Chapter 14

The TLC Model Checker

TLC is a program for finding errors in TLA$^+$ specifications. It was designed and implemented by Yuan Yu, with help from Leslie Lamport, Mark Hayden, and Mark Tuttle. It is available through the TLA Web page. This chapter describes TLC Version 2. At the time I am writing this, Version 2 is still being implemented and only Version 1 is available. Consult the documentation that accompanies the software to find out what version it is and how it differs from the version described here.

14.1 Introduction to TLC

TLC handles specifications that have the standard form

(14.1) $Init \land \Box[Next]_{vars} \land Temporal$

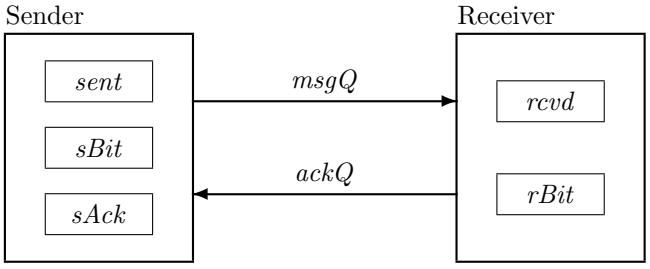
where *Init* is the initial predicate, *Next* is the next-state action, *vars* is the tuple of all variables, and *Temporal* is a temporal formula that usually specifies a liveness condition. Liveness and temporal formulas are explained in Chapter 8. If your specification contains no *Temporal* formula, so it has the form $Init \land \Box[Next]_{vars}$, then you can ignore the discussion of temporal checking. TLC does not handle the hiding operator $\exists$ (temporal existential quantification). You can check a specification with hidden variables by checking the internal specification, in which those variables are visible.

The most effective way to find errors in a specification is by trying to verify that it satisfies properties that it should. TLC can check that the specification satisfies (implies) a large class of TLA formulas—a class whose main restriction is that formulas may not contain $\exists$. You can also run TLC without having it check any property, in which case it will just look for two kinds of errors:

- "Silliness" errors. As explained in Section 6.2, a silly expression is one like $3 + \langle 1, 2 \rangle$, whose meaning is not determined by the semantics of TLA$^+$. A specification is incorrect if whether or not some particular behavior satisfies it depends on the meaning of a silly expression.

- Deadlock. The absence of deadlock is a particular property that we often want a specification to satisfy; it is expressed by the invariance property $\Box$(ENABLED *Next*). A counterexample to this property is a behavior exhibiting deadlock—that is, reaching a state in which *Next* is not enabled, so no further (nonstuttering) step is possible. TLC normally checks for deadlock, but this checking can be disabled since, for some systems, deadlock may just indicate successful termination.

The use of TLC will be illustrated with a simple example—a specification of the alternating bit protocol for sending data over a lossy FIFO transmission line. An algorithm designer might describe the protocol as a system that looks like this:



The sender can send a value when the one-bit values *sBit* and *sAck* are equal. It sets the variables *sent* to the value it is sending and complements *sBit*. This value is eventually delivered to the receiver by setting the variable *rcvd* and complementing the one-bit value *rBit*. Some time later, the sender's *sAck* value is complemented, permitting the next value to be sent. The protocol uses two lossy FIFO transmission lines: the sender sends data and control information on *msgQ*, and the receiver sends acknowledgments on *ackQ*.

The complete protocol specification appears in module *AlternatingBit* in Figure 14.1 on the following two pages. It is fairly straightforward, except for the liveness condition. Because messages can be repeatedly lost from the queues, strong fairness of the actions that receive messages is required to ensure that a message that keeps getting resent is eventually received. However, don't worry about the details of the specification. For now, all you need to know are the declarations

CONSTANT *Data* The set of data values that can be sent.
VARIABLES *msgQ*, *ackQ*, *sBit*, *sAck*, *rBit*, *sent*, *rcvd*

and the types of the variables:

MODULE *AlternatingBit*

This specification describes a protocol for using lossy FIFO transmission lines to transmit a sequence of values from a sender to a receiver. The sender sends a data value d by sending a sequence of $\langle b, d \rangle$ messages on $msgQ$, where b is a control bit. It knows that the message has been received when it receives the ack b from the receiver on $ackQ$. It sends the next value with a different control bit. The receiver knows that a message on $msgQ$ contains a new value when the control bit differs from the last one it has received. The receiver keeps sending the last control bit it received on $ackQ$.

EXTENDS *Naturals, Sequences*

CONSTANTS *Data* The set of data values that can be sent.

VARIABLES
- $msgQ$, The sequence of ⟨control bit, data value⟩ messages in transit to the receiver.
- $ackQ$, The sequence of one-bit acknowledgments in transit to the sender.
- $sBit$, The last control bit sent by sender; it is complemented when sending a new data value.
- $sAck$, The last acknowledgment bit received by the sender.
- $rBit$, The last control bit received by the receiver.
- $sent$, The last value sent by the sender.
- $rcvd$, The last value received by the receiver.

$$
\begin{array}{lll}
ABInit \triangleq & \land\ msgQ = \langle\,\rangle & \text{The initial condition:} \\
& \land\ ackQ = \langle\,\rangle & \text{Both message queues are empty.} \\
& \land\ sBit \in \{0,1\} & \text{All the bits equal 0 or 1} \\
& \land\ sAck = sBit & \text{and are equal to each other.} \\
& \land\ rBit = sBit & \\
& \land\ sent \in Data & \text{The initial values of } sent \text{ and } rcvd \\
& \land\ rcvd \in Data & \text{are arbitrary data values.}
\end{array}
$$

$$
\begin{array}{lll}
ABTypeInv \triangleq & \land\ msgQ \in Seq(\{0,1\} \times Data) & \text{The type-correctness invariant.} \\
& \land\ ackQ \in Seq(\{0,1\}) & \\
& \land\ sBit \in \{0,1\} & \\
& \land\ sAck \in \{0,1\} & \\
& \land\ rBit \in \{0,1\} & \\
& \land\ sent \in Data & \\
& \land\ rcvd \in Data &
\end{array}
$$

$SndNewValue(d) \triangleq$ The action in which the sender sends a new data value d.

$$
\begin{array}{ll}
\land\ sAck = sBit & \text{Enabled iff } sAck \text{ equals } sBit. \\
\land\ sent' = d & \text{Set } sent \text{ to } d. \\
\land\ sBit' = 1 - sBit & \text{Complement control bit } sBit \\
\land\ msgQ' = Append(msgQ, \langle sBit', d \rangle) & \text{Send value on } msgQ \text{ with new control bit.} \\
\land\ \text{UNCHANGED } \langle ackQ, sAck, rBit, rcvd \rangle &
\end{array}
$$

Figure 14.1a: The alternating bit protocol (beginning).

$ReSndMsg \triangleq$ The sender resends the last message it sent on $msgQ$.
- $\land\ sAck \neq sBit$ — Enabled iff $sAck$ doesn't equal $sBit$.
- $\land\ msgQ' = Append(msgQ, \langle sBit, sent\rangle)$ — Resend the last value in $send$.
- $\land$ UNCHANGED $\langle ackQ, sBit, sAck, rBit, sent, rcvd\rangle$

$RcvMsg \triangleq$ The receiver receives the message at the head of $msgQ$.
- $\land\ msgQ \neq \langle\rangle$ — Enabled iff $msgQ$ not empty.
- $\land\ msgQ' = Tail(msgQ)$ — Remove message from head of $msgQ$.
- $\land\ rBit' = Head(msgQ)[1]$ — Set $rBit$ to message's control bit.
- $\land\ rcvd' = Head(msgQ)[2]$ — Set $rcvd$ to message's data value.
- $\land$ UNCHANGED $\langle ackQ, sBit, sAck, sent\rangle$

$SndAck \triangleq \land\ ackQ' = Append(ackQ, rBit)$ The receiver sends $rBit$ on $ackQ$ at any time.
$\land$ UNCHANGED $\langle msgQ, sBit, sAck, rBit, sent, rcvd\rangle$

$RcvAck \triangleq$ The sender receives an ack on $ackQ$. It removes the ack and sets $sAck$ to its value.
- $\land\ ackQ \neq \langle\rangle$
- $\land\ ackQ' = Tail(ackQ)$
- $\land\ sAck' = Head(ackQ)$
- $\land$ UNCHANGED $\langle msgQ, sBit, rBit, sent, rcvd\rangle$

$Lose(q) \triangleq$ The action of losing a message from queue q.
- $\land\ q \neq \langle\rangle$ — Enabled iff q is not empty.
- $\land\ \exists\, i \in 1\,..\,Len(q)$: — For some i,
- $q' = [j \in 1\,..\,(Len(q)-1) \mapsto$ IF $j < i$ THEN $q[j]$ ELSE $q[j+1]]$ — remove the i^{th} message from q.
- $\land$ UNCHANGED $\langle sBit, sAck, rBit, sent, rcvd\rangle$ — Leave every variable unchanged except $msgQ$ and $ackQ$.

$LoseMsg \triangleq Lose(msgQ) \land$ UNCHANGED $ackQ$ Lose a message from $msgQ$.

$LoseAck \triangleq Lose(ackQ) \land$ UNCHANGED $msgQ$ Lose a message from $ackQ$.

$ABNext \triangleq$ The next-state action.
- $\lor\ \exists\, d \in Data : SndNewValue(d)$
- $\lor\ ReSndMsg \lor RcvMsg \lor SndAck \lor RcvAck$
- $\lor\ LoseMsg \lor LoseAck$

$abvars \triangleq \langle msgQ, ackQ, sBit, sAck, rBit, sent, rcvd\rangle$ The tuple of all variables.

$ABFairness \triangleq$ The liveness condition.
- $\land\ \mathrm{WF}_{abvars}(ReSndMsg) \land \mathrm{WF}_{abvars}(SndAck)$
- $\land\ \mathrm{SF}_{abvars}(RcvMsg) \land \mathrm{SF}_{abvars}(RcvAck)$

$ABSpec \triangleq ABInit \land \Box[ABNext]_{abvars} \land ABFairness$ The complete specification.

THEOREM $ABSpec \Rightarrow \Box ABTypeInv$

Figure 14.1b: The alternating bit protocol (end).

- $msgQ$ is a sequence of elements in $\{0, 1\} \times Data$.
- $ackQ$ is a sequence of elements in $\{0, 1\}$.
- $sBit$, $sAck$, and $rBit$ are elements of $\{0, 1\}$.
- $sent$ and $rcvd$ are elements of $Data$.

The input to TLC consists of a TLA$^+$ module and a configuration file. TLC assumes the specification has the form of formula (14.1) on page 221. The configuration file tells TLC the names of the specification and of the properties to be checked. For example, the configuration file for the alternating bit protocol will contain the declaration

```
SPECIFICATION ABSpec
```

telling TLC to take *ABSpec* as the specification. If your specification has the form $Init \land \Box[Next]_{vars}$, with no liveness condition, then instead of using a `SPECIFICATION` statement, you can declare the initial predicate and next-state action by putting the following two statements in the configuration file:

```
INIT Init
NEXT Next
```

The property or properties to be checked are specified with a `PROPERTY` statement. For example, to check that *ABTypeInv* is actually an invariant, we could have TLC check that the specification implies $\Box ABTypeInv$ by adding the definition

$$InvProperty \triangleq \Box ABTypeInv$$

to module *AlternatingBit* and putting the statement

```
PROPERTY InvProperty
```

in the configuration file. Invariance checking is so common that TLC allows you instead to put the following statement in the configuration file:

```
INVARIANT ABTypeInv
```

The `INVARIANT` statement must specify a state predicate. To check invariance with a `PROPERTY` statement, the specified property has to be of the form $\Box P$. Specifying a state predicate P in a `PROPERTY` statement tells TLC to check that the specification implies P, meaning that P is true in the initial state of every behavior satisfying the specification.

TLC works by generating behaviors that satisfy the specification. To do this, it must be given what we call a *model* of the specification. To define a model, we must assign values to the specification's constant parameters. The only constant parameter of the alternating bit protocol specification is the set

Data of data values. We can tell TLC to let *Data* equal the set containing two arbitrary elements, named $d1$ and $d2$, by putting the following declaration in the configuration file:

```
CONSTANT  Data = {d1, d2}
```

The keywords `CONSTANT` and `CONSTANTS` are equivalent, as are `INVARIANT` and `INVARIANTS`.

(We can use any sequence of letters and digits containing at least one letter as the name of an element.)

There are two ways to use TLC. The default method is *model checking*, in which it tries to find all reachable states—that is, all states[1] that can occur in behaviors satisfying the formula $Init \wedge \Box[Next]_{vars}$. You can also run TLC in *simulation* mode, in which it randomly generates behaviors, without trying to check all reachable states. We now consider model checking; simulation mode is described in Section 14.3.2 on page 243.

Exhaustively checking all reachable states is impossible for the alternating bit protocol because the sequences of messages can get arbitrarily long, so there are infinitely many reachable states. We must further constrain the model to make it finite—that is, so it allows only a finite number of possible states. We do this by defining a state predicate called the *constraint* that asserts bounds on the lengths of the sequences. For example, the following constraint asserts that $msgQ$ and $ackQ$ have length at most 2:

Section 14.3 below describes how actions as well as state predicates can be used as constraints.

$$\begin{aligned} &\wedge\ Len(msgQ) \leq 2 \\ &\wedge\ Len(ackQ)\ \leq 2 \end{aligned}$$

Instead of specifying the bounds on the lengths of sequences in this way, I prefer to make them parameters and to assign them values in the configuration file. We don't want to put into the specification itself declarations and definitions that are just for TLC's benefit. So, we write a new module, called *MCAlternatingBit*, that extends the *AlternatingBit* module and can be used as input to TLC. This module appears in Figure 14.2 on the next page. A possible configuration file for the module appears in Figure 14.3 on the next page. Observe that the configuration file must specify values for all the constant parameters of the specification—in this case, the parameter *Data* from the *AlternatingBit* module and the two parameters declared in module *MCAlternatingBit* itself. You can put comments in the configuration file, using the TLA$^+$ comment syntax described in Section 3.5 (page 32).

When a constraint *Constr* is specified, TLC checks every state that appears in a behavior satisfying $Init \wedge \Box[Next]_{vars} \wedge \Box Constr$. In the rest of this chapter, these states will be called the *reachable* ones.

?

[1] As explained in Section 2.3 (page 18), a state is an assignment of values to all possible variables. However, when discussing a particular specification, we usually consider a state to be an assignment of values to that specification's variables. That's what I'm doing in this chapter.

MODULE *MCAlternatingBit*

EXTENDS *AlternatingBit*

CONSTANTS *msgQLen, ackQLen*

$$SeqConstraint \triangleq \begin{array}{ll} & Len(msgQ) \leq msgQLen \\ \wedge & Len(ackQ) \leq ackQLen \end{array}$$

A constraint on the lengths of sequences for use by TLC.

Figure 14.2: Module *MCAlternatingBit*.

Having TLC check the type invariant will catch many simple mistakes. When we've corrected all the errors we can find that way, we then want to look for less obvious ones. A common error is for an action not to be enabled when it should be, preventing some states from being reached. You can discover if an action is never enabled by using the *coverage* option, described on page 252. To discover if an action is just sometimes incorrectly disabled, try checking liveness properties. An obvious liveness property for the alternating bit protocol is that every message sent is eventually delivered. A message d has been sent when $sent = d$ and $sBit \neq sAck$. So, a naive way to state this property is

The temporal operator $\rightsquigarrow$ is defined on page 91.

$$SentLeadsToRcvd \triangleq \forall\, d \in Data : (sent = d) \wedge (sBit \neq sAck) \rightsquigarrow (rcvd = d)$$

Formula *SentLeadsToRcvd* asserts that, for any data value d, if *sent* ever equals d when *sBit* does not equal *sAck*, then *rcvd* must eventually equal d. This doesn't assert that every message sent is eventually delivered. For example, it is satisfied by a behavior in which a particular value d is sent twice, but received only once. However, the formula is good enough for our purposes because the protocol doesn't depend on the actual values being sent. If it were possible for the same value to be sent twice but received only once, then it would be possible for two different values to be sent and only one received, violating *SentLeadsToRcvd*. We therefore add the definition of *SentLeadsToRcvd* to module *MCAlternatingBit* and add the following statement to the configuration file:

```
    PROPERTY SentLeadsToRcvd

CONSTANTS     Data    = {d1, d2}   (* Is this big enough?  *)
              msgQLen = 2
              ackQLen = 2   \* Try 3 next.
SPECIFICATION ABSpec
INVARIANT     ABTypeInv
CONSTRAINT    SeqConstraint
```

Figure 14.3: A configuration file for module *MCAlternatingBit*.

Checking liveness properties is a lot slower than other kinds of checking, so you should do it only after you've found all the errors you can by checking invariance properties.

Checking type correctness and property *SentLeadsToRcvd* is a good way to start looking for errors. But ultimately, we would like to see if the protocol meets its specification. However, we don't have its specification. In fact, it is typical in practice that we are called upon to check the correctness of a system design without any formal specification of what the system is supposed to do. In that case, we can write an *ex post facto* specification. Module *ABCorrectness* in Figure 14.4 on the next page is such a specification of correctness for the alternating bit protocol. It is actually a simplified version of the protocol's specification in which, instead of being read from messages, the variables *rcvd*, *rBit*, and *sAck* are obtained directly from the variables of the other process.

We want to check that the specification *ABSpec* of module *AlternatingBit* implies formula *ABCSpec* of module *ABCorrectness*. To do this, we modify module *MCAlternatingBit* by adding the statement

INSTANCE *ABCorrectness*

and we modify the `PROPERTY` statement of the configuration file to

```
PROPERTIES ABCSpec SentLeadsToRcvd
```

The keywords `PROPERTY` and `PROPERTIES` are equivalent.

This example is atypical because the correctness specification *ABCSpec* does not involve variable hiding (temporal existential quantification). Let's now suppose module *ABCorrectness* did declare another variable h that appeared in *ABCSpec*, and that the correctness condition for the alternating bit protocol was *ABCSpec* with h hidden. The correctness condition would then be expressed formally in TLA$^+$ as follows:

$AB(h) \triangleq$ INSTANCE *ABCorrectness*
THEOREM $ABSpec \Rightarrow \exists\, h : AB(h)!ABCSpec$

This use of INSTANCE is explained in Section 4.3 (page 41).

TLC could not check this theorem directly because it cannot handle the temporal existential quantifier $\exists$. We would check this theorem with TLC the same way we would try to prove it—namely, by using a refinement mapping. As explained in Section 5.8 on page 62, we would define a state function *oh* in terms of the variables of module *AlternatingBit* and we would prove

(14.2) $ABSpec \Rightarrow AB(oh)!ABCSpec$

To get TLC to check this theorem, we would add the definition

$ABCSpecBar \triangleq AB(oh)!ABCSpec$

and have TLC check the property *ABCSpecBar*.

MODULE *ABCorrectness*

EXTENDS *Naturals*

CONSTANTS *Data*

VARIABLES *sBit*, *sAck*, *rBit*, *sent*, *rcvd*

$$
\begin{aligned}
ABCInit &\triangleq \land\ sBit \in \{0, 1\} \\
&\quad\ \land\ sAck = sBit \\
&\quad\ \land\ rBit = sBit \\
&\quad\ \land\ sent \in Data \\
&\quad\ \land\ rcvd \in Data \\
\\
CSndNewValue(d) &\triangleq \land\ sAck = sBit \\
&\quad\ \land\ sent' = d \\
&\quad\ \land\ sBit' = 1 - sBit \\
&\quad\ \land \text{UNCHANGED } \langle sAck, rBit, rcvd \rangle \\
\\
CRcvMsg &\triangleq \land\ rBit \neq sBit \\
&\quad\ \land\ rBit' = sBit \\
&\quad\ \land\ rcvd' = sent \\
&\quad\ \land \text{UNCHANGED } \langle sBit, sAck, sent \rangle \\
\\
CRcvAck &\triangleq \land\ rBit \neq sAck \\
&\quad\ \land\ sAck' = rBit \\
&\quad\ \land \text{UNCHANGED } \langle sBit, rBit, sent, rcvd \rangle \\
\\
ABCNext &\triangleq \lor \exists\, d \in Data : CSndNewValue(d) \\
&\quad\ \lor CRcvMsg \lor CRcvAck
\end{aligned}
$$

$$
\begin{aligned}
cvars &\triangleq \langle sBit, sAck, rBit, sent, rcvd \rangle \\
\\
ABCFairness &\triangleq \text{WF}_{cvars}(CRcvMsg) \land \text{WF}_{cvars}(CRcvAck) \\
\\
ABCSpec &\triangleq ABCInit \land \Box[ABCNext]_{cvars} \land ABCFairness
\end{aligned}
$$

Figure 14.4: A specification of correctness of the alternating bit protocol.

When TLC checks a property, it does not actually verify that the specification implies the property. Instead, it checks that (i) the safety part of the specification implies the safety part of the property and (ii) the specification implies the liveness part of the property. For example, suppose that the specification *Spec* and the property *Prop* are

$$
\begin{aligned}
Spec &\triangleq Init \land \Box[Next]_{vars} \land Temporal \\
Prop &\triangleq ImpliedInit \land \Box[ImpliedAction]_{pvars} \land ImpliedTemporal
\end{aligned}
$$

where *Temporal* and *ImpliedTemporal* are liveness properties. In this case, TLC checks the two formulas

$$Init \wedge \Box[Next]_{vars} \Rightarrow ImpliedInit \wedge \Box[ImpliedAction]_{pvars}$$
$$Spec \Rightarrow ImpliedTemporal$$

This means that you cannot use TLC to check that a non-machine-closed specification satisfies a safety property. (Machine closure is discussed in Section 8.9.2 on page 111.) Section 14.3 below more precisely describes how TLC checks properties.

14.2 What TLC Can Cope With

No model checker can handle all the specifications that we can write in a language as expressive as TLA$^+$. However, TLC seems able to handle most TLA$^+$ specifications that people actually write. Getting TLC to handle a specification may require a bit of trickery, but it can usually be done without having to make any changes to the specification itself.

This section explains what TLC can and cannot cope with, and gives some ways to make it cope. The best way to understand TLC's limitations is to understand how it works. So, this section describes how TLC "executes" a specification.

14.2.1 TLC Values

A state is an assignment of values to variables. TLA$^+$ allows you to describe a wide variety of values—for example, the set of all sequences of prime numbers. TLC can compute only a restricted class of values, called TLC values. Those values are built from the following four types of primitive values:

Booleans	The values TRUE and FALSE.
Integers	Values like 3 and −1.
Strings	Values like "ab3".
Model Values	These are values introduced in the `CONSTANT` statement of the configuration file. For example, the configuration file shown in Figure 14.3 on page 227 introduces the model values `d1` and `d2`. Model values with different names are assumed to be different.

A TLC value is defined inductively to be either

1. a primitive value, or

2. a finite set of comparable TLC values (*comparable* is defined below), or

3. a function f whose domain is a TLC value such that $f[x]$ is a TLC value, for all x in DOMAIN f.

For example, the first two rules imply that

(14.3) {{"a", "b"}, {"b", "c"}, {"c", "d"}}

is a TLC value because rules 1 and 2 imply that {"a", "b"}, {"b", "c"}, and {"c", "d"} are TLC values, and the second rule then implies that (14.3) is a TLC value. Since tuples and records are functions, rule 3 implies that a record or tuple whose components are TLC values is a TLC value. For example, $\langle 1, \text{"a"}, 2, \text{"b"} \rangle$ is a TLC value.

To complete the definition of what a TLC value is, I must explain what *comparable* means in rule 2. The basic idea is that two values should be comparable iff the semantics of TLA$^+$ determines whether or not they are equal. For example, strings and numbers are not comparable because the semantics of TLA$^+$ doesn't tell us whether or not "abc" equals 42. The set {"abc", 42} is therefore not a TLC value; rule 2 doesn't apply because "abc" and 42 are not comparable. On the other hand, {"abc"} and $\{4, 2\}$ are comparable because sets having different numbers of elements must be unequal. Hence, the two-element set {{"abc"}, $\{4, 2\}$} is a TLC value. TLC considers a model value to be comparable to, and unequal to, any other value. The precise rules for comparability are given in Section 14.7.2.

14.2.2 How TLC Evaluates Expressions

Checking a specification requires evaluating expressions. For example, TLC does invariance checking by evaluating the invariant in each reachable state—that is, computing its TLC value, which should be TRUE. To understand what TLC can and cannot do, you have to know how it evaluates expressions.

TLC evaluates expressions in a straightforward way, generally evaluating subexpressions "from left to right". In particular:

- It evaluates $p \land q$ by first evaluating p and, if it equals TRUE, then evaluating q.

- It evaluates $p \lor q$ by first evaluating p and, if it equals FALSE, then evaluating q. It evaluates $p \Rightarrow q$ as $\neg p \lor q$.

- It evaluates IF p THEN e_1 ELSE e_2 by first evaluating p, then evaluating either e_1 or e_2.

To understand the significance of these rules, let's consider a simple example. TLC cannot evaluate the expression $x[1]$ if x equals $\langle\rangle$, since $\langle\rangle[1]$ is silly. (The

empty sequence $\langle\rangle$ is a function whose domain is the empty set and hence does not contain 1.) The first rule implies that, if x equals $\langle\rangle$, then TLC can evaluate the formula

$$(x \neq \langle\rangle) \land (x[1] = 0)$$

but not the (logically equivalent) formula

$$(x[1] = 0) \land (x \neq \langle\rangle)$$

(When evaluating the latter formula, TLC first tries to compute $\langle\rangle[1] = 0$, reporting an error because it can't.) Fortunately, we naturally write the first formula rather than the second because it's easier to understand. People understand a formula by "mentally evaluating" it from left to right, much the way TLC does.

TLC evaluates $\exists\, x \in S : p$ by enumerating the elements $s_1, \ldots, s_n$ of S in some order and then evaluating p with s_i substituted for x, successively for $i = 1, \ldots, n$. It enumerates the elements of a set S in a very straightforward way, and it gives up and declares an error if the set is not obviously finite. For example, it can obviously enumerate the elements of $\{0, 1, 2, 3\}$ and $0 \,..\, 3$. It enumerates a set of the form $\{x \in S : p\}$ by first enumerating S, so it can enumerate $\{i \in 0 \,..\, 5 : i < 4\}$ but not $\{i \in Nat : i < 4\}$.

???

TLC evaluates the expressions $\forall\, x \in S : p$ and CHOOSE $x \in S : p$ by first enumerating the elements of S, much the same way as it evaluates $\exists\, x \in S : p$. The semantics of TLA$^+$ states that CHOOSE $x \in S : p$ is an arbitrary value if there is no x in S for which p is true. However, this case almost always arises because of a mistake, so TLC treats it as an error. Note that evaluating the expression

$$\text{IF } n > 5 \text{ THEN CHOOSE } i \in 1 \,..\, n : i > 5 \text{ ELSE } 42$$

will not produce an error because TLC will not evaluate the CHOOSE expression if $n \leq 5$. (TLC would report an error if it tried to evaluate the CHOOSE expression when $n \leq 5$.)

TLC cannot evaluate "unbounded" quantifiers or CHOOSE expressions—that is, expressions having one of the forms

$$\exists\, x : p \qquad \forall\, x : p \qquad \text{CHOOSE } x : p$$

TLC cannot evaluate any expression whose value is not a TLC value, as defined in Section 14.2.1 above. In particular, TLC can evaluate a set-valued expression only if that expression equals a finite set, and it can evaluate a function-valued expression only if that expression equals a function whose domain is a finite set. TLC will evaluate expressions of the following forms only if it can enumerate the set S:

$\exists\, x \in S : p$	$\forall\, x \in S : p$	CHOOSE $x \in S : p$
$\{x \in S : p\}$	$\{e : x \in S\}$	$[x \in S \mapsto e]$
SUBSET S	UNION S	

TLC can often evaluate an expression even when it can't evaluate all subexpressions. For example, it can evaluate

$$[n \in Nat \mapsto n * (n + 1)][3]$$

which equals the TLC value 12, even though it can't evaluate

$$[n \in Nat \mapsto n * (n + 1)]$$

which equals a function whose domain is the set Nat. (A function can be a TLC value only if its domain is a finite set.)

TLC evaluates recursively defined functions with a simple recursive procedure. If f is defined by $f[x \in S] \triangleq e$, then TLC evaluates $f[c]$ by evaluating e with c substituted for x. This means that it can't handle some legal function definitions. For example, consider this definition from page 68:

$$\begin{array}{l} mr[n \in Nat] \triangleq \\ \quad [f \mapsto \text{IF } n = 0 \text{ THEN } 17 \text{ ELSE } mr[n-1].f * mr[n].g\,, \\ \quad\ g \mapsto \text{IF } n = 0 \text{ THEN } 42 \text{ ELSE } mr[n-1].f + mr[n-1].g\,] \end{array}$$

To evaluate $mr[3]$, TLC substitutes 3 for n and starts evaluating the right-hand side. But because $mr[n]$ appears in the right-hand side, TLC must evaluate the subexpression $mr[3]$, which it does by substituting 3 for n and starting to evaluate the right-hand side. And so on. TLC eventually detects that it's in an infinite loop and reports an error.

Legal recursive definitions that cause TLC to loop like this are rare, and they can be rewritten so TLC can handle them. Recall that we defined mr to express the mutual recursion:

$$\begin{array}{l} f[n] = \text{IF } n = 0 \text{ THEN } 17 \text{ ELSE } f[n-1] * g[n] \\ g[n] = \text{IF } n = 0 \text{ THEN } 42 \text{ ELSE } f[n-1] + g[n-1] \end{array}$$

The subexpression $mr[n]$ appeared in the expression defining $mr[n]$ because $f[n]$ depends on $g[n]$. To eliminate it, we have to rewrite the mutual recursion so that $f[n]$ depends only on $f[n-1]$ and $g[n-1]$. We do this by expanding the definition of $g[n]$ in the expression for $f[n]$. Since the ELSE clause applies only to the case $n \neq 0$, we can rewrite the expression for $f[n]$ as

$$f[n] = \text{IF } n = 0 \text{ THEN } 17 \text{ ELSE } f[n-1] * (f[n-1] + g[n-1])$$

This leads to the following equivalent definition of mr:

$$\begin{array}{l} mr[n \in Nat] \triangleq \\ \quad [f \mapsto \text{IF } n = 0 \text{ THEN } 17 \\ \qquad\qquad\qquad \text{ELSE } mr[n-1].f * (mr[n-1].f + mr[n-1].g)\,, \\ \quad\ g \mapsto \text{IF } n = 0 \text{ THEN } 42 \text{ ELSE } mr[n-1].f + mr[n-1].g\,] \end{array}$$

With this definition, TLC has no trouble evaluating $mr[3]$.

The evaluation of ENABLED predicates and the action-composition operator "$\cdot$" are described on page 240 in Section 14.2.6. Section 14.3 explains how TLC evaluates temporal-logic formulas for temporal checking.

If you're not sure whether TLC can evaluate an expression, try it and see. But don't wait until TLC gets to the expression in the middle of checking the entire specification. Instead, make a small example in which TLC evaluates just that expression. See the explanation on page 14.5.3 of how to use TLC as a TLA$^+$ calculator.

14.2.3 Assignment and Replacement

As we saw in the alternating bit example, the configuration file must determine the value of each constant parameter. To assign a TLC value v to a constant parameter c of the specification, we write `c = v` in the configuration file's `CONSTANT` statement. The value v may be a primitive TLC value or a finite set of primitive TLC values written in the form $\{v_1, \ldots, v_n\}$—for example, `{1, -3, 2}`. In v, any sequence of characters like `a1` or `foo` that is not a number, a quoted string, or `TRUE` or `FALSE` is taken to be a model value.

In the assignment `c = v`, the symbol c need not be a constant parameter; it can also be a defined symbol. This assignment causes TLC to ignore the actual definition of c and to take v to be its value. Such an assignment is often used when TLC cannot compute the value of c from its definition. In particular, TLC cannot compute the value of *NotAnS* from the definition

$$NotAnS \triangleq \text{CHOOSE } n : n \notin S$$

because it cannot evaluate the unbounded CHOOSE expression. You can override this definition by assigning *NotAnS* a value in the `CONSTANT` statement of the configuration file. For example, the assignment

```
NotAnS = NS
```

causes TLC to assign to *NotAnS* the model value `NS`. TLC ignores the actual definition of *NotAnS*. If you used the name *NotAnS* in the specification, you'd probably want TLC's error messages to call it `NotAnS` rather than `NS`. So, you'd probably use the assignment

```
NotAnS = NotAnS
```

which assigns to the symbol *NotAnS* the model value `NotAnS`. Remember that, in the assignment `c = v`, the symbol c must be defined or declared in the TLA$^+$ module, and v must be a primitive TLC value or a finite set of such values.

Note that d is a defined symbol in the replacement `c <- d`, while `v` is a TLC value in the substitution `c = v`.

The `CONSTANT` statement of the configuration file can also contain *replacements* of the form `c <- d`, where c and d are symbols defined in the TLA$^+$

module. This causes TLC to replace c by d when performing its calculations. One use of replacement is to give a value to an operator parameter. For example, suppose we wanted to use TLC to check the write-through cache specification of Section 5.6 (page 54). The *WriteThroughCache* module extends the *MemoryInterface* module, which contains the declaration

CONSTANTS $Send(\_, \_, \_, \_)$, $Reply(\_, \_, \_, \_)$, ...

We have to tell TLC how to evaluate the operators *Send* and *Reply*. We do this by first writing a module *MCWriteThroughCache* that extends the *WriteThroughCache* module and defines two operators

$$MCSend(p, d, old, new) \triangleq \ldots$$
$$MCReply(p, d, old, new) \triangleq \ldots$$

We then add to the configuration file's `CONSTANT` statement the replacements

```
Send  <- MCSend
Reply <- MCReply
```

A replacement can also replace one defined symbol by another. In a specification, we usually write the simplest possible definitions. A simple definition is not always the easiest one for TLC to use. For example, suppose our specification requires an operator *Sort* such that $Sort(S)$ is a sequence containing the elements of S in increasing order, if S is a finite set of numbers. Our specification in module *SpecMod* might use the simple definition

$$Sort(S) \triangleq \text{CHOOSE } s \in [1 \,..\, Cardinality(S) \to S] : \\ \forall\, i, j \in \text{DOMAIN } s : (i < j) \Rightarrow (s[i] < s[j])$$

To evaluate $Sort(S)$ for a set S containing n elements, TLC has to enumerate the n^n elements in the set $[1 \,..\, n \to S]$ of functions. This may be unacceptably slow. We can write a module *MCSpecMod* that extends *SpecMod* and defines *FastSort* so it equals *Sort* when applied to finite sets of numbers, but can be evaluated more efficiently by TLC. We can then run TLC with a configuration file containing the replacement

```
Sort <- FastSort
```

One possible definition of *FastSort* is given in Section 14.4, on page 250.

14.2.4 Evaluating Temporal Formulas

Section 14.2.2 (page 231) explains what kind of ordinary expressions TLC can evaluate. The specification and properties that TLC checks are temporal formulas; this section describes the class of temporal formulas it can handle.

TLC can evaluate a TLA temporal formula iff (i) the formula is *nice*—a term defined in the next paragraph—and (ii) TLC can evaluate all the ordinary expressions of which the formula is composed. For example, a formula of the form $P \leadsto Q$ is nice, so TLC can evaluate it iff it can evaluate P and Q. (Section 14.3 below explains on what states and pairs of states TLC evaluates the component expressions of a temporal formula.)

A temporal formula is nice iff it is the conjunction of formulas that belong to one of the following four classes:

State Predicate

Invariance Formula A formula of the form $\Box P$, where P is a state predicate.

Box-Action Formula A formula of the form $\Box[A]_v$, where A is an action and v is a state function.

Simple Temporal Formula To define this class, we first make the following definitions:

The terminology used here is not standard.

- The *simple Boolean operators* consist of the operators

  $\land \quad \lor \quad \neg \quad \Rightarrow \quad \equiv \quad$ TRUE $\quad$ FALSE

  of propositional logic together with quantification over finite, constant sets.

- A *temporal state formula* is one obtained from state predicates by applying simple Boolean operators and the temporal operators $\Box$, $\Diamond$, and $\leadsto$. For example, if N is a constant, then

  $$\forall\, i \in 1 \,..\, N \,:\, \Box((x = i) \Rightarrow \exists\, j \in 1 \,..\, i \,:\, \Diamond(y = j))$$

  is a temporal state formula.

- A *simple action formula* is one of the following, where A is an action and v a state function:

  $\mathrm{WF}_v(A) \quad \mathrm{SF}_v(A) \quad \Box\Diamond\langle A\rangle_v \quad \Diamond\Box[A]_v$

  The component expressions of $\mathrm{WF}_v(A)$ and $\mathrm{SF}_v(A)$ are $\langle A\rangle_v$ and ENABLED $\langle A\rangle_v$. (The evaluation of ENABLED formulas is described on page 240.)

A simple temporal formula is then defined to be one constructed from temporal state formulas and simple action formulas by applying simple Boolean operators.

For convenience, we exclude invariance formulas from the class of simple temporal formulas, so these four classes of nice temporal formulas are disjoint.

TLC can therefore evaluate the temporal formula

$$\forall\, i \in 1 \,..\, N \,:\, \Diamond(y = i) \Rightarrow \mathrm{WF}_y((y' = y + 1) \land (y \geq i))$$

if N is a constant, because this is a simple temporal formula (and hence nice) and TLC can evaluate all of its component expressions. TLC cannot evaluate $\Diamond\langle x' = 1\rangle_x$, since this is not a nice formula. It cannot evaluate the formula $\mathrm{WF}_x(x'[1] = 0)$ if it must evaluate the action $\langle x'[1] = 0\rangle_x$ on a step $s \to t$ in which $x = \langle\,\rangle$ in state t.

A `PROPERTY` statement can specify any formulas that TLC can evaluate. The formula of a `SPECIFICATION` statement must contain exactly one conjunct that is a box-action formula. That conjunct specifies the next-state action.

14.2.5 Overriding Modules

TLC cannot compute $2 + 2$ from the definition of $+$ contained in the standard *Naturals* module. Even if we did use a definition of $+$ from which TLC could compute sums, it would not do so very quickly. Arithmetic operators like $+$ are implemented directly in Java, the language in which TLC is written. This is achieved by a general mechanism of TLC that allows a module to be overridden by a Java class that implements the operators defined in the module. When TLC encounters an EXTENDS *Naturals* statement, it loads the Java class that overrides the *Naturals* module rather than reading the module itself. There are Java classes to override the following standard modules: *Naturals*, *Integers*, *Sequences*, *FiniteSets*, and *Bags*. (The *TLC* module described below in Section 14.4 is also overridden by a Java class.) Intrepid Java programmers will find that writing a Java class to override a module is not too hard.

14.2.6 How TLC Computes States

When TLC evaluates an invariant, it is calculating the invariant's value, which is either TRUE or FALSE. When TLC evaluates the initial predicate or the next-state action, it is computing a set of states—for the initial predicate, the set of all initial states, and for the next-state action, the set of possible *successor states* (primed states) reached from a given starting (unprimed) state. I will describe how TLC does this for the next-state action; the evaluation of the initial predicate is analogous.

Recall that a state is an assignment of values to variables. TLC computes the successors of a given state s by assigning to all unprimed variables their values in state s, assigning no values to the primed variables, and then evaluating the next-state action. TLC evaluates the next-state action as described in Section 14.2.2 (page 231), except for two differences, which I now describe. This description assumes that TLC has already performed all the assignments and replacements specified by the `CONSTANT` statement of the configuration file and has expanded all definitions. Thus, the next-state action is a formula containing only variables, primed variables, model values, and built-in TLA$^+$ operators and constants.

The first difference in evaluating the next-state action is that TLC does not evaluate disjunctions from left to right. Instead, when it evaluates a subformula $A_1 \lor \ldots \lor A_n$, it splits the computation into n separate evaluations, each taking the subformula to be one of the A_i. Similarly, when it evaluates $\exists\, x \in S : p$, it splits the computation into separate evaluations for each element of S. An implication $P \Rightarrow Q$ is treated as the disjunction $(\neg P) \lor Q$. For example, TLC splits the evaluation of

$$(A \Rightarrow B) \lor (C \land (\exists\, i \in S \,:\, D(i)) \land E)$$

into separate evaluations of the three disjuncts $\neg A$, B, and

$$C \land (\exists\, i \in S \,:\, D(i)) \land E$$

To evaluate the latter disjunct, it first evaluates C. If it obtains the value TRUE, then it splits this evaluation into the separate evaluations of $D(i) \land E$, for each i in S. It evaluates $D(i) \land E$ by first evaluating $D(i)$ and, if it obtains the value TRUE, then evaluating E.

The second difference in the way TLC evaluates the next-state action is that, for any variable x, if it evaluates an expression of the form $x' = e$ when x' has not yet been assigned a value, then the evaluation yields the value TRUE and TLC assigns to x' the value obtained by evaluating the expression e. TLC evaluates an expression of the form $x' \in S$ as if it were $\exists\, v \in S : x' = v$. It evaluates UNCHANGED x as $x' = x$ for any variable x, and UNCHANGED $\langle e_1, \ldots, e_n \rangle$ as

$$(\text{UNCHANGED } e_1) \land \ldots \land (\text{UNCHANGED } e_n)$$

for any expressions e_i. Hence, TLC evaluates UNCHANGED $\langle x, \langle y, z \rangle \rangle$ as if it were

$$(x' = x) \land (y' = y) \land (z' = z)$$

Except when evaluating an expression of the form $x' = e$, TLC reports an error if it encounters a primed variable that has not yet been assigned a value. An evaluation stops, finding no states, if a conjunct evaluates to FALSE. An evaluation that completes and obtains the value TRUE finds the state determined by the values assigned to the primed variables. In the latter case, TLC reports an error if some primed variable has not been assigned a value.

To illustrate how this works, let's consider how TLC evaluates the next-state action

$$\begin{array}{ll} \lor & \land\ x' \in 1 \,..\, Len(y) \\ & \quad \land\ y' = Append(Tail(y), x') \\ \lor & \land\ x' = x + 1 \\ & \quad \land\ y' = Append(y, x') \end{array} \tag{14.4}$$

We first consider the starting state with $x = 1$ and $y = \langle 2, 3 \rangle$. TLC splits the computation into evaluating the two disjuncts separately. It begins evaluating

the first disjunct of (14.4) by evaluating its first conjunct, which it treats as $\exists\, i \in 1 \,..\, Len(y) : x' = i$. Since $Len(y) = 2$, the evaluation splits into separate evaluations of

(14.5)
$$\begin{array}{ll} \land\ x' = 1 & \land\ x' = 2 \\ \land\ y' = Append(Tail(y), x') & \land\ y' = Append(Tail(y), x') \end{array}$$

TLC evaluates the first of these actions as follows. It evaluates the first conjunct, obtaining the value TRUE and assigning to x' the value 1; it then evaluates the second conjunct, obtaining the value TRUE and assigning to y' the value $Append(Tail(\langle 2, 3\rangle), 1)$. So, evaluating the first action of (14.5) finds the successor state with $x = 1$ and $y = \langle 3, 1\rangle$. Similarly, evaluating the second action of (14.5) finds the successor state with $x = 2$ and $y = \langle 3, 2\rangle$. In a similar way, TLC evaluates the second disjunct of (14.4) to find the successor state with $x = 2$ and $y = \langle 2, 3, 2\rangle$. Hence, the evaluation of (14.4) finds three successor states.

Next, consider how TLC evaluates the next-state action (14.4) in a state with $x = 1$ and y equal to the empty sequence $\langle\,\rangle$. Since $Len(y) = 0$ and $1 \,..\, 0$ is the empty set $\{\,\}$, TLC evaluates the first disjunct as

$$\begin{array}{l} \land\ \exists\, i \in \{\,\} : x' = i \\ \land\ y' = Append(Tail(y), x') \end{array}$$

Evaluating the first conjunct yields FALSE, so the evaluation of the first disjunct of (14.4) stops, finding no successor states. Evaluating the second disjunct yields the successor state with $x = 2$ and $y = \langle 2\rangle$.

Since TLC evaluates conjuncts from left to right, their order can affect whether or not TLC can evaluate the next-state action. For example, suppose the two conjuncts in the first disjunct of (14.4) were reversed, like this:

$$\begin{array}{l} \land\ y' = Append(Tail(y), x') \\ \land\ x' \in 1 \,..\, Len(y) \end{array}$$

When TLC evaluates the first conjunct of this action, it encounters the expression $Append(Tail(y), x')$ before it has assigned a value to x', so it reports an error. Moreover, even if we were to change that x' to an x, TLC could still not evaluate the action starting in a state with $y = \langle\,\rangle$, since it would encounter the silly expression $Tail(\langle\,\rangle)$ when evaluating the first conjunct.

The description given above of how TLC evaluates an arbitrary next-state action is good enough to explain how it works in almost all cases that arise in practice. However, it is not completely accurate. For example, interpreted literally, it would imply that TLC can cope with the following two next-state actions, which are both logically equivalent to $(x' = \text{TRUE}) \land (y' = 1)$:

(14.6) $(x' = (y' = 1)) \land (x' = \text{TRUE})$ IF $x' = \text{TRUE}$ THEN $y' = 1$ ELSE FALSE

?????

In fact, TLC will produce error messages when presented with either of these bizarre next-state actions.

Remember that TLC computes initial states by using a similar procedure to evaluate the initial predicate. Instead of starting from given values of the unprimed variables and assigning values to the primed variables, it assigns values to the unprimed variables.

TLC evaluates ENABLED formulas essentially the same way it evaluates a next-state action. More precisely, to evaluate a formula ENABLED A, TLC computes successor states as if A were the next-state action. The formula evaluates to TRUE iff there exists a successor state. To check if a step $s \to t$ satisfies the composition $A \cdot B$ of actions A and B, TLC first computes all states u such that $s \to u$ is an A step and then checks if $u \to t$ is a B step for some such u.

Action composition is discussed on page 77.

TLC may also have to evaluate an action when checking a property. In that case, it evaluates the action as it would any expression, and it has no trouble evaluating even the bizarre actions (14.6).

14.3 How TLC Checks Properties

Section 14.2 above explains how TLC evaluates expressions and computes initial states and successor states. This section describes how TLC uses evaluation to check properties—first for model-checking mode (its default), and then for simulation mode.

First, let's define some formulas that are obtained from the configuration file. In these definitions, a *specification conjunct* is a conjunct of the formula named by the `SPECIFICATION` statement (if there is one), a *property conjunct* is a conjunct of a formula named by a `PROPERTY` statement, and the conjunction of an empty set of formulas is defined to be TRUE. The definitions use the four classes of nice temporal formulas defined above in Section 14.2.4 on page 235.

Init The specification's initial state predicate. It is specified by an `INIT` or `SPECIFICATION` statement. In the latter case, it is the conjunction of all specification conjuncts that are state predicates.

Next The specification's next-state action. It is specified by a `NEXT` statement or a `SPECIFICATION` statement. In the latter case, it is the action N such that there is a specification conjunct of the form $\Box[N]_v$. There must not be more than one such conjunct.

Temporal The conjunction of every specification conjunct that is neither a state predicate nor a box-action formula. It is usually the specification's liveness condition.

Invariant The conjunction of every state predicate I that is either named by an `INVARIANT` statement or for which some property conjunct equals $\Box I$.

ImpliedInit The conjunction of every property conjunct that is a state predicate.

ImpliedAction The conjunction of every action $[A]_v$ such that some property conjunct equals $\Box[A]_v$.

ImpliedTemporal The conjunction of every property conjunct that is a simple temporal formula, but is not of the form $\Box I$, where I is a state predicate.

Constraint The conjunction of all state predicates named by `CONSTRAINT` statements.

ActionConstraint The conjunction of all actions named by `ACTION-CONSTRAINT` statements. An action constraint is similar to an ordinary constraint, except it eliminates possible transitions rather than states. An ordinary constraint P is equivalent to the action constraint P'.

14.3.1 Model-Checking Mode

TLC keeps two data structures: a directed graph $\mathcal{G}$ whose nodes are states, and a queue (a sequence) $\mathcal{U}$ of states. A *state in* $\mathcal{G}$ means a state that is a node of the graph $\mathcal{G}$. The graph $\mathcal{G}$ is the part of the state reachability graph that TLC has found so far, and $\mathcal{U}$ contains all states in $\mathcal{G}$ whose successors TLC has not yet computed. TLC's computation maintains the following invariants:

- The states of $\mathcal{G}$ satisfy the *Constraint* predicate.
- For every state s in $\mathcal{G}$, the edge from s to s is in $\mathcal{G}$.
- If there is an edge in $\mathcal{G}$ from state s to a different state t, then t is a successor state of s that satisfies the action constraint. In other words, the step $s \to t$ satisfies $Next \land ActionConstraint$.
- Each state s of $\mathcal{G}$ is reachable from an initial state (one that satisfies the *Init* predicate) by a path in $\mathcal{G}$.
- $\mathcal{U}$ is a sequence of distinct states that are nodes in $\mathcal{G}$.
- For every state s in $\mathcal{G}$ that is not in $\mathcal{U}$, and for every state t satisfying *Constraint* such that the step $s \to t$ satisfies $Next \land ActionConstraint$, the state t and the edge from s to t are in $\mathcal{G}$.

TLC executes the following algorithm, starting with $\mathcal{G}$ and $\mathcal{U}$ empty:

1. Check that every ASSUME in the specification is satisfied by the values assigned to the constant parameters.
2. Compute the set of initial states by evaluating the initial predicate *Init*, as described above in Section 14.2.6. For each initial state s found:

(a) Evaluate the predicates *Invariant* and *ImpliedInit* in state s; report an error and stop if either is false.

(b) If the predicate *Constraint* is true in state s, then add s to the queue $\mathcal{U}$ and add node s and edge $s \to s$ to the graph $\mathcal{G}$.

3. While $\mathcal{U}$ is nonempty, do the following:

(a) Remove the first state from $\mathcal{U}$ and let s be that state.

(b) Find the set T of all successor states of s by evaluating the next-state action starting from s, as described above in Section 14.2.6.

(c) If T is empty and the *deadlock* option is *not* selected, then report a deadlock error and stop.

(d) For each state t in T, do the following:

i. If *Invariant* is false in state t or *ImpliedAction* is false for the step $s \to t$, then report an error and stop.

ii. If the predicate *Constraint* is true in state t and the step $s \to t$ satisfies *ActionConstraint*, then

A. If t is not in $\mathcal{G}$, then add it to the tail of $\mathcal{U}$ and add the node t and the edge $t \to t$ to $\mathcal{G}$.

B. Add the edge $s \to t$ to $\mathcal{G}$.

TLC can use multiple threads, and steps 3(b)–(d) may be performed concurrently by different threads for different states s. See the description of the *workers* option on page 253 below.

If formula *ImpliedTemporal* is not equal to TRUE, then whenever it adds an edge $s \to t$ in the procedure above, TLC evaluates all the predicates and actions that appear in formulas *Temporal* and *ImpliedTemporal* for the step $s \to t$. (It does this when adding any edge, including the self-loops $s \to s$ and $t \to t$ in steps 2(b) and 3(d)ii.A.)

Periodically during the computation of $\mathcal{G}$, and when it has finished computing $\mathcal{G}$, TLC checks the *ImpliedTemporal* property as follows. Let $\mathcal{T}$ be the set consisting of every behavior τ that is the sequence of states in an infinite path in $\mathcal{G}$ starting with an initial state. (For example, $\mathcal{T}$ contains the path $s \to s \to s \to \ldots$ for every initial state s in $\mathcal{G}$.) Note that every behavior in $\mathcal{T}$ satisfies $Init \wedge \Box[Next]_{vars}$. TLC checks that every behavior in $\mathcal{T}$ also satisfies the formula $Temporal \Rightarrow ImpliedTemporal$. (This is conceptually what happens; TLC does not actually check each behavior separately.) See Section 14.3.5 on page 247 below for a discussion of why TLC's checking of the *ImpliedTemporal* property may not do what you expect.

The computation of $\mathcal{G}$ terminates only if the set of reachable states is finite. Otherwise, TLC will run forever—that is, until it runs out of resources or is stopped.

See page 226 for the definition of *reachable state*.

TLC does not always perform all three of the steps described above. It does step 2 only for a non-constant module, in which case the configuration file must specify an *Init* formula. TLC does step 3 only if the configuration file specifies a *Next* formula, which it must do if it specifies an *Invariant*, *ImpliedAction*, or *ImpliedTemporal* formula.

14.3.2 Simulation Mode

In simulation mode, TLC repeatedly constructs and checks individual behaviors of a fixed maximum length. The maximum length can be specified with the *depth* option, as described on page 251 below. (Its default value is 100 states.) In simulation mode, TLC runs until you stop it.

To create and check a behavior, TLC uses the procedure described above for constructing the graph $\mathcal{G}$—except with the following difference. After computing the set of initial states, and after computing the set T of successors for a state s, TLC randomly chooses an element of that set. If the element does not satisfy the constraint, then the computation of $\mathcal{G}$ stops. Otherwise, TLC puts only that state in $\mathcal{G}$ and $\mathcal{U}$, and checks the *Invariant* and the *ImpliedInit* or the *ImpliedAction* formula for it. (The queue $\mathcal{U}$ isn't actually maintained, since it would never contain more than a single element.) The construction of $\mathcal{G}$ stops, and the formula $\mathit{Temporal} \Rightarrow \mathit{ImpliedTemporal}$ is checked, when the specified maximum number of states have been generated. TLC then repeats the procedure, starting with $\mathcal{G}$ and $\mathcal{U}$ empty.

TLC's choices are not strictly random, but are generated using a pseudo-random number generator from a randomly chosen seed. The seed and another value called the *aril* are printed if TLC finds an error. As described in Section 14.5.1 below, using the *key* and *aril* options, you can get TLC to generate the behavior that displayed the error.

14.3.3 Views and Fingerprints

In the description above of how TLC checks properties, I wrote that the nodes of the graph $\mathcal{G}$ are states. That is not quite correct. The nodes of $\mathcal{G}$ are values of a state function called the *view*. TLC's default view is the tuple of all declared variables, whose value determines the state. However, you can specify that the view should be some other state function *myview* by putting the statement

```
VIEW myview
```

in the configuration file, where *myview* is an identifier that is either defined or else declared to be a variable.

Remember that we are using the term *state* informally to mean an assignment of values to declared variables, rather than to all variables.

When TLC computes initial states, it puts their views rather than the states themselves in $\mathcal{G}$. (The view of a state s is the value of the `VIEW` state function in

state s.) If there are multiple initial states with the same view, only one of them is put in the queue $\mathcal{U}$. Instead of inserting an edge from a state s to a state t, TLC inserts the edge from the view of s to the view of t. In step 3(d)ii.A in the algorithm above, TLC checks if the view of t is in $\mathcal{G}$.

When using a view other than the default one, TLC may stop before it has found all reachable states. For the states it does find, it correctly performs safety checks—that is, the *Invariant*, *ImpliedInit*, and *ImpliedAction* checks. Moreover, it prints out a correct counterexample (a finite sequence of states) if it finds an error in one of those properties. However, it may incorrectly check the *ImpliedTemporal* property. Because the graph $\mathcal{G}$ that TLC is constructing is not the actual reachability graph, it may report an error in the *ImpliedTemporal* property when none exists, printing out a bogus counterexample.

Specifying a nonstandard view can cause TLC not to check many states. You should do it when there is no need to check different states that have the same view. The most likely alternate view is a tuple consisting of some, but not all, declared variables. For example, you may have added one or more variables to help debug the specification. Using the tuple of the original variables as the view lets you add debugging variables without increasing the number of states that TLC must explore. If the properties being checked do not mention the debugging variables, then TLC will find all reachable states of the original specification and will correctly check all properties.

In the actual implementation, the nodes of the graph $\mathcal{G}$ are not the views of states, but *fingerprints* of those views. A TLC fingerprint is a 64-bit number generated by a "hashing" function. Ideally, the probability that two different views have the same fingerprint is 2^{-64}, which is a very small number. However, it is possible for a *collision* to occur, meaning that TLC mistakenly thinks that two different views are the same because they have the same fingerprint. If this happens, TLC will not explore all the states that it should. In particular, with the default view, TLC will report that it has checked all reachable states when it hasn't.

When it terminates, TLC prints out two estimates of the probability that a fingerprint collision occurred. The first is based on the assumption that the probability of two different views having the same fingerprint is 2^{-64}. (Under this assumption, if TLC generated n views with m distinct fingerprints, then the probability of a collision is about $m * (n - m) * 2^{-64}$.) However, the process of generating states is highly nonrandom, and no known fingerprinting scheme can guarantee that the probability of any two distinct states generated by TLC having the same fingerprint is actually 2^{-64}. So, TLC also prints an empirical estimate of the probability that a collision occurred. It is based on the observation that, if there was a collision, then it is likely that there was also a "near miss". The estimate is the maximum value of $1/|f_1 - f_2|$ over all pairs $\langle f_1, f_2 \rangle$ of distinct fingerprints generated by TLC. In practice, the probability of collision turns out to be very small unless TLC is generating billions of distinct states.

Views and fingerprinting apply only to model-checking mode. In simulation mode, TLC ignores any `VIEW` statement.

14.3.4 Taking Advantage of Symmetry

The memory specifications of Chapter 5 are symmetric in the set *Proc* of processors. Intuitively, this means that permuting the processors doesn't change whether or not a behavior satisfies a specification. To define symmetry more precisely, we first need some definitions.

A *permutation of* a finite set S is a function whose domain and range both equal S. In other words, π is a permutation of S iff

$$(S = \text{DOMAIN } \pi) \ \land \ (\forall\, w \in S \ : \ \exists\, v \in S \ : \ \pi[v] = w)$$

A *permutation* is a function that is a permutation of its (finite) domain. If π is a permutation of a set S of values and s is a state, let s^π be the state obtained from s by replacing each value v in S with $\pi[v]$. To see what s^π means, let's take as an example the permutation π of $\{\text{“a”}, \text{“b”}, \text{“c”}\}$ such that $\pi[\text{“a”}] = \text{“b”}$, $\pi[\text{“b”}] = \text{“c”}$, and $\pi[\text{“c”}] = \text{“a”}$. Suppose that, in state s, the values of the variables x and y are

$$\begin{aligned} x &= \langle\, \text{“b”}, \text{“c”}, \text{“d”} \,\rangle \\ y &= [i \in \{\text{“a”}, \text{“b”}\} \mapsto \text{IF } \ i = \text{“a”} \text{ THEN } \ 7 \ \text{ ELSE } \ 42] \end{aligned}$$

Then in state s^π, the values of the variables x and y are

$$\begin{aligned} x &= \langle\, \text{“c”}, \text{“a”}, \text{“d”} \,\rangle \\ y &= [i \in \{\text{“b”}, \text{“c”}\} \mapsto \text{IF } \ i = \text{“b”} \text{ THEN } \ 7 \ \text{ ELSE } \ 42] \end{aligned}$$

This example should give you an intuitive idea of what s^π means; I won't try to define it rigorously. If σ is the behavior $s_1, s_2, \ldots$, let σ^π be the behavior $s_1^\pi, s_2^\pi, \ldots$.

We can now define what symmetry means. A specification *Spec* is *symmetric with respect to* a permutation π iff the following condition holds: for any behavior σ, formula *Spec* is satisfied by σ iff it is satisfied by σ^π.

The memory specifications of Chapter 5 are symmetric with respect to any permutation of *Proc*. This means that there is no need for TLC to check a behavior σ if it has already checked the behavior σ^π for some permutation π of *Proc*. (Any error revealed by σ would also be revealed by σ^π.) We can tell TLC to take advantage of this symmetry by putting the following statement in the configuration file:

```
SYMMETRY Perms
```

where *Perms* is defined in the module to equal *Permutations*(*Proc*), the set of all permutations of *Proc*. (The *Permutations* operator is defined in the *TLC*

module, described in Section 14.4 below.) This `SYMMETRY` statement causes TLC to modify the algorithm described on pages 241–242 so that it never adds a state s to its queue $\mathcal{U}$ of unexamined states and to its state graph $\mathcal{G}$ if $\mathcal{G}$ already contains the state s^π, for some permutation π of *Proc*. If there are n processes, this reduces the number of states that TLC examines by a factor of $n!$.

The memory specifications of Chapter 5 are also symmetric with respect to any permutation of the set *Adr* of memory addresses. To take advantage of this symmetry as well as the symmetry with respect to permutations of processors, we define the symmetry set (the set specified by the `SYMMETRY` statement) to equal

$$Permutations(Proc) \cup Permutations(Adr)$$

In general, the `SYMMETRY` statement can specify an arbitrary symmetry set Π, each element of which is a permutation of a set of model values. More precisely, each element π in Π must be a permutation such that all the elements of DOMAIN π are assigned model values by the configuration file's `CONSTANT` statement. (If the configuration has no `SYMMETRY` statement, we take the symmetry set Π to be the empty set.)

To explain what TLC does when given an arbitrary symmetry set Π, I need a few more definitions. If τ is a sequence $\langle \pi 1, \ldots, \pi n \rangle$ of permutations in Π, let s^τ equal $(\ldots((s^{\pi 1})^{\pi 2})\ldots)^{\pi n}$. (If τ is the empty sequence, then s^τ is defined to equal s.) Define the *equivalence class* $\widehat{s}$ of a state s to be the set of states s^τ for all sequences τ of permutations in Π. For any state s, TLC keeps only a single element of $\widehat{s}$ in $\mathcal{U}$ and $\mathcal{G}$. This is accomplished by the following modifications to the algorithm on pages 241–242. In step 2(b), TLC adds the state s to $\mathcal{U}$ and $\mathcal{G}$ only if $\mathcal{U}$ and $\mathcal{G}$ do not already contain a state in $\widehat{s}$. Step 3(d)ii is changed to

A. If no element in $\widehat{t}$ is in $\mathcal{G}$, then add t to the tail of $\mathcal{U}$ and add the node t and the edge $t \to t$ to $\mathcal{G}$.

B. Add the edge $s \to tt$ to $\mathcal{G}$, where tt is the unique element of $\widehat{t}$ that is (now) in $\mathcal{G}$.

When a `VIEW` statement appears in the configuration file, these changes are modified as described in Section 14.3.3 above so that views rather than states are put in $\mathcal{G}$.

If the specification and the properties being checked are, indeed, symmetric with respect to all permutations in the symmetry set, then TLC's *Invariant*, *ImpliedInit*, and *ImpliedAction* checking will find and correctly report any error that they would have found had the `SYMMETRY` statement been omitted. However, TLC may perform *ImpliedTemporal* checking incorrectly—it may miss errors, report an error that doesn't exist, or report a real error with an incorrect counterexample. So, you should do *ImpliedTemporal* checking when using a `SYMMETRY` statement only if you understand exactly what TLC is doing.

If the specification and properties are not symmetric with respect to all permutations in the symmetry set, then TLC may be unable to print an error trace if it does find an error. In that case, it will print the error message

```
Failed to recover the state from its fingerprint.
```

The symmetry set is used only in model-checking mode. TLC ignores it in simulation mode.

14.3.5 Limitations of Liveness Checking

If a specification violates a safety property, then there is a finite behavior that displays the violation. That behavior can be generated with a finite model. It is therefore, in principle, possible to discover the violation with TLC. It may be impossible to discover a violation of a liveness property with any finite model. To see why, consider the following simple specification *EvenSpec* that starts with x equal to zero and repeatedly increments it by 2:

Safety properties were defined on page 87.

$$EvenSpec \triangleq (x = 0) \land \Box[x' = x + 2]_x \land \mathrm{WF}_x(x' = x + 2)$$

Obviously, x never equals 1 in any behavior satisfying *EvenSpec*. So, *EvenSpec* does not satisfy the liveness property $\Diamond(x = 1)$. Suppose we ask TLC to check if *EvenSpec* implies $\Diamond(x = 1)$. To get TLC to terminate, we must provide a constraint that limits it to generating a finite number of reachable states. All the infinite behaviors satisfying $(x = 0) \land \Box[x' = x + 2]_x$ that TLC generates will then end in an infinite number of stuttering steps. In any such behavior, action $x' = x + 2$ is always enabled, but only a finite number of $x' = x + 2$ steps occur, so $\mathrm{WF}_x(x' = x + 2)$ is false. TLC will therefore not report an error because the formula

$$\mathrm{WF}_x(x' = x + 2) \Rightarrow \Diamond(x = 1)$$

is satisfied by all the infinite behaviors it generates.

When doing temporal checking, make sure that your model will permit infinite behaviors that satisfy the specification's liveness condition. For example, consider the finite model of the alternating bit protocol specification defined by the configuration file of Figure 14.3 on page 227. You should convince yourself that it allows infinite behaviors that satisfy formula *ABFairness*.

It's a good idea to verify that TLC is performing the liveness checking you expect. Have it check a liveness property that the specification does not satisfy and make sure it reports an error.

MODULE *TLC*

LOCAL INSTANCE *Naturals*
LOCAL INSTANCE *Sequences*

The keyword LOCAL means that definitions from the instantiated module are not obtained by a module that extends *TLC*.

OPERATORS FOR DEBUGGING

$Print(out, val) \triangleq val$ Causes TLC to print the values *out* and *val*.

$Assert(val, out) \triangleq$ IF $val =$ TRUE THEN TRUE ELSE CHOOSE v : TRUE Causes TLC to report an error and print *out* if *val* is not true.

$JavaTime \triangleq$ CHOOSE $n : n \in Nat$ Causes TLC to print the current time, in milliseconds elapsed since 00:00 on 1 Jan 1970 UT, modulo 2^{31}.

OPERATORS FOR REPRESENTING FUNCTIONS AND SETS OF PERMUTATIONS

$d :> e \triangleq [x \in \{d\} \mapsto e]$
$f \mathbin{@@} g \triangleq [x \in (\text{DOMAIN } f) \cup (\text{DOMAIN } g) \mapsto$ IF $x \in$ DOMAIN f THEN $f[x]$ ELSE $g[x]]$

The function f with domain $\{d_1, \ldots, d_n\}$ such that $f[d_i] = e_i$, for $i = 1, \ldots, n$ can be written $d_1 :> e_1 \mathbin{@@} \ldots \mathbin{@@} d_n :> e_n$

$Permutations(S) \triangleq \{f \in [S \to S] : \forall w \in S : \exists v \in S : f[v] = w\}$ The set of permutations of S.

AN OPERATOR FOR SORTING

$SortSeq(s, \_ \prec \_) \triangleq$ The result of sorting sequence s according to the ordering $\prec$.
LET $Perm \triangleq$ CHOOSE $p \in Permutations(1 \,..\, Len(s)) :$
$\forall i, j \in 1 \,..\, Len(s) : (i < j) \Rightarrow (s[p[i]] \prec s[p[j]]) \vee (s[p[i]] = s[p[j]])$
IN $[i \in 1 \,..\, Len(s) \mapsto s[Perm[i]]]$

Figure 14.5: The standard module *TLC*.

14.4 The *TLC* Module

The standard *TLC* module, in Figure 14.5 on this page, defines operators that are handy when using TLC. The module on which you run TLC usually EXTENDS the *TLC* module, which is overridden by its Java implementation.

Module overriding is explained above in Section 14.2.5.

Module *TLC* begins with the statement

LOCAL INSTANCE *Naturals*

As explained on page 171, this is like an EXTENDS statement, except that the definitions included from the *Naturals* module are not obtained by any other module that extends or instantiates module *TLC*. Similarly, the next statement locally instantiates the *Sequences* module.

Module *TLC* next defines three operators *Print*, *Assert*, and *JavaTime*. They are of no use except in running TLC, when they can help you track down problems.

The operator *Print* is defined so that *Print*(*out*, *val*) equals *val*. But when TLC evaluates this expression, it prints the values of *out* and *val*. You can add *Print* expressions to a specification to help locate an error. For example, if your specification contains

$\land$ *Print*("a", TRUE)
$\land$ *P*
$\land$ *Print*("b", TRUE)

and TLC prints the "a" but not the "b" before reporting an error, then the error happened while TLC was evaluating *P*. If you know where the error is but don't know why it's occurring, you can add *Print* expressions to give you more information about what values TLC has computed.

To understand what gets printed when, you must know how TLC evaluates expressions, which is explained above in Sections 14.2 and 14.3. TLC usually evaluates an expression many times, so inserting a *Print* expression in the specification can produce a lot of output. One way to limit the amount of output is to put the *Print* expression inside an IF/THEN expression, so it is executed only in interesting cases.

The *TLC* module next defines the operator *Assert* so *Assert*(*val*, *out*) equals TRUE if *val* equals TRUE. If *val* does not equal TRUE, evaluating *Assert*(*val*, *out*) causes TLC to print the value of *out* and to halt. (In this case, the value of *Assert*(*val*, *out*) is irrelevant.)

Next, the operator *JavaTime* is defined to equal an arbitrary natural number. However, TLC does not obey the definition of *JavaTime* when evaluating it. Instead, evaluating *JavaTime* yields the time at which the evaluation takes place, measured in milliseconds elapsed since 00:00 Universal Time on 1 January 1970, modulo 2^{31}. If TLC is generating states slowly, using the *JavaTime* operator in conjunction with *Print* expressions can help you understand why. If TLC is spending too much time evaluating an operator, you may be able to replace the operator's definition with an equivalent one that TLC can evaluate more efficiently. (See Section 14.2.3 on page 234.)

The TLC module next defines the operators :> and @@ so that the expression

$d_1 :> e_1 \ @@ \ \ldots \ @@ \ d_n :> e_n$

is the function f with domain $\{d_1, \ldots, d_n\}$ such that $f[d_i] = e_i$, for $i = 1, \ldots, n$. For example, the sequence $\langle$ "ab", "cd" $\rangle$, which is a function with domain {1, 2}, can be written as

```
1 :> "ab"  @@  2 :> "cd"
```

TLC uses these operators to represent function values that it prints when evaluating a *Print* expression or reporting an error. However, it usually prints values the way they appear in the specification, so it usually prints a sequence as a sequence, not in terms of the :> and @@ operators.

Next comes the definition of $Permutations(S)$ to be the set of all permutations of S, if S is a finite set. The *Permutations* operator can be used to specify a set of permutations for the `SYMMETRY` statement described in Section 14.3.4 above. More complicated symmetries can be expressed by defining a set $\{\pi_1, \ldots, \pi_n\}$ of permutations, where each π_i is written as an explicit function using the :> and @@ operators. For example, consider a specification of a memory system in which each address is in some way associated with a processor. The specification would be symmetric under two kinds of permutations: ones that permute addresses associated with the same processor, and ones that permute the processors along with their associated addresses. Suppose we tell TLC to use two processors and four addresses, where addresses $a11$ and $a12$ are associated with processor $p1$ and addresses $a21$ and $a22$ are associated with processor $p2$. We can get TLC to take advantage of the symmetries by giving it the following set of permutations as the symmetry set:

$$\begin{aligned} Permutations(\{a11, a12\}) \cup \{&p1 :> p2 \ @@ \ p2 :> p1 \\ &@@ \ a11 :> a21 \ @@ \ a21 :> a11 \\ &@@ \ a12 :> a22 \ @@ \ a22 :> a12\} \end{aligned}$$

The permutation $p1 :> p2 \ @@ \ldots @@ \ a22 :> a12$ interchanges the processors and their associated addresses. The permutation that just interchanges $a21$ and $a22$ need not be specified explicitly because it is obtained by interchanging the processors, interchanging $a11$ and $a12$, and interchanging the processors again.

The TLC module ends by defining the operator *SortSeq*, which can be used to replace operator definitions with ones that TLC can evaluate more efficiently. If s is a finite sequence and $\prec$ is a total ordering relation on its elements, then $SortSeq(s, \prec)$ is the sequence obtained from s by sorting its elements according to $\prec$. For example, $SortSeq(\langle 3, 1, 3, 8\rangle, >)$ equals $\langle 8, 3, 3, 1\rangle$. The Java implementation of *SortSeq* allows TLC to evaluate it more efficiently than a user-defined sorting operator. For example, here's how we can use *SortSeq* to define an operator *FastSort* to replace the *Sort* operator defined on page 235.

$$\begin{array}{l} FastSort(S) \ \triangleq \\ \quad \text{LET} \ MakeSeq[SS \in \text{SUBSET} \ S] \ \triangleq \\ \qquad \text{IF} \ SS = \{\} \ \text{THEN} \ \langle\,\rangle \\ \qquad\qquad \text{ELSE} \ \text{LET} \ ss \ \triangleq \ \text{CHOOSE} \ ss \in SS \ : \ \text{TRUE} \\ \qquad\qquad\qquad \text{IN} \ \ Append(MakeSeq[SS \setminus \{ss\}], ss) \\ \quad \text{IN} \ \ SortSeq(MakeSeq[S], <) \end{array}$$

14.5 How to Use TLC

14.5.1 Running TLC

Exactly how you run TLC depends on what operating system you are using and how it is configured. You will probably type a command of the form

program_name options spec_file

where

program_name is specific to your system. It might be `java tlatk.TLC`.

spec_file is the name of the file containing the TLA$^+$ specification. Each TLA$^+$ module named M that appears in the specification must be in a separate file named M`.tla`. The extension `.tla` may be omitted from *spec_file*.

options is a sequence consisting of zero or more of the following options:

`-deadlock`
: Tells TLC not to check for deadlock. Unless this option is specified, TLC will stop if it finds a deadlock—that is, a reachable state with no successor state.

`-simulate`
: Tells TLC to run in simulation mode, generating randomly chosen behaviors, instead of generating all reachable states. (See Section 14.3.2 above.)

`-depth` *num*
: This option causes TLC to generate behaviors of length at most *num* in simulation mode. Without this option, TLC will generate runs of length at most 100. This option is meaningful only when the *simulate* option is used.

`-seed` *num*
: In simulation mode, the behaviors generated by TLC are determined by the initial seed given to a pseudorandom number generator. Normally, the seed is generated randomly. This option causes TLC to let the seed be *num*, which must be an integer from -2^{63} to $2^{63} - 1$. Running TLC twice in simulation mode with the same seed and aril (see the *aril* option below) will produce identical results. This option is meaningful only when using the *simulate* option.

`-aril` *num*
: This option causes TLC to use *num* as the *aril* in simulation mode. The aril is a modifier of the initial seed. When TLC finds an error in simulation mode, it prints out both the initial seed and an aril

number. Using this initial seed and aril will cause the first trace generated to be that error trace. Adding *Print* expressions will usually not change the order in which TLC generates traces. So, if the trace doesn't tell you what went wrong, you can try running TLC again on just that trace to print out additional information.

`-coverage` *num*
: This option causes TLC to print "coverage" information every *num* minutes and at the end of its execution. For every action conjunct that "assigns a value" to a variable, TLC prints the number of times that conjunct has actually been used in constructing a new state. The values it prints may not be accurate, but their magnitude can provide useful information. In particular, a value of 0 indicates part of the next-state action that was never "executed". This might indicate an error in the specification, or it might mean that the model TLC is checking is too small to exercise that part of the action.

`-recover` *run_id*
: This option causes TLC to start executing the specification not from the beginning, but from where it left off at the last checkpoint. When TLC takes a checkpoint, it prints the run identifier. (That identifier is the same throughout an execution of TLC.) The value of *run_id* should be that run identifier.

`-cleanup`
: TLC creates a number of files when it runs. When it completes, it erases all of them. If TLC finds an error, or if you stop it before it finishes, TLC can leave some large files around. The *cleanup* option causes TLC to delete all files created by previous runs. Do not use this option if you are currently running another copy of TLC in the same directory; if you do, it can cause the other copy to fail.

`-difftrace` *num*
: When TLC finds an error, it prints an error trace. Normally, that trace is printed as a sequence of complete states, where a state lists the values of all declared variables. The *difftrace* option causes TLC to print an abridged version of each state, listing only the variables whose values are different than in the preceding state. This makes it easier to see what is happening in each step, but harder to find the complete state.

`-terse`
: Normally, TLC completely expands values that appear in error messages or in the output from evaluating *Print* expressions. The *terse* option causes TLC instead to print partially evaluated, shorter versions of these values.

`-workers` *num*
: Steps 3(b)–(d) of the TLC execution algorithm described on pages 241–242 can be speeded up on a multiprocessor computer by the use of multiple threads. This option causes TLC to use *num* threads when finding reachable states. There is no reason to use more threads than there are actual processors on your computer. If the option is omitted, TLC uses a single thread.

`-config` *config_file*
: Specifies that the configuration file is named *config_file*, which must be a file with extension `.cfg`. The extension `.cfg` may be omitted from *config_file*. If this option is omitted, the configuration file is assumed to have the same name as *spec_file*, except with the extension `.cfg`.

`-nowarning`
: There are TLA$^+$ expressions that are legal but are sufficiently unlikely that their presence probably indicates an error. For example, the expression $[f \text{ EXCEPT } ![v] = e]$ is probably incorrect if v is not an element of the domain of f. (In this case, the expression just equals f.) TLC normally issues a warning when it encounters such an unlikely expression; this option suppresses these warnings.

14.5.2 Debugging a Specification

When you write a specification, it usually contains errors. The purpose of running TLC is to find as many of those errors as possible. We hope an error in the specification will cause TLC to report an error. The challenge of debugging is to find the error in the specification that caused the error reported by TLC. Before addressing this challenge, let's first examine TLC's output when it finds no error.

TLC's Normal Output

When you run TLC, the first thing it prints is the version number and creation date:

TLC's messages may differ in format from the ones described here.

```
TLC Version 2.12 of 26 May 2003
```

Always include this information when reporting any problems with TLC. Next, TLC describes the mode in which it's being run. The possibilities are

```
Model-checking
```

in which it is exhaustively checking all reachable states, or

```
Running Random Simulation with seed 1901803014088851111.
```

in which it is running in simulation mode, using the indicated seed. (Seeds are described on pages 251–252.) Let's suppose it's running in model-checking mode. If you asked TLC to do liveness checking, it will now print something like

```
Implied-temporal checking--relative complexity = 8.
```

The time TLC takes for liveness checking is approximately proportional to the relative complexity. Even with a relative complexity of 1, checking liveness takes longer than checking safety. So, if the relative complexity is not small, TLC will probably take a very long time to complete, unless the model is very small. In simulation mode, a large complexity means that TLC will not be able to simulate very many behaviors. The relative complexity depends on the number of terms and the size of sets being quantified over in the temporal formulas.

TLC next prints a message like

```
Finished computing initial states:
4 states generated, with 2 of them distinct.
```

This indicates that, when evaluating the initial predicate, TLC generated 4 states, among which there were 2 distinct ones. TLC then prints one or more messages such as

```
Progress(9):  2846 states generated, 984 distinct states
found.  856 states left on queue.
```

This message indicates that TLC has thus far constructed a state graph $\mathcal{G}$ of diameter[2] 9, that it has generated and examined 2846 states, finding 984 distinct ones, and that the queue $\mathcal{U}$ of unexplored states contains 856 states. After running for a while, TLC generates these progress reports about once every five minutes. For most specifications, the number of states on the queue increases monotonically at the beginning of the execution and decreases monotonically at the end. The progress reports therefore provide a useful guide to how much longer the execution is likely to take.

$\mathcal{G}$ and $\mathcal{U}$ are described in Section 14.3.1 on page 241.

When TLC successfully completes, it prints

```
Model checking completed.  No error has been found.
```

It then prints something like

[2]The diameter of $\mathcal{G}$ is the smallest number d such that every state in $\mathcal{G}$ can be reached from an initial state by a path containing at most d states. It is the depth TLC has reached in its breadth-first exploration of the set of states. When using multiple threads (specified with the *workers* option), the diameter TLC reports may not be quite correct.

```
Estimates of the probability that TLC did not check all
reachable states because two distinct states had the same
fingerprint:
  calculated (optimistic):  .000003
  based on the actual fingerprints:  .00007
```

As explained on page 244, these are TLC's two estimates of the probability of a fingerprint collision. Finally, TLC prints a message like

```
2846 states generated, 984 distinct states found,
0 states left on queue.
The state graph has diameter 15.
```

with the total number of states and the diameter of the state graph.

While TLC is running, it may also print a message such as

```
-- Checkpointing run states/99-05-20-15-47-55 completed
```

This indicates that it has written a checkpoint that you can use to restart TLC in the event of a computer failure. (As explained in Section 14.5.3 on page 260, checkpoints have other uses as well.) The run identifier

```
states/99-05-20-15-47-55
```

is used with the *recover* option to restart TLC from where the checkpoint was taken. If only part of this message was printed—for example, because your computer crashed while TLC was taking the checkpoint—there is a slight chance that all the checkpoints are corrupted and you must start TLC again from the beginning.

Error Reports

The first problems you find in your specification will probably be syntax errors. TLC reports them with

```
ParseException in parseSpec:
```

followed by the error message generated by the Syntactic Analyzer. Chapter 12 explains how to interpret the analyzer's error messages. Running your specification through the analyzer as you write it will catch a lot of simple errors quickly.

As explained in Section 14.3.1 above, TLC executes three basic phases. In the first phase, it checks assumptions; in the second, it computes the initial states; and in the third, it generates the successor states of states on the queue $\mathcal{U}$ of unexplored states. You can tell if it has entered the third phase by whether or not it has printed the "initial states computed" message.

TLC's most straightforward error report occurs when it finds that one of the properties it is checking does not hold. Suppose we introduce an error into our alternating bit specification (Figure 14.1 on pages 223 and 224) by replacing the first conjunct of the invariant *ABTypeInv* with

$\land\ msgQ \in Seq(Data)$

TLC quickly finds the error and prints

```
Invariant ABTypeInv is violated
```

It next prints a minimal-length[3] behavior that leads to the state not satisfying the invariant:

Note that TLC indicates which part of the next-state action allows the step that produces each state.

```
The behavior up to this point is:
STATE 1: <Initial predicate>
/\ rBit = 0
/\ sBit = 0
/\ ackQ = <<  >>
/\ rcvd = d1
/\ sent = d1
/\ sAck = 0
/\ msgQ = <<  >>

STATE 2: <Action at line 66 in AlternatingBit>
/\ rBit = 0
/\ sBit = 1
/\ ackQ = <<  >>
/\ rcvd = d1
/\ sent = d1
/\ sAck = 0
/\ msgQ = << << 1, d1 >> >>
```

TLC prints each state as a TLA$^+$ predicate that determines the state. When printing a state, TLC describes functions using the operators :> and @@ defined in the TLC module. (See Section 14.4 on page 248.)

The hardest errors to locate are usually the ones detected when TLC is forced to evaluate an expression that it can't handle, or one that is "silly" because its value is not specified by the semantics of TLA$^+$. As an example, let's introduce a typical "off-by-one" error into the alternating bit protocol by replacing the second conjunct in the definition of *Lose* with

$$\begin{array}{l}\exists\, i \in 1\,..\,Len(q) :\\ \quad q' = [j \in 1\,..\,(Len(q)-1) \mapsto \text{IF } j < i \text{ THEN } q[j-1]\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \text{ELSE } q[j]]\end{array}$$

[3]When using multiple threads, it is possible, though unlikely, for there to be a shorter behavior that also violates the invariant.

If q has length greater than 1, then this defines $Lose(q)[1]$ to equal $q[0]$, which is a nonsensical value if q is a sequence. (The domain of a sequence q is the set $1 \ldots Len(q)$, which does not contain 0.) Running TLC produces the error message

```
Error: Applying tuple
<< << 1, d1 >>, << 1, d1 >> >>
to integer 0 which is out of domain.
```

It then prints a behavior leading to the error. TLC finds the error when evaluating the next-state action to compute the successor states for some state s, and s is the last state in that behavior. Had the error occurred when evaluating the invariant or the implied-action, TLC would have been evaluating it on the last state or step of the behavior.

Finally, TLC prints the location of the error:

```
The error occurred when TLC was evaluating the nested
expressions at the following positions:
0. Line 57, column 7 to line 59, column 60 in AlternatingBit
1. Line 58, column 55 to line 58, column 60 in AlternatingBit
```

The first position identifies the second conjunct of the definition of *Lose*; the second identifies the expression $q[j-1]$. This tells you that the error occurred while TLC was evaluating $q[j-1]$, which it was doing as part of the evaluation of the second conjunct of the definition of *Lose*. You must infer from the printed trace that it was evaluating the definition of *Lose* while evaluating the action *LoseMsg*. In general, TLC prints a tree of nested expressions—higher-level ones first. It seldom locates the error as precisely as you would like; often it just narrows it down to a conjunct or disjunct of a formula. You may need to insert *Print* expressions to locate the problem. See the discussion on page 259 for further advice on locating errors.

14.5.3 Hints on Using TLC Effectively

Start Small

The constraint and the assignment of values to the constant parameters define a model of the specification. How long it takes TLC to check a specification depends on the specification and the size of the model. Running on a 600MHz work station, TLC finds about 700 distinct reachable states per second for the alternating bit protocol specification. For some specifications, the time it takes TLC to generate a state grows with the size of the model; it can also increase as the generated states become more complicated. For some specifications run on larger models, TLC finds fewer than one reachable state per second.

You should always begin testing a specification with a tiny model, which TLC can check quickly. Let sets of processes and of data values have only one element; let queues be of length one. A specification that has not been tested probably has lots of errors. A small model will quickly catch most of the simple ones. When a very small model reveals no more errors, you can then run TLC with larger models to try to catch more subtle errors.

One way to figure out how large a model TLC can handle is to estimate the approximate number of reachable states as a function of the parameters. However, this can be hard. If you can't do it, increase the model size very gradually. The number of reachable states is typically an exponential function of the model's parameters; and the value of a^b grows very fast with increasing values of b.

Many systems have errors that will show up only on models too large for TLC to check exhaustively. After having TLC model check your specification on as large a model as your patience allows, you can run it in simulation mode on larger models. Random simulation is not an effective way to catch subtle errors, but it's worth trying; you might get lucky.

Be Suspicious of Success

Section 14.3.5 on page 247 explains why you should be suspicious if TLC does not find a violation of a liveness property; the finite model may mask errors. You should also be suspicious if TLC finds no error when checking safety properties. It's very easy to satisfy a safety property by simply doing nothing. For example, suppose we forgot to include the *SndNewValue* action in the alternating bit protocol specification's next-state action. The sender would then never try to send any values. But the resulting specification would still satisfy the protocol's correctness condition, formula *ABCSpec* of module *ABCorrectness*. (The specification doesn't require that values must be sent.)

The *coverage* option described on page 252 provides one way to catch such problems. Another way is to make sure that TLC finds errors in properties that should be violated. For example, if the alternating bit protocol is sending messages, then the value of *sent* should change. You can verify that it does change by checking that TLC reports a violation of the property

$$\forall\, d \in Data : (sent = d) \Rightarrow \Box(sent = d)$$

A good sanity check is to verify that TLC finds states that are reached only by performing a number of operations. For example, the caching memory specification of Section 5.6 should have reachable states in which a particular processor has both a read and two write operations in the *memQ* queue. Reaching such a state requires a processor to perform two writes followed by a read to an uncached address. We can verify that such a state is reachable by having TLC find a violation of an invariant declaring that there aren't a read and two writes for

the same processor in *memQ*. (Of course, this requires a model in which *memQ* can be large enough.) Another way to check that certain states are reached is by using the *Print* operator inside an IF/THEN expression in an invariant to print a message when a suitable state is reached.

Let TLC Help You Figure Out What Went Wrong

When TLC reports that an invariant is violated, it may not be obvious what part of the invariant is false. If you give separate names to the conjuncts of your invariant and list them separately in the configuration file's `INVARIANT` statement, TLC will tell you which conjunct is false. However, it may be hard to see why even an individual conjunct is false. Instead of spending a lot of time trying to figure it out by yourself, it's easier to add *Print* expressions and let TLC tell you what's going wrong.

If you rerun TLC from the beginning with a lot of *Print* expressions, it will print output for every state it checks. Instead, you should start TLC from the state in which the invariant is false. Define a predicate, say *ErrorState*, that describes this state, and modify the configuration file to use *ErrorState* as the initial predicate. Writing the definition of *ErrorState* is easy—just copy the last state in TLC's error trace.[4]

You can use the same trick if any safety property is violated, or if TLC reports an error when evaluating the next-state action. For an error in a property of the form $\Box[A]_v$, rerun TLC using the next-to-last state in the error trace as the initial predicate, and using the last state in the trace, with the variable names primed, as the next-state action. To find an error that occurs when evaluating the next-state action, use the last state in the error trace as the initial predicate. (In this case, TLC may find several successor states before reporting the error.)

If you have introduced model values in the configuration file, they will undoubtedly appear in the states printed by TLC. So, if you are to copy those states into the module, you will have to declare the model values as constant parameters and then assign to each of these parameters the model value of the same name. For example, the configuration file we used for the alternating bit protocol introduces model values $d1$ and $d2$. So, we would add to module *MCAlternatingBit* the declaration

CONSTANTS $d1$, $d2$

and add to the `CONSTANT` statement of the configuration file the assignments

```
d1 = d1    d2 = d2
```

which assign to the constant parameters $d1$ and $d2$ the model values `d1` and `d2`, respectively.

[4] Defining *ErrorState* is not so easy if you use the *difftrace* option, which is a reason for not using that option.

Don't Start Over After Every Error

After you've eliminated the errors that are easy to find, TLC may have to run for a long time before finding an error. Very often, it takes more than one try to fix an error properly. If you start TLC from the beginning after correcting an error, it may run for a long time only to report that you made a silly mistake in the correction. If the error was discovered when taking a step from a correct state, then it's a good idea to check your correction by starting TLC from that state. As explained above, you do this by defining a new initial predicate that equals the state printed by TLC.

Another way to avoid starting from scratch after an error is by using checkpoints. A checkpoint saves the current state graph $\mathcal{G}$ and queue $\mathcal{U}$ of unexplored states. It does not save any other information about the specification. You can restart TLC from a checkpoint even if you have changed the specification, as long as the specification's variables and the values that they can assume haven't changed. More precisely, you can restart from a checkpoint iff the view of any state computed before the checkpoint has not changed and the symmetry set is the same. When you correct an error that TLC found after running for a long time, you may want to use the *recover* option (page 252) to continue TLC from the last checkpoint instead of having it recheck all the states it has already checked.[5]

The view and symmetry set are defined in Sections 14.3.3 and 14.3.4, respectively.

Check Everything You Can

Verify that your specification satisfies all the properties you think it should. For example, you shouldn't be content to check that the alternating bit protocol specification satisfies the higher-level specification *ABCSpec* of module *ABCorrectness*. You should also check lower-level properties that you expect it to satisfy. One such property, revealed by studying the algorithm, is that there should never be more than two different messages in the *msgQ* queue. So, we can check that the following predicate is an invariant:

$$Cardinality(\{msgQ[i] : i \in 1 \,..\, Len(msgQ)\}) \leq 2$$

(We must add the definition of *Cardinality* to module *MCAlternatingBit* by adding *FiniteSets* to its EXTENDS statement.)

It's a good idea to check as many invariance properties as you can. If you think that some state predicate should be an invariant, let TLC test if it is. Discovering that the predicate isn't an invariant may not reveal an error, but it will probably teach you something about your specification.

[5] Some states in the graph $\mathcal{G}$ may not be saved by a checkpoint; they will be rechecked when restarting from the checkpoint.

Be Creative

Even if a specification seems to lie outside the realm of what it can handle, TLC may be able to help check it. For example, suppose a specification's next-state action has the form $\exists\, n \in Nat : A(n)$. TLC cannot evaluate quantification over an infinite set, so it apparently can't deal with this specification. However, we can enable TLC to evaluate the quantified formula by using the configuration file's `CONSTANT` statement to replace Nat with the finite set $0 \,..\, n$, for some n. This replacement profoundly changes the specification's meaning. However, it might nonetheless allow TLC to reveal errors in the specification. Never forget that your objective in using TLC is not to verify that a specification is correct; it's to find errors.

Replacement is explained in Section 14.2.3.

Use TLC as a TLA$^+$ Calculator

Misunderstanding some aspect of TLA$^+$ can lead to errors in your specification. Use TLC to check your understanding of TLA$^+$ by running it on small examples. TLC checks assumptions, so you can turn it into a TLA$^+$ calculator by having it check a module with no specification, only ASSUME statements. For example, if g equals

$$[f \text{ EXCEPT } ![d] = e_1,\ ![d] = e_2]$$

what is the value of $g[d]$? You can ask TLC by letting it check a module containing

$$\begin{array}{lll} \text{ASSUME} & \text{LET } f & \triangleq\ [i \in 1 \,..\, 10 \mapsto 1] \\ & \quad\ g & \triangleq\ [f \text{ EXCEPT } ![2] = 3,\ ![2] = 4] \\ & \text{IN} & \mathit{Print}(g[2], \text{TRUE}) \end{array}$$

You can have it verify that $(F \Rightarrow G) \equiv (\neg F \vee G)$ is a tautology by checking

$$\text{ASSUME}\ \ \forall\, F,\ G \in \text{BOOLEAN} : (F \Rightarrow G) \equiv (\neg F \vee G)$$

TLC can even look for counterexamples to a conjecture. Can every set be written as the disjunction of two different sets? Check it for all subsets of $1 \,..\, 4$ with

$$\begin{array}{l} \text{ASSUME}\ \ \forall\, S \in \text{SUBSET}\,(1 \,..\, 4) : \\ \qquad \text{IF}\ \ \exists\, T, U \in \text{SUBSET}\,(1 \,..\, 4) : (T \neq U) \wedge (S = T \cup U) \\ \qquad\quad \text{THEN}\ \ \text{TRUE} \\ \qquad\quad \text{ELSE}\ \ \mathit{Print}(S, \text{TRUE}) \end{array}$$

When TLC is run just to check assumptions, it may need no information from the configuration file. But you must provide a configuration file, even if that file is empty.

14.6 What TLC Doesn't Do

We would like TLC to generate all the behaviors that satisfy a specification. But no program can do this for an arbitrary specification. I have already mentioned some limitations of TLC. There are other limitations that you may stumble on. One of them is that the Java classes that override the *Naturals* and *Integers* modules handle only numbers in the interval $-2^{31} \,..\, (2^{31} - 1)$; TLC reports an error if any computation generates a value outside this interval.

TLC can't generate all behaviors satisfying an arbitrary specification, but it might achieve the easier goal of ensuring that every behavior it does generate satisfies the specification. However, for reasons of efficiency, TLC doesn't always meet this goal. It deviates from the semantics of TLA$^+$ in two ways.

The first deviation is that TLC doesn't preserve the precise semantics of CHOOSE. As explained in Section 16.1, if S equals T, then CHOOSE $x \in S : P$ should equal CHOOSE $x \in T : P$. However, TLC guarantees this only if S and T are syntactically the same. For example, TLC might compute different values for the two expressions

$$\text{CHOOSE } x \in \{1, 2, 3\} : x < 3 \qquad \text{CHOOSE } x \in \{3, 2, 1\} : x < 3$$

A similar violation of the semantics of TLA$^+$ exists with CASE expressions, whose semantics are defined (in Section 16.1.4) in terms of CHOOSE.

The second part of the semantics of TLA$^+$ that TLC does not preserve is the representation of strings. In TLA$^+$, the string "abc" is a three-element sequence—that is, a function with domain {1, 2, 3}. TLC treats strings as primitive values, not as functions. It thus considers the legal TLA$^+$ expression "abc"[2] to be an error.

14.7 The Fine Print

This section describes in detail two aspects of TLC that were sketched above: the grammar of the configuration file, and the precise definition of TLC values.

14.7.1 The Grammar of the Configuration File

The grammar of TLC's configuration file is described in the TLA$^+$ module *ConfigFileGrammar* in Figure 14.6 on the next page. More precisely, the set of sentences *ConfigGrammar.File*, where *ConfigGrammar* is defined in the module, describes all syntactically correct configuration files from which comments have been removed. The *ConfigFileGrammar* module extends the *BNFGrammars* module, which is explained above in Section 11.1.4 (page 179).

Here are some additional restrictions on the configuration file that are not specified by module *ConfigFileGrammar*. There can be at most one `INIT` and

```
MODULE ConfigFileGrammar
EXTENDS BNFGrammars

LEXEMES
Letter ≜ OneOf("abcdefghijklmnopqrstuvwxyz_ABCDEFGHIJKLMNOPQRSTUVWXYZ")
Num ≜ OneOf("0123456789")
LetterOrNum ≜ Letter ∪ Num
AnyChar ≜ LetterOrNum ∪ OneOf("~!@#\$%^&*-+=|(){}[],:;'' <>.?/")
SingularKW ≜ {"SPECIFICATION", "INIT", "NEXT", "VIEW", "SYMMETRY"}
PluralKW ≜
  {"CONSTRAINT", "CONSTRAINTS", "ACTION-CONSTRAINT", "ACTION-CONSTRAINTS",
   "INVARIANT", "INVARIANTS", "PROPERTY", "PROPERTIES"}
Keyword ≜ SingularKW ∪ PluralKW ∪ {"CONSTANT", "CONSTANTS"}
AnyIdent ≜ LetterOrNum* & Letter & LetterOrNum*
Ident ≜ AnyIdent \ Keyword

ConfigGrammar ≜ THE BNF GRAMMAR
  LET P(G) ≜
        ∧ G.File ::= G.Statement+
        ∧ G.Statement ::= Tok(SingularKW) & Tok(Ident)
                        | Tok(PluralKW) & Tok(Ident)*
                        | Tok({"CONSTANT", "CONSTANTS"})
                          & (G.Replacement | G.Assignment)*
        ∧ G.Replacement ::= Tok(Ident) & tok("<-") & Tok(AnyIdent)
        ∧ G.Assignment ::= Tok(Ident) & tok("=") & G.IdentValue
        ∧ G.IdentValue ::= Tok(AnyIdent) | G.Number | G.String
                         | tok("{")
                           & (Nil | G.IdentValue & (tok(",") & G.IdentValue)*)
                           & tok("}")
        ∧ G.Number ::= (Nil | tok("-")) & Tok(Num+)
        ∧ G.String ::= tok(" ") & Tok(AnyChar*) & tok(" ")
  IN LeastGrammar(P)
```

Figure 14.6: The BNF grammar of the configuration file.

one `NEXT` statement. There can be one `SPECIFICATION` statement, but only if there is no `INIT` or `NEXT` statement. (See page 243 in Section 14.3.1 for conditions on when these statements must appear.) There can be at most one `VIEW` statement and at most one `SYMMETRY` statement. Multiple instances of other statements are allowed. For example, the two statements

```
INVARIANT Inv1
INVARIANT Inv2 Inv3
```

specify that TLC is to check the three invariants $Inv1$, $Inv2$, and $Inv3$. These statements are equivalent to the single statement

```
INVARIANT Inv1 Inv2 Inv3
```

14.7.2 Comparable TLC Values

Section 14.2.1 (page 230) describes TLC values. That description is incomplete because it does not define exactly when values are comparable. The precise definition is that two TLC values are comparable iff the following rules imply that they are:

1. Two primitive values are comparable iff they have the same value type.

   This rule implies that "abc" and "123" are comparable, but "abc" and 123 are not.

2. A model value is comparable with any value. (It is equal only to itself.)

3. Two sets are comparable if they have different numbers of elements, or if they have the same numbers of elements and all the elements in one set are comparable with all the elements in the other.

   This rule implies that $\{1\}$ and $\{\text{"a"}, \text{"b"}\}$ are comparable and that $\{1, 2\}$ and $\{2, 3\}$ are comparable. However, $\{1, 2\}$ and $\{\text{"a"}, \text{"b"}\}$ are not comparable.

4. Two functions f and g are comparable iff (i) their domains are comparable and (ii) if their domains are equal, then $f[x]$ and $g[x]$ are comparable for every element x in their domain.

   This rule implies that $\langle 1, 2 \rangle$ and $\langle \text{"a"}, \text{"b"}, \text{"c"} \rangle$ are comparable, and that $\langle 1, \text{"a"} \rangle$ and $\langle 2, \text{"bc"} \rangle$ are comparable. However, $\langle 1, 2 \rangle$ and $\langle \text{"a"}, \text{"b"} \rangle$ are not comparable.