

Current Versions of the TLA⁺ Tools

Leslie Lamport

7 April 2012

This document describes differences between the descriptions of the TLA⁺ tools in the book *Specifying Concurrent Systems* and the currently released versions. References are to the version of the book currently available on the web. The current versions of the tools are:

TLC Version 2.05 of 7 April 2012
SANY Version 2.2 of 20 July 2011
PlusCal Version 1.8 of 30 March 2012
TLAT_EX Version 0.9 of 19 September 2007

Contents

1	SANY (The Semantic Analyzer)	1
2	TLC	1
2.1	Limitations	1
2.2	Bugs	1
2.3	Additional Features	2
2.3.1	Enhanced Replacement	2
2.3.2	Strings	2
2.3.3	New Options	2
2.3.4	New Features in the TLC Module	3
2.3.5	Typed Model Values	6
3	TLAT_EX	7
3.1	Bugs	7
3.2	Inserting TLA ⁺ in a L _A T _E X Document	7
4	PlusCal	8

1 SANY (The Semantic Analyzer)

The current release of SANY has no known limitations. However, the constructs for writing proofs will be modified to handle temporal-logic proofs.

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 verify a property 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 Bugs

- Under some circumstances, when TLC finds an error in a liveness property, it reports a bogus counterexample.
- TLC does not properly handle certain recursive function definitions that there's no reason to write—namely, ones that apparently perform “recursive priming”. For example, consider

```

VARIABLE x
foo[n ∈ Nat] ≜ IF n = 0 THEN x ELSE foo[n - 1]'

```

If you examine the semantics carefully, you'll discover that this defines *foo* so $foo[n] = x$ for all n in *Nat*. However, TLC computes $foo[n]$ to be x' for $n > 0$.

2.3 Additional Features

2.3.1 Enhanced Replacement

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.

2.3.2 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.)

2.3.3 New Options

TLC currently provides the following options that are not described in the book. These options may not be supported in future releases. If you find any of them useful and want them to continue to be supported, let us know.

-metadir *dir*

Specifies the directory *dir* in which TLC is to store its files, such as the ones that hold the queue of unexamined reachable states. The default directory is `./states`, where “.” is the current directory in which TLC is run.

-dump *state_file*

Causes TLC to write all the reachable states that it finds into file *state_file*.

-continue

Causes TLC to continue running after it finds an error, printing a new message for each error it finds.

-fp *num*

Causes TLC to choose seed number *num* for its fingerprint algorithm, where *num* is an integer from 0 (the default) through 130. As explained in Section 14.3.3, TLC can fail to check all reachable states, and thus fail to find an error, because of a fingerprint collision—that is, because two different states have the same fingerprint. If you run TLC on the same specification using two different seeds, and both runs report no error and find the same number of states, then it’s enormously unlikely that a collision has occurred.

-maxSetSize *num*

The cardinality of the largest set that TLC can enumerate. The default is 1000000 (one million). TLC enumerates sets when computing the set of all initial states. If you are trying to run a spec with lots of initial states, you may have to increase the value of this parameter.

2.3.4 New Features in the TLC Module

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:

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

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

$$\begin{aligned} & \wedge TLCSet(42, \langle \text{"a"}, 1 \rangle) \\ & \wedge \forall i \in \{1, 2, 3\} : \wedge Print(TLCGet(42), TRUE) \\ & \quad \wedge TLCSet(42, [TLCGet(42) \text{ EXCEPT } ![2] = @ + 1]) \end{aligned}$$

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 1000th 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

```
IF TLCSet(i, TLCGet(i) + 1) THEN e 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 *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.

TLC Eval

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* ∈ *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* ∈ *Any* during its execution. However, unrestricted use of *Any* can get TLC to verify incorrect modules. For example, it will evaluate *Any* ∈ *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

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

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, 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

TLC 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 (*o*) in the argument of the *Print* or *PrintT*, you can get TLC to print nicer-looking output than it ordinarily does.

2.3.5 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 “ $\tau_$ ” 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 .

3 TLAT_EX

3.1 Bugs

There are some bugs in TLAT_EX that cause an occasional misalignment in the output. TLAT_EX 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 L^AT_EX Document

There is a version of TLAT_EX for typesetting pieces of TLA⁺ specifications in a L^AT_EX document. In the `.tex` file, you put

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

Running TLAT_EX 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.

You run this version of TLAT_EX 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.