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

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

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

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

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 $(x' = \text{TRUE}) \land (y' = 1)$:

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

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 $\Box I$.