to handle *Inv* as an initial predicate? The answer is that *Inv* almost always must assert type correctness of all the variables, so a natural way to define *Inv* is

$$(2) \quad Inv \triangleq TypeOK \wedge \mathcal{H}$$

where *TypeOK* asserts type correctness and $\mathcal{H}$ is the interesting part. Let's suppose that the spec contains the three variables x, y, and z. The definition of *TypeOK* will probably have the form

$$(3) \quad \begin{array}{l} \wedge\ x \in S \\ \wedge\ y \in T \\ \wedge\ z \in U \end{array}$$

for some set expressions S, T, and U. If TLC can evaluate these set expressions then it can, in principle, handle *Inv* as an initial predicate.

Unfortunately, in principle doesn't mean in practice. When *Inv* is defined by (2), TLC computes the set of initial states by first computing all states satisfying *TypeOK*, then throwing away the ones that don't satisfy $\mathcal{H}$. If *TypeOK* equals (3), then the number of states satisfying it is $|S|*|T|*|U|$, where $|S|$ is the number of elements in the set S. In practice, even for fairly small models, the number of type-correct states is enormous. For example,

in a spec I once wrote for the bakery algorithm, a model with 3 processes that can choose numbers in $0\,..\,4$ has about $1.6 * 10^{13}$ type-correct states. TLC can't compute such a huge set of states—even though it takes a laptop only about 10 minutes to check an ordinary invariant on that model.

For an invariant *Inv* with *TypeOK* equal to (3), TLC works by first computing the possible values of x, then for each value of x computing the possible values of y, then for each pair of values of x and y computing the possible values of z, and then throwing away states that don't satisfy $\mathcal{H}$. Suppose $\mathcal{H}$ equals $(x < y) \wedge \mathcal{K}$. We can then rewrite *Inv* as

$$\begin{array}{l} \wedge\ x \in S \\ \wedge\ y \in T \\ \wedge\ x < y \\ \wedge\ z \in U \\ \wedge\ \mathcal{K} \end{array}$$

When written this way, for each pair of values TLC finds for x and y, it checks that the pair satisfies $x < y$ before finding possible values of z. (The conjunct $x < y$ must come after the two conjuncts that determine possible values of x and y.) Moving conjuncts from $\mathcal{H}$ up in the formula like this can reduce by a few orders of magnitude the amount of work TLC must do to compute the initial states, which can allow it to check a larger model. But reducing $1.6 * 10^{13}$ by a few orders of magnitude doesn't help much.

This method of checking inductive invariance is complete, meaning that it will find any error that can occur for a given model. However, it works only on *extremely* tiny models. The tiniest of models will probably reveal errors in your first attempts at an inductive invariant. Eventually, you'll correct all the errors a tiny model can find. You would then like to check it on a larger model.

When complete checking on a larger model becomes impractical, we can switch to probabilistic checking. Here's how it works with the inductive invariant *Inv* defined by (2) and the type-correctness invariant (3). Instead of checking that *Inv* is an invariant of (2), we check that it's an invariant of the specification $\mathit{IInit} \wedge \Box[\mathit{Next}]_{\mathit{vars}}$, with

$$\mathit{IInit} \triangleq \mathit{ITypeOK} \wedge \mathcal{H}$$

where *ITypeOK* equals

$$\begin{array}{ll} (4) & \wedge\ x \in RS \\ & \wedge\ y \in RT \\ & \wedge\ z \in RU \end{array}$$

and RS, RT, and RU are randomly chosen subsets of S, T, and U containing c, d, and e elements respectively. TLC then has to examine only $c * d * e$ type-correct states instead of $|S| * |T| * |U|$ of them. We choose c, d, and e so that $c * d * e$ is small enough so TLC can generate the initial states within a few minutes.

Probabilistic checking is not perfect. Any error it finds produces a counterexample showing that the formula *Inv* isn't an inductive invariant. The counterexample is extremely useful in correcting a problem with *Inv*, getting us closer to an inductive invariant. But failure to find an error proves nothing. The best we can hope for is that checking that *Inv* is an invariant of $IInit \wedge \Box[Next]_{vars}$ has a sufficiently high probability of finding an error in a suitably short length of time. We then run multiple tests, stopping tests if they run too long.

How well does this work? I've tried it on models of five specs. Three were (unpublished) specs I wrote many years ago containing what I then believed were inductive invariants. Probabilistic checking showed that they weren't. In two other examples, probabilistic checking found errors that I introduced into correct inductive invariants. The performance ranged from good to spectacular. On a specification of the Paxos consensus algorithm, in 6 out of 10 tries probabilistic checking found an error in about 17 minutes on a model that is too large for TLC to run ordinary model checking on. On a simpler, abstract version of the Paxos algorithm, every test found an error in seconds on a model on which it takes TLC about $2\frac{3}{4}$ days to do ordinary model checking using 16 cores on a large, fast machine.

The method now needs to be tested on more examples. If you have a spec with an inductive invariant that you can send me, that would be fine. Please propose one or more errors to add for checking. The most interesting errors are ones that weaken the inductive invariant so it's still an invariant, since they can't be found by ordinary invariance checking.

Although I'd be happy to have examples sent to me, I would prefer that you test the approach yourself. You're likely to use it in ways I wouldn't think of, perhaps by taking advantage of what you know about the spec. I'd like to find out what did and didn't work for you, so I can help others use the method.

2 How to Do it Yourself

The latest I will describe how to do probabilistic random inductive invariant checking using the example above, where we check that *Inv* is an invariant

of $IInit \wedge \Box[Next]_{vars}$, where $IInit$ equals $ITypeOK \wedge \mathcal{H}$ and $ITypeOK$ equals (4). If conjuncts of $\mathcal{H}$ can be moved into $TypeOK$ as described above to improve the efficiency of complete inductive invariance checking, they can be moved in the same way into $ITypeOK$.

You will start by doing complete inductive invariance checking on a tiny model. Only when you have done this on the largest model you can and found no errors should you try probabilistic checking. This requires writing $IInit$ in the form (4), which requires describing the randomly chosen subsets RS, RT, and RU in TLA$^+$.

2.1 Describing Random Subsets

You should install the latest version of the Toolbox, which is Version 1.5.7 of 18 July 2018. It contains a new standard module named *Randomization* that defines $RandomSubset(k, S)$ to equal a pseudo-randomly chosen subset of S containing k elements. The $RandomSubset$ operator makes a new pseudo-random choice every time it is evaluated. In our example, you can define the predicate $IInit$ to equal

$$\begin{array}{ll} (5) & \wedge\ x \in RandomSubset(c, S) \\ & \wedge\ y \in RandomSubset(d, T) \\ & \wedge\ z \in RandomSubset(e, U) \end{array}$$

To evaluate $RandomSubset(c, S)$, TLC must enumerate the elements of the set S. If S has a huge number of elements, this can take too long. A common case of this in a type-correctness condition $x \in S$ is when S equals SUBSET W, so it contains $2^{|W|}$ elements. A very large set of type-correct states is often the result of an initial condition of the form $x \in$ SUBSET W. (This condition sometimes appears in a type-correctness invariant in the form $x \subseteq W$, and it must be rewritten as $x \in$ SUBSET W for inductive invariant checking.) We could solve this problem by writing

$$x \in RandomSubset(c, \text{SUBSET } RandomSubset(p, W))$$

for some p. However, this is not ideal because all the resulting elements of SUBSET W this chooses are subsets of the same subset of W. There's another problem with this solution. If TLC can model check the spec, then the possible values of x probably comprise only a small number of subsets of W. This probably means that in reachable states, the value of x will be a small subset of W. So in choosing the subsets of W initially assigned to x, we want to choose mostly small subsets. Both these problems are solved by the $RandomSetOfSubsets$ operator, where $RandomSetOfSubsets(c, p, W)$

equals a pseudo-randomly chosen set of about c subsets of W, where the probability of any element of W being in a chosen subset of W is $p \,/\, |W|$. The operator works by choosing c subsets of W, each one being chosen by enumerating the elements of W and randomly deciding whether or not to put that element in the subset—putting it in the subset with probability $p \,/\, |W|$. This can produce fewer than c subsets because the same subset can be chosen more than once. I don't know how to compute in general the expected number of subsets this produces as a function of c, p, and $|W|$. But you can let TLC tell you how many subsets it generates for specific values of the parameters by using the *TestRandomSetOfSubsets* operator—at least for the common case in which c, p, and W are constants. The expression

(6) $\quad$ *TestRandomSetOfSubsets*(c, p, W)

equals a sequence of five integers that are the cardinalities of the sets produced by evaluating *RandomSetOfSubsets*(c, p, W) five times. When c, p, and W are constants, you can put expression (6) in the Evaluate Constant Expression field of a model's *Results* page to have TLC evaluate it. Trial and error will quickly lead you to the value of c that produces the number of sets of subsets you want.

2.2 Running Tests

You should start with a model that is too big for complete inductive invariance checking, but not very big. You should use one in which you have already checked that *Inv* is an ordinary invariant of the spec. To write *IInit* in the form (5), you must then choose the parameters c, d, and e. If the parameters are too small, then TLC will find no initial states. (None of the type-correct states will satisfy $\mathcal{H}$.) If the parameters are too large, TLC will spend a long time computing the initial states. I've found that TLC will find an error in at least about half its tries with fewer than 100 initial states. (Often a single initial state is enough.) So, you should increase the parameters until TLC finds a few dozen initial states. For a small model, it will take TLC just a few seconds to compute those initial states.

Of course, there are many ways to choose c, d, and e to produce the same number $c*d*e$ of initial states. It seems reasonable to base the relative sizes of these parameters on the numbers $|S|$, $|T|$, and $|U|$ of type-correct values of the variables in the model. I find it helpful to use a spreadsheet that computes the values of $|S|$, $|T|$, and $|U|$ based on the parameters of the model. For that computation, remember that the cardinality of $[V \to W]$ is $|W|^{|V|}$ and the cardinality of SUBSET W is $2^{|W|}$.

After you've chosen the model and the parameters c, d, and e, you can run TLC to check if *Inv* is an invariant of the spec $IInit \land \Box[Next]_{vars}$. If TLC finds an error, *Inv* is not an inductive invariant of your spec and TLC's error trace will show you why it isn't. TLC may quickly terminate without finding an error. It may also keep running for a long time. With complete inductive invariance checking, any error is found in a single step. With probabilistic invariance checking, it could take many steps. However, I've never seen it take an error trace of more than 10 steps to reveal an error. If the statistics section of the Model Checking Results page shows that TLC's breadth-first search has reached a diameter greater than around a dozen (which usually takes just a few seconds), then I advise stopping the run. You should try about a dozen runs with the same model and parameters before deciding that further tries won't find an error.

If you don't find an error, try adjusting the relative size of the parameters c, d, and e while keeping the total number of type-correct states about the same. If you still find no error, I suggest introducing an error in the interesting part $\mathcal{H}$ of the inductive invariant that you believe should be detectable with the model and check that TLC finds it. If TLC doesn't find the error, try increasing the parameters to get more initial states. If TLC finds the introduced error, then I believe it's quite likely that *Inv* is an inductive invariant for that model, so you should try checking larger models.

As your models get larger, you will have to increase the parameters c, d, and e to get enough initial states, and it will take TLC longer to compute those initial states. The largest part of the execution time in inductive invariant checking is generally spent finding the initial states. This is done sequentially by TLC. Hence, if you have a multi-core computer, it's best to use those cores by running multiple instances of TLC, each using a single worker thread. You can create and save a model with the Toolbox, and then run multiple instances of TLC on the `MC.tla` file created for that model. (You can find that file by searching inside the spec's `.toolbox` directory.) You can run multiple instances of TLC in the same directory by redirecting each instance's output to a different file. In a Unix (or Cygwin) shell, you can follow the command to run TLC on the `MC.tla` file with

```
&> filename.$(date +%s)
```

This creates a unique file name for the output (as long as successive commands are executed more than one second apart). You can check the progress of all the runs by running `grep` on the output files to look for these phrases indicating what TLC is up to:

`Finished computing`	TLC has finished computing the initial states.
`violated`	TLC has found an error. (It can still take it a while to produce the error trace.)
`Finished in`	The TLC run has terminated.

If TLC finds an error, the error trace will appear in its output file.

2.3 Ordering Conjuncts

The order of the conjuncts in the type-correctness invariant (3) makes no difference, assuming S, T, and U are constants (so their values don't depend on x, y, or z). However, this is not the case for (5). If you remember how TLC computes initial states, you will see that the $c * d * e$ initial states TLC computes that satisfy (5) contain c different values of x, but up to $c * d * e$ different values of z.

I don't know how to determine which order results in a set of initial states that is most likely to reveal the error. It probably depends on the spec and is hard to figure out. The best approach seems to be to choose the ordering that minimizes the time needed to compute the initial states. With the ordering of (5), $\mathit{RandomSubset}(c, S)$ is computed once and $\mathit{RandomSubset}(e, U)$ is computed $c * d$ times. Minimizing execution time requires ordering the conjuncts in decreasing order of the time needed to compute their sets. The ordering is probably significant only if there are large difference in the execution times, so the following very rough approximations of those times will suffice. The time to compute $\mathit{RandomSubset}(c, S)$ can be taken to be $|S|$ except in two special cases: (i) if S equals $[V \to W]$, then it is approximately $c * |V|$; and (ii) if S is the set $[l_1 : W_1, \ldots, l_n : W_n]$ of records, then it is approximately $n * |W|$ where $|W|$ is the average of the $|W_i|$. The time to compute $\mathit{RandomSetOfSubsets}(c, p, S)$ is roughly $c * |S|$.

3 Why Does this Work?

I don't know. However, I do have one possible partial answer. Suppose we have found an invariant J that is almost inductive. It can be turned into the desired inductive invariant Inv by conjoining a small number of conditions. Each of those conditions rules out a significant fraction of the states satisfying J. Therefore, if we choose a random state that satisfies J, there is a good chance that it does not satisfy Inv; and it's almost certain that a couple of dozen randomly chosen states will include ones that don't satisfy Inv, and so could potentially lead to states that don't satisfy J. What

I don't understand is why there is a good chance that one of those initial states that satisfies $J \wedge \neg Inv$ will, in a few steps satisfying the next-state relation, lead to a state not satisfying J.