Current Versions of the TLA$^+$ Tools

Leslie Lamport

28 February 2020

This document describes differences between the descriptions of the TLA$^+$ tools in the book *Specifying Systems* and the currently released versions. References are to the version of the book currently available on the web. The book and this document do not describe the features provided by the TLA$^+$ Toolbox for using the tools. They are described by the Toolbox's help pages and the TLA$^+$ Hyperbook.

Contents

1 SANY (The Semantic Analyzer)

The current release of SANY has no known limitations.

2 TLC

2.1 Limitations

Below are all the known ways in which the current release of TLC differs from the version described in the book.

- TLC doesn't implement the $\cdot$ (action composition) operator.
- TLC cannot handle definitions that comes from a parametrized instantiation. For example, suppose and module M, which has the variable parameter x, defines the specification *Spec*. If you define *ISpec* by

  ```
  IM(x) == INSTANCE M
  ISpec == IM(xbar)!Spec
  ```

  then TLC will not be able to check the property *ISpec*. However, TLC will be able to check *ISpec* if it's defined in the following equivalent way:

  ```
  IM    == INSTANCE M WITH x <- xbar
  ISpec == IM!Spec
  ```

- TLC cannot handle natural numbers greater than $2^{31} - 1$.

2.2 Additional Features

2.2.1 Enhanced Replacement

Most users now run TLC from the Toolbox, where overriding of definitions is performed in the Definition Override section of the model's Advanced Model Options page. Search for *override* in the Toolbox's Help pages.

When running TLC from the command line, definition overriding is specified by the configuration file. As described in *Specifying Systems*, when running TLC on a module M, a replacement

```
foo <- bar
```

replaces *foo* by *bar* in all operators either defined in M or imported into M through EXTEND statements. (For example, if M extends $M1$ which extends $M2$, then the replacement will occur in operators defined in $M1$ and $M2$, as well as in M.) It does not perform the replacement on any operators imported into M by an INSTANCE statement. The replacement

```
foo <-[Mod] bar
```

replaces *foo* by *bar* in all operators defined in module *Mod* or imported into *Mod* through EXTEND statements. You should use this if you want the replacement to be made in a module *Mod* that is instantiated either by the module M on which TLC is being run, or by some module imported into M through EXTEND statements.

An operator may be imported into the current module by multiple paths. For example, the identifier *Nat* can be imported directly from the *Integers* module by an EXTENDS statement or indirectly through an instantiated module, often under a different name. In that case, to redefine *Nat* to equal $0\,..\,2$, it's safest to put both of the following in the configuration file:

```
Nat <- 0..2
Nat <- [Integers] 0..2
```

2.2.2 Strings

TLA$^+$ defines strings to be sequences, but the TLC implementation does not regard them as first-class sequences. The Java implementation of the *Sequences* module has been enhanced so that $\circ$ and *Len* do what they should for strings. For example, TLC knows that "ab" $\circ$ "c" equals "abc" and that *Len*("abc") equals 3. However, *Len* does not work right for strings containing special characters written with "\". (See the bottom of page 307 of the TLA$^+$ book.)

Having TLC Set Values

TLC can now read and set a special list of values while evaluating expressions. This works as follows. The *TLC* module defines two new operators:

$$\begin{array}{ll} TLCGet(i) & \triangleq \ \text{CHOOSE } n : \text{TRUE} \\ TLCSet(i, v) & \triangleq \ \text{TRUE} \end{array}$$

When TLC evaluates $TLCSet(i, v)$, for any positive integer i and arbitrary value v, in addition to obtaining the value TRUE, it sets the i^{th} element of the list to v. When TLC evaluates $TLCGet(i)$, the value it obtains is

the current value of the i^{th} element of this list. For example, when TLC evaluates the formula

$$\begin{array}{l} \land\ \mathit{TLCSet}(42, \langle \text{“a”}, 1 \rangle) \\ \land\ \forall\, i \in \{1, 2, 3\} : \land\ \mathit{Print}(\mathit{TLCGet}(42), \text{TRUE}) \\ \qquad\qquad\qquad\qquad \land\ \mathit{TLCSet}(42, [\mathit{TLCGet}(42) \text{ EXCEPT } ![2] = @ + 1]) \end{array}$$

it prints

```
<< "a", 1 >>  TRUE
<< "a", 2 >>  TRUE
<< "a", 3 >>  TRUE
```

One use of this feature is to check TLC's progress during long computations. For example, suppose TLC is evaluating a formula $\forall\, x \in S : P$ where S is a large set, so it evaluates P many times. You can use *TLCGet*, *TLCSet*, and *Print* to print something after every 1000^{th} time TLC evaluates P.

As explained in the description of the *TLCEval* operator below, you may also want to use this feature to count how many times TLC is evaluating an expression e. To use value number i as the counter, just replace e by

$$\text{IF } \mathit{TLCSet}(i,\ \mathit{TLCGet}(i) + 1) \text{ THEN } e \text{ ELSE } 42$$

(The ELSE expression is never evaluated.)

For reasons of efficiency, *TLCGet* and *TLCSet* behave somewhat strangely when TLC is run with multiple worker threads (using the `-workers` option). Each worker thread maintains its own individual copy of the list of values on which it evaluates *TLCGet* and *TLCSet*. The worker threads are activated only after the computation and invariance checking of the initial states. Before then, evaluating $\mathit{TLCSet}(i, v)$ sets the element i of the list maintained by all threads. Thus, the lists of all the worker threads can be initialized by putting the appropriate *TLCSet* expression in an ASSUME expression or in the initial predicate.

TLCEval

TLC often uses lazy evaluation. For example, it may not enumerate the elements of a set of the form $\{x \in T : P(x)\}$ unless it has to; and it doesn't have to if it only needs to check if an element e is in that set. (TLC can do that by evaluating $x \in T$ and $P(e)$.) TLC uses heuristics to determine when it should completely evaluate an expression. Those heuristics work well most of the time. However, sometimes lazy evaluation can result in the expression

ultimately being evaluated multiple times instead of just once. This can especially be a problem when evaluating a recursively defined operator.

You can solve this problem with the *TLCEval* operator. The *TLC* module defines the operator *TLCEval* by

$$TLCEval(x) \triangleq x$$

TLC evaluates the expression $TLCEval(e)$ by completely evaluating e.

If TLC is taking a long time to evaluate something, you can check if lazy evaluation is the source of the problem by using the *TLC* module's *TLCSet* and *TLCGet* operators to count how many times expressions are being evaluated, as described above.

Any

Originally, TLA$^+$ allowed only functions to be defined recursively. One problem with this was that it's sometimes a nuisance to have to write the domain of the function f. There were two reasons it might be a nuisance: the domain might be complicated, or TLC might spend a lot of time when evaluating $f[x]$ in checking that x is in the domain of f. The operator *Any* was added to the *TLC* module As a hack to work around this problem. With the introduction of recursive operator definitions, this problem disappeared and there is no reason to use *Any*. However, it is retained for backwards compatibility. Here is its description.

The definition of the constant *Any* doesn't matter. This constant has the special property that, for any value v, TLC evaluates the expression $v \in Any$ to equal TRUE. You can avoid having to specify the domain in a function definition by letting the domain be *Any*.

The use of *Any* sounds dangerous, since it acts like the set of all sets and raises the specter of Russell's paradox. However, suppose a specification uses *Any* only in function definitions without doing anything sneaky. Then for any execution of TLC that terminates successfully, there is a finite set that can be substituted for *Any* that yields the same execution of TLC. That set is just the set of all values v for which TLC evaluates $v \in Any$ during its execution. However, unrestricted use of *Any* can get TLC to verify incorrect modules. For example, it will evaluate $Any \in Any$ to equal TRUE, even though it equals FALSE for any actual set *Any*.

You should not use *Any* in an actual specification; it is intended only to help in using TLC. In the actual specification, you should write the definition like

$$f[x \in Dom] \triangleq \ldots$$

where the domain *Dom* is either defined or declared as a constant parameter. In the configuration file, you can tell TLC to substitute *Any* for *Dom*.

PrintT
The *TLC* module defines

$$PrintT(out) \triangleq \text{TRUE}$$

However, evaluating $PrintT(out)$ causes TLC to print the value of *out*. This allows you to eliminate the annoying "`TRUE`" produced by evaluating $Print(out, \text{TRUE})$.

RandomElement
The *TLC* module defines

$$RandomElement(S) \triangleq \text{CHOOSE } x \in S : \text{TRUE}$$

so $RandomElement(S)$ is an arbitrarily chosen element of the set S. However, contrary to what the definition says, TLC actually makes an independent choice every time it evaluates $RandomElement(S)$, so it could evaluate

$$RandomElement(S) = RandomElement(S)$$

to equal FALSE.

When TLC evaluates $RandomElement(S)$, it chooses the element of S pseudo-randomly with a uniform probability distribution. This feature was added to enable the computation of statistical properties of a specification's executions by running TLC in simulation mode. We haven't had a chance to do this yet; let us know if you try it.

ToString
TLA$^+$ defines $ToString(v)$ to be an arbitrarily chosen string whose value depends on v. TLC evaluates it to be a string that is the TLA$^+$ expression whose value equals the value of v. By using *ToString* and string concatenation ($\circ$) in the argument of the *Print* or *PrintT*, you can get TLC to print nicer-looking output than it ordinarily does.

2.2.3 Typed Model Values

One way that TLC finds bugs is by reporting an error if it tries to compare two incomparable values—for example, a string and a set. The use of model values can cause TLC to miss bugs because it will compare a model value to

any value without complaining (finding it unequal to anything but itself). Typed model values have been introduced to solve this problem.

For any character τ, a model value whose name begins with the two-character string "τ_" is defined to have type τ. For example, the model value x_1 has type x. Any other model value is *untyped*. TLC treats untyped model values as before, being willing to compare them to anything. However it reports an error if it tries to compare a typed model value to anything other than a model value of the same type or an untyped model value. Thus, TLC will find the model value x_1 unequal to the model values x_ab2 and *none*, but will report an error if it tries to compare x_1 to a_1.

2.2.4 Overriding Modules

TLC permits definitions from a module M to be overridden by Java code in a file M`.class`. This is used primarily for implementing standard modules, but it can be applied to any module if you are willing to write the appropriate Java code. Beginning with Version 2.09 of TLC (appearing in release 1.5.3 of the tools), you can put the `.class` file in the same folder/directory as the module's `.tla` file. See the files in the `tlc2.model` package to see how the Java code is written.

2.3 Command-Line Options

Several command-line options have been added to TLC since *Specifying Systems* was written. Moreover, the book did not list all the options available then. Most users now run TLC from the Toolbox, which provides a convenient way to specify the most commonly used TLC options; few people will use command-line options. However, since there is no conveniently available list of all those options, they are presented here. A parameter *file* is the path name of a file—either an absolute path or one relative to the directory from which TLC is run. Similarly, a parameter *dir* is the absolute or relative path name of a directory.

`-aril` ***num***
: Adjusts the seed for random simulation. (See page 251 of the book.) It defaults to 0 if not specified.

`-checkpoint` ***num***
: Tells TLC to take a checkpoint every *num* minutes. The default is 30.

`-cleanup`
: Cleans up the states directory, removing all existing checkpoint files.

`-config` ***file***
: Provides the configuration (`.cfg`) file. Defaults to `spec.cfg` if not provided.

`-continue`
: Normally, TLC stops when it finds a violation of a property it is checking. This option tells TLC to continue running when it finds a violation of a safety property. (It always stops when a liveness property is violated.)

`-coverage` ***num***
: This option tells TLC to print coverage information every *num* minutes. Without the option, TLC prints no coverage information.

`-deadlock`
: This tells TLC not to check for deadlock.

`-debug`
: Tells TLC to print information useful for debugging its own code.

`-depth` ***num***
: Specifies the depth (number of steps) of a random simulation. Without this option, the default depth is 100.

`-dfid` ***num***
: Directs TLC to do depth-first model checking with iterative deepening, beginning with initial depth *num*.

`-difftrace`
: Tells TLC to show only the differences between successive states when printing an error trace. Otherwise, it prints the full state descriptions.

`-dump` ***format file***
: The *format* parameter can be omitted, or it can be a comma-separated list beginning with `dot` that may also contain one or both of the items `colorize` and `actionlabels`. If *format* is omitted, TLC writes a list of all reachable states, described by TLA$^+$ formulas, on *file*. Otherwise, TLC writes the state graph in dot format, the input format of the GraphViz program for displaying graphs. The parameter `colorize` indicates that state transitions should be colored according to the action generating the transition, and `actionlabels` indicates that they should be labeled with the name of the action.

`-fp` ***num***
: TLC's state fingerprinting algorithm uses one of a list of irreducible polynomials, numbered 0 through 130. This option tells it to use polynomial number *num*. Through release version 1.5.7 of the tools, The default is to used number 0. In later versions the default will be to use a randomly chosen one.

`-fpbits` ***num***
: Directs TLC to partition its fingerprint set into 2^{num} separate disk files. (On some systems, using multiple files can improve efficiency of reading and writing fingerprints when they don't fit in memory.) The default value of *num* is 0.

`-fpmem` ***num***
: Tells TLC how much memory to use to store the fingerprints of found states. If *num* is an integer, it specifies the number of megabytes; it it's a fraction between 0 and 1, it specifies that fraction of the memory size.The default value of *num* is .25.

`-gzip`
: This tells TLC to compress the state queue when writing it to disk.

`-help`
: Causes TLC to type a help message and stop.

`-lncheck` ***param***
: If this is omitted or *param* equals `default`, TLC performs liveness checking periodically, when the number of distinct states it finds increases by 10%. If *param* equals `final`, TLC does liveness checking only after it has computed the complete state graph.

`-maxSetSize` ***num***
: The cardinality of the largest set that TLC can handle. TLC reports an error if it tries to compute a set containing more elements. It defaults to 1000000 if this option is not specified.

`-metadir` ***dir***
: Tells TLC to store its metadata in the directory given by *dir*. Without this option, the default is to use the `states` subfolder of the directory containing the specification file.

`-modelcheck`
: Tells TLC to run in model checking mode, which is the default.

`-nowarning`
: Tells TLC not to issue any warnings. Otherwise, TLC reports all warnings. (A warning indicates a possible error, but does not cause TLC to stop.)

`-recover` ***dir***
: Recover from the checkpoint found in the directory specified by *dir*. If not specified, TLC performs a fresh execution of the model.

`-seed` ***num***
: Provide the seed for the pseudo-random number generator used for random simulation. Defaults to a randomly chosen seed if not specified.

`-simulate` ***dir***
: The *dir* parameter is optional. This tells TLC to run in simulation mode. If *dir* is present, then TLC writes each trace it finds to a separate file in directory *dir*.

`-terse`
: Tells TLA not to expand values in the output produced by *Print* and *PrintT*. If not specified, the values are expanded.

`-tool`
: Tells TLC to print its output in a format to be read by a program such as the Toolbox.

`-userFile` ***file***
: It tells TLC to write output produced by the *Print* and *PrintT* operators in *file*.

`-view`
: If the configuration file specifies a `VIEW`, then this option tells TLC to apply that view to the states when printing an error trace.

`-workers` ***num***
: Specifies the number of TLC worker threads. Without this option, TLC uses only a single worker thread.

3 TLATEX

3.1 Bugs

There are some bugs in TLATEX that cause an occasional misalignment in the output. TLATEX also doesn't do a good job of formatting CASE statements. We would appreciate suggestions for how CASE statements *should* be formatted.

3.2 Inserting TLA$^+$ in a LATEX Document

There is a version of TLATEX for typesetting pieces of TLA$^+$ specifications in a LATEX document. In the `.tex` file, you put

```
\begin{tla}
```
An arbitrary portion of a TLA$^+$ specification
```
\end{tla}
```

Running TLATEX on the file inserts a `tlatex` environment immediately after this `tla` environment that contains the typeset version of that portion of the TLA$^+$ specification, replacing any previous version of the `tlatex` environment.

There are analogous LATEX `pcal` and `ppcal` environments for formatting PlusCal code. The `pcal` environment is for code written in PlusCal's C-syntax; the `ppcal` environment is for P-syntax code.

You run this version of TLATEX with the command "`java tlatex.TeX`". Executing

```
java tlatex.TeX -info
```

will type out reasonably detailed directions on using the program.

4 PlusCal

The PlusCal manual describes the current release of the PlusCal translator. It can be found here.