A PlusCal User's Manual

C-Syntax* Version 1.8

Leslie Lamport

31 August 2018

* There is also a P-Syntax version of this manual. See page 3 for a description of the two syntaxes.

Contents

Preface

This is an instruction manual for Version 1.4 of the c-syntax version of the PlusCal algorithm language. The following section, on page 3, explains the difference between this syntax and the alternative p-syntax. Section 1 explains what an algorithm language is and why you'd want to use one. Section 2 tells you what you need to know to get started using PlusCal. After reading it, you'll be able to write and check PlusCal algorithms.

You can read the other parts of this manual as you need them. The table of contents and the index can help you find what you need. Pages 70–72 at the end, just before the index, contain a series of tables that summarize a lot of useful information. The rest of the manual is arranged in the order you're likely to want to look at it:

- Section 3 describes the things you'll find in most programming language manuals, like the statements of the language. Once you've started writing PlusCal algorithms, you should browse this chapter to learn about features of PlusCal not mentioned in Section 2.

- We run programs, but we check algorithms. Section 2 gets you started using the translator and TLC model checker to check PlusCal algorithms. Section 4 tells you more about the translator and TLC. It's mostly about TLC, describing some of its additional features and how to use it to debug an algorithm. You should go to Section 4 if you don't understand what the translator or TLC is trying to say when it reports an error.

- Section 5 is mainly about writing PlusCal expressions. The expression language of PlusCal is much richer and more powerful than that of any programming language because it is based on mathematics, not on programming. The ten or so pages about expressions in Section 5 just introduce the subject. You can learn more from the book *Specifying Systems*, referred to here as the TLA$^+$ book [3], or from any books on the elementary mathematics of sets, functions, and logic—especially ones written by mathematicians and not computer scientists.

- Section A of the appendix contains a BNF grammar of PlusCal. The subjects of Appendix Sections B and C will make no sense to you until you've read Section 1.

Don't forget about the table of contents, the tables on pages 70–72, and the index.

I wish to thank the people who helped make PlusCal possible. Keith Marzullo collaborated in the writing of the translator and helped with the design of the PlusCal language. Georges Gonthier made many useful suggestions for the language.

The Two Syntaxes

PlusCal has two separate syntaxes, the prolix *p-syntax* and the more compact *c-syntax*. Here is a snippet of code written in the two syntaxes:

P-Syntax

```
while x > 0 do
  if y > 0 then y := y-1;
                x := x-1
           else x := x-2
  end if
end while;
print y;
```

C-Syntax

```
while (x > 0)
  { if (y > 0) { y := y-1;
                 x := x-1 }
    else x := x-2  } ;
print y;
```

The additional wordiness of the p-syntax makes the meaning of the code clearer, and its use of explicit `end`s instead of “`}`”s makes it easier to find syntax errors. However, the c-syntax version is shorter, and sensible formatting makes the meaning of the code clear enough. You may prefer the c-syntax version if you’re used to programming in a language derived from C, such as C++, C#, or Java.

This manual describes the c-syntax version. A manual for the p-syntax version is available from the TLA$^+$ tools Web site.

1 Introduction

PlusCal is an algorithm language. An algorithm language is meant for writing algorithms, not programs. Algorithms differ from programs in several ways:

- Algorithms perform operations on arbitrary mathematical objects, such as graphs and vector spaces. Programs perform operations on simple objects such as Booleans and integers; operations on more complex data types must be coded using lower-level operations such as integer addition and method invocation.

- A program describes one method of computing a result; an algorithm may describe a class of possible computations. For example, an algorithm might simply require that a certain operation be performed for all values of i from 1 to N. A program specifies in which order those operations are performed.

- Execution of an algorithm consists of a sequence of steps. An algorithm's computational complexity is the number of steps it takes to compute the result; defining a concurrent algorithm requires specifying what constitutes a single (atomic) step. There is no well-defined notion of a step of a program.

These differences between algorithms and programs are reflected in the following differences between PlusCal and programming languages.

- The language of PlusCal expressions is TLA$^+$, a high-level specification language based on set theory and first-order logic [3]. TLA$^+$ is infinitely more expressive than the expression language of any programming language. Even the subset of TLA$^+$ that can be executed by the TLC model checker is far more expressive than any programming language.[1]

- PlusCal provides simple constructs for expressing nondeterminism.

- PlusCal uses labels to describe the algorithm's steps. However, you can omit the labels and let the PlusCal translator add them as necessary. You are likely to do this only for uniprocess (sequential) algorithms,

[1]SETL [4] provides many of the set-theoretic primitives of TLA$^+$, but it can implement higher-level operations only by programming them with procedures and it cannot conveniently express nondeterminism.

where the partitioning of the computation into steps does not affect the values computed. For multiprocess (concurrent) algorithms, you will probably want to specify the grain of atomicity explicitly.

The primary goals of a programming language are efficiency of execution and ease of writing large programs. The primary goals of an algorithm language are making algorithms easier to understand and helping to check their correctness. Efficiency matters when executing a program that implements the algorithm. Algorithms are much shorter than programs, typically dozens of lines rather than thousands. An algorithm language doesn't need complicated concepts like objects or sophisticated type systems that were developed for writing large programs.

It is easy to write a PlusCal algorithm that cannot be executed—for example, one containing a statement that assigns to x the smallest integer for which Goldbach's conjecture[2] is false, if one exists, or else the value 0. An unexecutable algorithm can be interesting, and may represent a step in the development of a practical algorithm. However, most PlusCal users will want to execute their algorithms. The PlusCal translator compiles a PlusCal algorithm into a TLA$^+$ specification. If the algorithm manipulates only finite objects in a sensible way, then the TLC model checker will probably be able to execute that specification. When used in model-checking mode, TLC will check all possible executions of the algorithm. It can also be used in simulation mode to check randomly generated executions.

The Toolbox

You will almost certainly use the PlusCal translator and the TLC model checker with the TLA$^+$ Toolbox. This manual assumes that's you will be doing. However, it does not explain in any detail how to use the Toolbox. The Toolbox's *Help* pages explain that. See the TLA web site [1] to find out how to obtain the Toolbox.

The translator and TLC are ordinary Java programs and can be run from a command line. See the TLA$^+$ Tools page on the TLA web site to find out how to run the translator. The TLA$^+$ book explains how to run TLC.

[2]Goldbach's conjecture, which has not been proved or disproved, asserts that any even number greater than 2 is the sum of two primes.

2 Getting Started

I assume here that you've programmed in an imperative language like Java or Pascal or C. I will therefore not bother to explain the meaning of something like a `while` statement that appears in such languages. You can find the meaning of `while` and all other PlusCal statements in Section 3. (The index can help you.)

2.1 Typing the Algorithm

As an example, consider the following bit of PlusCal code that describes Euclid's algorithm, adapted from a version given by Sedgewick [5, page 8]. It sets `v` to the gcd (greatest common divisor) of `u` and `v`.

```
while (u ≠ 0)  (* ≠ is typed  #  or  /=  or  \neq . *)
   { if (u < v) { u := v || v := u } ;  \* swap u and v.
     u := u - v } ;
```

(The `{` and `}` in the `if` statement are not necessary because the multiple assignment is considered to be a single statement.) Comments indicate how to type symbols such as "$\neq$" that appear in the examples. A complete list of the ASCII versions of symbols appears in Table 5 on page 72.

You should find this code easy to understand, except for the "`||`" in the `if` statement on the second line. Assignments separated by "`||`"s (rather than by semicolons) form a single multiple assignment statement that is executed by first evaluating all the right-hand sides, then doing the assignments. Thus, as the comment says, the multiple assignment swaps the values of `u` and `v`.

The snippet of algorithm also indicates the two ways comments are written: either begun with "`\*`" and ended by the end of the line, or enclosed in matching "`(*`" and "`*)`" delimiters. Comments can be nested, so you can use "`(*`" and "`*)`" to comment out commented code.

Let's now put this piece of code into a complete algorithm. The algorithm begins

```
--algorithm EuclidAlg {
```

where we've given it the name `EuclidAlg`. We next declare the variables `u` and `v` and specify their initial values. (We could omit their initial values and initialize them with assignment statements, but it's better to do it this way.) Just to illustrate the two kinds of initialization, we give `u` the initial value 24, but let the initial value of `v` be any integer from 1 through some parameter `N`.

```
variables u = 24; v ∈ 1..N;   \* ∈  is typed \in .
```

The declaration of `v` asserts that its initial value is an element of the set `1 .. N` of integers from 1 through `N`. (Individual declarations can be separated by either semicolons or commas; the final semicolon [or comma] is required in the c-syntax.)

We add `print` statements to print out the initial values of the variables and the final value of `v`. The `print` statement can print the value of any arbitrary expression; to print multiple values, we can either let that expression be a tuple or else use multiple `print` statements. The complete algorithm is as follows, where the `while` loop is the same as above.

```
--algorithm EuclidAlg {
  variables u = 24 ; v ∈ 1 .. N ;
    { print ⟨u, v⟩ ;     \* ⟨ ... ⟩ is typed << ... >> .
      while (u ≠ 0)
         { ... } ;
      print ⟨"have gcd", v⟩  } }
```

2.2 The TLA$^+$ Module

The translated version of the algorithm is put inside a TLA$^+$ module. The algorithm must go in the same file as the module. The module begins

MODULE *Euclid*

which is typed as

```
----------------- MODULE Euclid ----------------
```

(The number of dashes in each "`--` ⋯ `--`" doesn't matter, as long as there are at least four.) The module name is arbitrary, but a module named *Euclid* must go in a file named `Euclid.tla`.

The module next imports two standard TLA$^+$ modules.

EXTENDS *Naturals*, *TLC*

The *Naturals* module defines common operators on natural numbers, including subtraction ("−") and the operator "`..`" that appear in the algorithm's expressions. The *TLC* module is needed if the algorithm uses a `print` statement. The EXTENDS statement must be the first statement in the module.

Next, the module declares the parameter `N`.

CONSTANT N

Every symbol or identifier that appears in an expression in the algorithm must be either (a) a built-in TLA$^+$ operator like $=$ or $\langle \dots \rangle$, (b) declared or defined in the module, or (c) declared or defined in an imported module.

The algorithm itself should appear in the module within a single comment:

```
(*  --algorithm EuclidAlg {
          ...    }  *)
```

(It may also be placed before or after the module in file `Euclid.tla`, but that's not a good idea.)

The translation is put in the module between the two single-line comments

```
\*  BEGIN TRANSLATION
\*  END TRANSLATION
```

If those lines are already present, the translator will delete everything between them and replace it with the TLA$^+$ translation of the algorithm. If not, the translator will insert the `BEGIN/END TRANSLATION` comment lines immediately after the comment containing the algorithm. You can put them elsewhere (as you must if the algorithm is outside the module), but you shouldn't.

The module ends with

which is typed as a string of four or more "`=`" characters.

You can create the file *Euclid.tla* in your favorite text editor and then open a new specification in the Toolbox with that as its root file. However, you will probably prefer to create the file in the Toolbox as a new specification.

2.3 Translating and Executing the Algorithm

You run the translator on the *Eulcid* module with the *Translate PlusCal Algorithm* command on the Toolbox's *File* menu (or by typing Control+T). (Because this is a uniprocess algorithm that contains no labels, the translator will automatically add the necessary labels.) After translating the algorithm, we can execute it by using the *New Model* command on the Toolbox's *TLC Model Checker* menu to have it create a new model. We must specify the value the model should assign to N. We do that in the *What is the model?* section of the model's *Model Overview* page. Let's assign it the (ordinary) value 3000.

Let's now run the TLC model checker on the specification produced by the translation. We'll first run it in simulation mode, which performs randomly-chosen possible executions—for this algorithm, by randomly choosing the initial value of N. (Use the *TLC Options* section of the *Advanced Options* model page.) TLC produces a gush of output like

```
<< 24, 1005 >>
<< "have gcd", 3 >>
<< 24, 200 >>
<< "have gcd", 8 >>
<< 24, 2717 >>
<< "have gcd", 1 >>
<< 24, 898 >>
<< "have gcd", 2 >>
<< 24, 1809 >>
      ⋮
```

that ends only when we stop TLC (with the *Cancel* button on the progress dialog).

Instead of having TLC randomly generate possible executions, we can run it in *model-checking mode*, in which it checks all possible executions of the algorithm. Go back to the *TLC Options* section of the *Advanced Options* page and select *Model-checking mode.* To avoid a huge mass of output, let's change the model to have it set N to 4, so there are only 4 possible executions of the algorithm. Running TLC now produces the following output:

```
<< 24, 1 >>
<< 24, 2 >>
<< 24, 3 >>
<< 24, 4 >>
<< "have gcd", 4 >>
<< "have gcd", 3 >>
<< "have gcd", 2 >>
<< "have gcd", 1 >>
```

TLC has checked the four possible executions, producing the eight possible executions of the `print` statements. But it did not perform those executions separately. Instead, TLC found all reachable states using a breadthfirst search. In doing so, it performed the four possible first steps before performing any of the four possible last steps.

If you want sensible output from running TLC in model-checking mode, you should have the algorithm execute only a single `print` statement at the end. For our example algorithm, this requires saving the initial value of `v` in a separate variable. So, we modify the algorithm by introducing a new variable `v_ini` whose initial value is the initial value of `v`.

```
--algorithm EuclidAlg {
  variables u = 24 ; v ∈ 1 .. N ; v_ini = v ;
    { while (u ≠ 0) ...
        { ... } ;
      print <<24, v_ini, "have gcd", v>>  } }
```

Translating and running TLC in model-checking mode on this algorithm produces the output

```
<< 24, 4, "have gcd", 4 >>
<< 24, 3, "have gcd", 3 >>
<< 24, 2, "have gcd", 2 >>
<< 24, 1, "have gcd", 1 >>
```

2.4 Checking the Results

We don't have to print the results and examine them by hand to check them. We can let TLC do the checking by using an `assert` statement. Suppose we have defined $gcd(x, y)$ to be the gcd of x and y. We can then replace the `print` statement in algorithm *EuclidAlg* by

```
assert v = gcd(24, v_ini)
```

TLC will print an error message if this statement is executed when `v` does not equal `gcd(24, v_ini)`. For this to work, the operator *gcd* must be defined in the TLA$^+$ module, before the translated algorithm—that is, before the "`BEGIN TRANSLATION`" line. You may be able to understand the TLA$^+$ definition of *gcd* knowing that:

- $gcd(x, y)$ is defined to be the largest integer that divides both x and y.
- An integer p divides an integer q iff (if and only if) $q \,\%\, p$ equals 0, where $q \,\%\, p$ is the remainder when q is divided by p.
- The gcd of x and y is at most equal to x (or y).

The standard TLA$^+$ operators that are used in the definition are briefly explained in Tables 1 and 2 on pages 70 and 71. Here is the definition; give it a try.

$$\begin{array}{l} gcd(x, y) \triangleq \textsc{choose}\ i \in 1 \,..\, x : \\ \quad \land\ x \,\%\, i = 0 \\ \quad \land\ y \,\%\, i = 0 \\ \quad \land\ \forall j \in 1 \,..\, x : \begin{array}[t]{l} \land\ x \,\%\, j = 0 \\ \land\ y \,\%\, j = 0 \\ \Rightarrow i \geq j \end{array} \end{array}$$

If you can't understand it now, you should be able to after reading Section 5.

2.5 Checking Termination

To check that algorithm `EuclidAlg` always terminates, we perform the translation with the `-termination` option. We do this by putting the line

```
PlusCal options (-termination)
```

in the file–either in a comment or else before or after the module. (The "`-`" can be omitted from an option name when it appears in the `options` statement.) This produces the appropriate translation that should ensure termination. If we then create a new model, it will add *Termination* to the *Properties* part of the *What to check?* section of the *Model Overview* page. This will cause TLC to check that all possible executions terminate. (The *Termination* property is included by the Toolbox for all PlusCal algorithms, but its box is checked only if the `termination` property specified for the root module when the model is created.)

If TLC discovers a non-terminating execution, it will produce an error message indicating that property *Termination* is violated, and the Toolbox's error trace window will show the non-terminating trace. Section 4.4 on page 36 explains how to interpret the trace.

2.6 A Multiprocess Algorithm

Algorithm `EuclidAlg` is a uniprocess algorithm, with only a single thread of control. We now look at an example of a multiprocess algorithm written in PlusCal. The example is the Fast Mutual Exclusion Algorithm [2]. The algorithm has N processes, numbered from 1 through N. Figure 1 on the next page is the original description of process number i, except with the noncritical section and the outer infinite loop made explicit. Angle brackets

```
  ncs: $\mathit{noncritical\ section}$;
start: $\langle b[i] := \mathit{true} \rangle$;
       $\langle x := i \rangle$;
       if $\langle y \neq 0 \rangle$ then $\langle b[i] := \mathit{false} \rangle$;
                      await $\langle y = 0 \rangle$;
                      goto $\mathit{start}$  fi;
       $\langle y := i \rangle$;
       if $\langle x \neq i \rangle$ then $\langle b[i] := \mathit{false} \rangle$;
                      for $j := 1$ to $N$ do await $\langle \neg b[j] \rangle$ od;
                      if $\langle y \neq i \rangle$ then await $\langle y = 0 \rangle$;
                                       goto $\mathit{start}$  fi  fi;
       $\mathit{critical\ section}$;
       $\langle y := 0 \rangle$;
       $\langle b[i] := \mathit{false} \rangle$;
       goto $\mathit{ncs}$
```

Figure 1: Process i of the fast mutual exclusion algorithm, based on the original description.

enclose atomic operations (steps). For example, the evaluation of the expression $y \neq 0$ in the first **if** statement is performed as a single step. If that expression equals *true*, the next step of the process sets $b[i]$ to *false*. The process's next atomic operation is the execution of the **await** statement, which is performed only when y equals 0. (The step cannot be performed when y is not equal to 0.)

The PlusCal version of this algorithm is in Figure 2 on the next page. After the algorithm name comes the declaration of the global variables:

```
variables x ; y = 0 ; b = [i ∈ 1..N ↦ FALSE] ;
```

(Here too, declarations can be separated by either semicolons or commas, and the final semicolon [or comma] is required.) The initial value of x doesn't matter and is not specified. The declaration of `b` states that it is initially an array indexed by the set `1..N` such that `b[i]` equals `FALSE` for every `i` in `1..N`. (The symbol "↦" is typed "`|->`".) The expression

$$[v \in S \mapsto v + 1]$$

equals an array A indexed by the set S such that $A[v] = v + 1$ for every v in S. What programmers call an array, mathematicians call a function. Like a mathematician, I usually call A a *function with domain S* rather than an

```
--algorithm FastMutex {
variables x ; y = 0 ; b = [i ∈ 1..N ↦ FALSE] ;
process (Proc ∈ 1..N)
 variable j ;
  { ncs: while (TRUE) {
            skip ;  \* The noncritical section.
     start: b[self] := TRUE ;
        l1: x := self ;
        l2: if (y ≠ 0) { l3: b[self] := FALSE ;
                         l4: await y = 0 ;
                             goto start  } ;
        l5: y := self ;
        l6: if (x ≠ self)
              { l7: b[self] := FALSE ;
                    j := 1 ;
                l8: while (j ≤ N) { await ~b[j] ;
                                    j := j+1 } ;
                l9: if (y ≠ self) { l10: await y = 0 ;
                                         goto start } } ;
        cs: skip ;  \* The critical section.
       l11: y := 0 ;
       l12: b[self] := FALSE } } }
```

Figure 2: The fast mutual exclusion algorithm in PlusCal.

array indexed by S. However, TLA$^+$ and PlusCal use programmers' square brackets instead of mathematicians' parentheses to represent array/function application. The $\mapsto$ construct and other TLA$^+$ notation for functions is explained in Section 5.5 on page 47.

The algorithm continues with

```
process (Proc ∈ 1..N)
```

This statement begins a collection named `Proc` of N processes, each process identified by a number in `1..N`. The statement

```
variable j ;
```

declares `j` to be a local variable of these processes, meaning that each of the N processes has its own separate variable `j`. A local or global variable `z` can be initialized by a declaration of the form

`variable z =` *exp* `;`   or   `variable z ∈` *exp* `;`

The matching "`{`" and "`}`" enclose the code for each process in the collection `Proc`, where `self` is the identifier of that process (in this example, a number in `1..N`).

The obvious difference between the original pseudo-code for process i in Figure 1 and the code for each process `self` of the PlusCal algorithm in Figure 2 is that the angle brackets have been replaced by labels. A single step (atomic action) of a PlusCal algorithm consists of the execution from one label to the next. For example, the execution of the test $y \neq 0$ at label `l2` is atomic because a single step that begins at `l2` ends when control reaches either `l3` or `l4`.

In the c-syntax, any sequence of statements can be enclosed in `{ }` braces, and the label of the first statement can appear either before or after the opening brace. You can therefore enclose atomic actions in braces. For example, you can write the step beginning at label `l7` of algorithm *FastMutex* as:

```
l7: { b[self] := FALSE ;
      j := 1 } ;
```

The PlusCal algorithm represents the noncritical and critical sections as atomic `skip` operations whose execution consists of a single "no-op" step that does nothing. The **await** statement of the original version is represented by the PlusCal `await` statement. A step containing a statement "`await` *exp*" can be executed only if the expression *exp* equals `TRUE`. Think of the processor attempting to execute the entire step. The attempt succeeds if the "`await` *exp*" statement is executed with *exp* equal to TRUE. In that case, the step is actually executed and control advances to the next step. If *exp* equals FALSE, then the attempted execution fails and nothing is changed; the processor will try to execute the step again later.

The keyword `when` is a synonym for `await`. You can use `await` and `when` interchangeably.

The original algorithm uses a **for** loop to test the values of $b[j]$ in increasing order of j. The `for` loop is represented in the PlusCal version by the `while` loop at label `l8` and the assignment statement that precedes it. When control in a process is at statement `l8` and $j \leq N$, then the next step of the process consists of an execution from `l8` back to `l8`—that is, a complete execution of the body of the `while` loop. If $j > N$, a step that begins at `l8` performs the `while` test and ends with control at `l9`.

The *FastMutex* algorithm works if the $b[j]$ are tested in arbitrary order. We can rewrite the algorithm to perform the tests in a nondeterministic order by replacing that PlusCal code with

```
    j := 1..N ;
l8: while (j ≠ {}) {                           \* {} is the empty set.
      with (p ∈ j) { await ~b[p] ;              \* ~ is logical negation.
                     j := j \ {p} } };         \* \ is set difference.
```

(If you don't know the meaning of the operator "\", look it up in the index.) The statement

`with (` *id* ∈ *S* `) do` *body*

sets *id* to a nondeterministically chosen element of the set S and then executes *body.* (In model-checking mode, TLC will check the algorithm for all possible choices of *id*.) Replacing $id \in S$ with $id = exp$ causes the body to be executed with *id* equal to the current value of *exp*.

A multiprocess algorithm can have multiple "`process`" sections. The statement

`process (` *Name* = *e* `)`

begins a single process named *Name* with identifier e. Note that *Name* is an arbitrary name that you give to the process; e is an expression. Changing *Name* has no effect on the algorithm, but changing the process's identifier e can make a difference. Different processes must have different identifiers.

Moreover, the identifiers of all processes should have the same "type"—for example, they should all be integers or all be strings or all be sets of records.

The safety property that algorithm *FastMutex* should satisfy is mutual exclusion, meaning that at most one process can be in its critical section at any one time. For the PlusCal version, this means that no two processes can be at the statement labeled `cs`. An *invariant* is an assertion that is true in every state that can occur during an execution of the algorithm. Mutual exclusion is the invariance of the assertion "no two processes are at statement `cs`". We can tell TLC to check that this assertion is an invariant. But first, we must know how to express the assertion in TLA$^+$.

The TLA$^+$ translation introduces a new variable *pc* whose value is the label of the next statement to be executed. For algorithm *FastMutex*, a process with identifier i executes the statement labeled `l5` next iff $pc[i]$ equals the string "l5". For any multiprocess algorithm, the value of the variable *pc* is a function whose domain is the set of process identifiers. For a uniprocess algorithm, the value of *pc* is a single string equal to the label of the next statement to be executed. There is an implicit label `Done` at the end of every process, and at the end of a uniprocess algorithm. (Since a process of algorithm *FastMutex* never terminates, $pc[i]$ never equals "Done" for any process i.

In algorithm *FastMutex*, a process i is at statement `cs` iff $pc[i]$ equals "cs". Mutual exclusion is therefore asserted by the invariance of the predicate *Mutex*, defined by

$$\begin{array}{l} \mathit{Mutex} \;\triangleq \\ \qquad \forall\, i,\, k \in 1\,..\,N : (i \neq k) \Rightarrow \neg((pc[i] = \text{“cs”}) \wedge (pc[k] = \text{“cs”})) \end{array}$$

(The operators like $\forall$, $\Rightarrow$, $\neg$, and $\wedge$ are explained in Section 5.3 on page 44. Section 3.8 on page 30 explains why we could not use the identifier j instead of k in the $\forall$ expression.)

TLA$^+$ allows a definition to refer only to variables and operators that have already been defined or declared. Since the definition of *Mutex* uses the variable *pc*, which is declared by the translation of the algorithm, this definition must come after the translation—in other words, after the "`END TRANSLATION`" line.

We tell TLC to check the invariance of *Mutex* by adding it as an invariant in the *What to check?* section of the *Model Overview* page. We need not have defined *Mutex* in the module; we could instead just use its definition

$$\forall\, i,\, k \in 1\,..\,N : (i \neq k) \Rightarrow \neg((pc[i] = \text{“cs”}) \wedge (pc[k] = \text{“cs”}))$$

as the invariant to check.

The variable *pc* can be used in the algorithm's expressions. We could therefore also check mutual exclusion by replacing the `skip` statement `cs` with `assert` statement in statement `cs`:

```
cs: assert ∀ i ∈ 1..N : (i ≠ self) ⇒ (pc[i] ≠ "cs");
```

("∀" is typed "`\A`", and "⇒" is typed "`=>`".) When using an `assert`, we must also import the *TLC* module in the EXTENDS statement.

Invariance checking is discussed further in Section 4.5. Section 4.6 describes how to check liveness properties, which are the generalization of termination.

2.7 Where Labels Must and Can't Go

The labeling of statements in a PlusCal algorithm is not completely arbitrary but must obey certain rules. The complete list of rules is given in Section 3.7 on page 29. Here are the most common rules, which apply to the *FastMutex* algorithm.

- The code for each process (and a uniprocess program) must begin with a label.
- A `while` statement must be labeled.
- The `do` clause of a `with` statement cannot contain any labeled statements. This rule restricts what can appear within a `with` statement—for example, it prohibits nesting a `while` inside a `with`. These restrictions apply even when you let the translator add labels. The complete list of such restrictions is given in Section 3.2.6 on page 23.
- An `if` statement that contains a label within it must be followed by a label. For example, label `l5` of algorithm *FastMutex* cannot be omitted.
- In any control path from one label to the next, there cannot be two separate assignment statements to the same variable. For example, this rule would be violated by two assignments to variable *b* if labels `l1`, `l2`, and `l3` were all removed—even if one of the assignments was to a different element of the array. However, a single multiple assignment such as

  ```
  x[1] := 1 || x[2] := 2
  ```

may assign to different components of the same variable.

- In the c-syntax, an opening brace cannot be both preceded and followed by a label—that is, you can't write something like "`l1: { l2:`".

The PlusCal translator will tell you if you have violated any of these rules. Running it with the `-label` option causes the translator to add any necessary labels. (The text of the algorithm in the source file is not changed; the translator adds the labels internally.) The `-reportLabels` translator option is like the `-label` option, but it causes the translator to tell you where it added the labels. The `-label` option is the default for a uniprocess algorithm if (and only if) you don't type any labels yourself.

If the algorithm contains no labels, then the translator will add as few labels as possible, resulting in an algorithm with the fewest possible steps. Using the fewest steps makes model checking as fast as possible.

3 The Language

This section lists the statements and constructs of PlusCal and explains their meanings. In doing so, it also describes the language's grammar. A BNF specification of the grammar appears in Section A on page 57 of the appendix.

Before getting to the language description, we need some definitions. A *statement sequence* is a sequence of statements, each ended by a semicolon. For example, the body of a `while` statement consists of a sequence of statements. If there is an `if` statement in that sequence of statements, then its *then* clause (the statement or the sequence of statements enclosed in braces that follows the `if` test) consists of a separate sequence of statements. The statements in the *then* clause are not part of the sequence that forms the `while`'s body. (It is the `if` statement, not the statements that occur inside it, that is a statement of the `while`'s body.)

A *control path* is a path through a piece of PlusCal code that represents a syntactically possible execution sequence, if we ignore how the statements are executed. For example, in the code

```
a: if (FALSE) {    goto w ;
                b: x := 7 ;
                c: y := 8   } ;
d: x := 0 ;
```

there is a control path that goes from the label `a` to the label `c`—even though no execution can actually follow that path.

A *step* is a control path that starts at a label, ends at a label, and passes through no other labels. In the example above, there are two steps beginning at label `a`—one that ends at `b` and one that ends at `d`. Remember that there is an implicit label `Done` at the end of a uniprocess algorithm and at the end of each process in a multiprocess algorithm. An execution of a PlusCal algorithm consists of a sequence of executions of steps. Part of a step can never be executed by itself (except for a `print` or `assert` statement, as described below).

3.1 Expressions

The expressions in PlusCal algorithms can be any TLA$^+$ expressions that do not contain a PlusCal reserved word or symbol such as `begin` or "`||`". You can write arbitrary TLA$^+$ definitions in the module before the "`BEGIN TRANSLATION`" line and use the defined symbols in the algorithm's expressions. Section 5 explains how to write TLA$^+$ expressions and definitions. Table 1 on page 70 and Table 2 on page 71 provide a convenient summary.

You are probably used to programming languages that allow only simple operators in expressions and allow variables to have only simple values. In PlusCal, the following statement assigns to `x` a record whose `a` component is the set of integers from 1 to `N` and whose `bcd` component is the set of all prime numbers less than or equal to `N`.

```
x := [a   ↦ 1..N,
      bcd ↦ {i ∈ 2..N : ∀ j ∈ 2..(i-1) : i % j ≠ 0} ]
```

It may be a while before you learn how to take advantage of PlusCal's powerful expression language.

TLA$^+$ has the general rule that an identifier cannot be assigned a new meaning if it already has a meaning. Thus, the identifier i cannot be used as a bound variable in an expression like

$$[i \in 1 \,..\, N \mapsto \text{FALSE}]$$

if it already has a meaning—for example, if i is an algorithm variable. Assigning a new meaning to a symbol can result in a "multiply-defined symbol" syntax error in the algorithm's TLA$^+$ translation.

3.2 The Statements

The examples in Section 2 contain most PlusCal statements. A statement that appears in the body of an algorithm, process, procedure, or macro must be either one of the statements listed below or else a *compound statement*. A compound statement consists of a sequence of statements separated by semicolons, with an optional semicolon at the end, that is enclosed in braces (`{ }`). Each statement's description includes the rules for labels that pertain to it. The labeling rules are also all listed in Section 3.7 below.

3.2.1 Assignment

An assignment is either an assignment to a variable such as

```
y := A + B
```

or else an assignment to a component, such as

```
x.foo[i+1] := y+3
```

If the current value of `x` is a record with a `foo` component that is a function (array), then this assignment sets the component `x.foo[i+1]` to the current value of `y+3`. The value of this assignment is undefined if the value of `x` is not a record with a `foo` component, or if `x.foo` is not a function. Therefore, if such an assignment appears in the code, then `x` will usually be initialized to an element of the correct "type", or to be a member of some set of elements of the correct type. For example, the declaration

```
variable x ∈ [bar : BOOLEAN,
              foo : [1..N → {"on", "off"}] ] ;
```

asserts that initially `x` is a record with a `bar` component that is a Boolean (equal to `TRUE` or `FALSE`) and a `foo` component that is a function with domain `1..N` such that `x.foo[i]` equals either "on" or "off" for each `i` in `1..N`. (The symbol "→" is typed "`->`".)

An *assignment statement* consists of one or more assignments, separated by "`||`" tokens, ending with a semicolon. An assignment statement containing more than one assignment is called a *multiple assignment.* A multiple assignment is executed by first evaluating the right-hand sides of all its assignments, and then performing those assignments from left to right. For example, if `i` = `j` = 3, then executing

```
x[i] := 1 || x[j] := 2
```

sets `x[3]` to 2.

Assignments to the same variable cannot be made in two different assignment statements within the same step. In other words, in any control path, a label must come between two statements that assign to the same variable. However, assignments to components of the same variable may appear in a single multiple assignment, as in

```
x.foo[7] := 13 || y := 27 || x.bar := x.foo
```

3.2.2 If

The `if` statement has its usual meaning. The statement

`if (` *test* `)` *t_clause* `else` *e_clause*

is executed by evaluating the expression *test* and then executing the (possibly compound) statement *t_clause* or *e_clause* depending on whether *test* equals TRUE or FALSE. The `else` clause is optional. An `if` statement must have a non-empty `then` clause. An `if` statement that contains a `call`, `return`, or `goto` statement or a label within it must be followed by a labeled statement. (A label on the `if` statement itself is not considered to be within the statement.)

3.2.3 Either

The `either` statement has the form:

$$\begin{array}{l}\texttt{either}\ \textit{clause}_1\\ \quad\texttt{or}\ \textit{clause}_2\\ \quad\ \ \vdots\\ \quad\texttt{or}\ \textit{clause}_n\end{array}$$

where each *clause*$_i$ is a (possibly compound) statement. It is executed by nondeterministically choosing any *clause*$_i$ that is executable and executing it. The `either` statement can be executed iff at least one of those clauses can be executed. If any *clause*$_i$ contains a `call`, `return`, or `goto` statement or a label, then the `either` statement must be followed by a labeled statement. The statement

$$\texttt{if (}\textit{test}\texttt{)}\ \ \textit{t\_clause}\ \ \texttt{else}\ \ \textit{e\_clause}$$

is equivalent to the following, where the `await` statement is explained in Section 3.2.5 on the next page.

$$\begin{array}{l}\texttt{either}\ \{\texttt{await}\ \ \ \textit{test}\ ;\ \ \textit{t\_clause}\ \}\\ \quad\texttt{or}\ \{\texttt{await}\ \neg\,\textit{test}\ ;\ \ \textit{e\_clause}\ \}\end{array}$$

3.2.4 While

The `while` statement has its usual meaning. The statement

$$\texttt{lb : while (}\ \textit{test}\ \texttt{)}\ \ \textit{body}$$

where *body* is a (possibly compound) statement, is executed like the following `if` statement, where the `goto` statement is explained in Section 3.2.11 on page 25.

$$\texttt{lb : if (}\ \textit{test}\ \texttt{)}\ \{\textit{body}\ \ ;\ \texttt{goto lb}\}$$

A `while` statement must be labeled. However, the statement following a `while` statement need not be labeled, even if there is a label in *body*.

3.2.5 Await (When)

A step containing the statement `await` *expr* can be executed only when the value of the Boolean expression *expr* is `TRUE`. Although it usually appears at the beginning of a step, an `await` statement can appear anywhere within the step. For example, the following two pieces of code are equivalent.

```
a : x := y + 1 ;            a : await y + 1 > 0 ;
    await x > 0 ;                 x := y + 1 ;
b : ...                     b : ...
```

The step from `a` to `b` can be executed only when the current value of `y+1` is positive. (Remember that an entire step must be executed; part of a step cannot be executed by itself.) The keyword `when` can be used instead of `await`.

3.2.6 With

The statement

`with (` $id \in S$ `)` *body*

is executed by executing the (possibly compound) statement *body* with identifier *id* equal to a nondeterministically chosen element of S. (The symbol $\in$ is typed "`\in`".) Execution is impossible if S is empty. This `with` statement is therefore equivalent to

`await` $S \neq$ `{}` ; `with (` $id \in S$ `)` *body*

The two statements

`with (` *id* `=` *expr* `)` ...   `with (` $id \in$ `{`*expr*`}` `)` ...

are equivalent. (The expression `{`*expr*`}` equals the set containing a single element equal to *expr*.)

In general, a `with` statement has the form

`with (` $id_1 \star expr_1$ `; ... ;` $id_n \star expr_n$ `)` *body*

where each $\star$ may be either $=$ or $\in$. (Commas may be used instead of semicolons between the $id_i \star expr_i$ items.) This statement is equivalent to

`with (` $id_1 \star expr_1$ `) ... with (` $id_n \star expr_n$ `)` *body*

The body of a `with` statement may not contain a label. This rule, combined with the rules listed in Section 3.7 (page 29) for where labels are required, implies that the following may not appear within the body of a `with` statement.

- A `while` statement.
- Two separate assignment statements that assign values to the same variable. (A single multiple assignment may assign values to different components of the same variable.)
- Any statement following a `return`, or any statement other than a `return` following a `call`.

3.2.7 Skip

The statement `skip;` does nothing.

3.2.8 Print

Execution of the statement

```
print expr ;
```

is equivalent to `skip`, except it causes TLC to print the current value of *expr*. TLC may print the value even if the step containing the `print` statement is not executed because of an `await` statement that appears later in the step.

An algorithm containing a `print` statement must be in a module that EXTENDS the *TLC* module.

3.2.9 Assert

The statement

```
assert expr ;
```

is equivalent to `skip` if expression *expr* equals TRUE. If *expr* equals false, executing the statement causes TLC to produce an error message saying that the assertion failed and giving the location of the `assert` statement. TLC may report a failed assertion even if the step containing the `assert` statement is not executed because of an `await` statement that appears later in the step.

An algorithm containing an `assert` statement must be in a module that EXTENDS the *TLC* module.

3.2.10 Call and Return

The `call` and `return` statements are described below in Section 3.4 on page 26.

3.2.11 Goto

Executing the statement

> `goto` *lab* `;`

ends the execution of the current step and causes control to go to the statement labeled *lab*. In any control path, a `goto` must be immediately followed by a label. (Remember that the control path by definition ignores the meaning of the `goto` and continues to what is syntactically the next statement.)

It is legal for a `goto` to jump into the middle of a `while` or `if` statement, but this sort of trickery should be avoided.

3.3 Processes

A multiprocess algorithm contains one or more processes. A process begins in one of two ways:

> `process (` *ProcName* $\in$ *IdSet* `)`
>
> `process (` *ProcName* `=` *Id* `)`

The first form begins a *process set*, the second an individual process. The identifier *ProcName* is the process or process set's name. The elements of the set *IdSet* and the element *Id* are called *process identifiers*. The process identifiers of different processes in the same algorithm must all be different. This means that the semantics of TLA$^+$ must imply that they are different, which intuitively usually means that they must be of the same "type". (For example, the semantics of TLA$^+$ does not specify whether or not a string may equal a number.) For execution by TLC, this means that all process identifiers must be comparable values, as defined on page 264 of the TLA$^+$ book [3].

The name *ProcName* has no significance; changing it does not change the meaning of the `process` statement in any way. The name appears in the TLA$^+$ translation, and it should be different for different `process` statements

As explained above in Section 2.6 on page 11, the `process` statement is optionally followed by declarations of local variables. The process body is a sequence of statements enclosed by braces (`{ }`). Its first statement must be labeled. Within the body of a process set, `self` equals the current process's identifier.

A multiprocess algorithm is executed by repeatedly choosing an arbitrary process and executing one step of that process, if that step's execution is possible. Execution of the process's next step is impossible if the process has terminated, if its next step contains an `await` statement whose expression equals FALSE, or if that step contains a statement of the form "`await` $x \in S$" and S equals the empty set. As explained in Section 2.6 on page 11, fairness conditions may be specified on the choice of which processes' steps are to be executed.

3.4 Procedures

An algorithm may have one or more procedures. If it does, the algorithm must be in a TLA$^+$ module that EXTENDS the *Sequences* module.

The algorithm's procedures follow its global variable declarations and `define` section (if any) and precede the "`{`" that begins the body of a uniprocess algorithm or the first process of a multiprocess algorithm. A procedure named *PName* begins

`procedure` *PName* `(` $param_1$`,` ... `,` $param_n$ `)`

where the identifiers $param_i$ are the *formal parameters* of the procedure. These parameters are treated as variables and may be assigned to. As explained in Section 4.5 on page 37, there may also be initial-value assignments of the parameters.

The `procedure` statement is optionally followed by declarations of variables local to the procedure. These have the same form as the declarations of global variables, except that initializations may only have the form "*variable* `=` *expression*". The procedure's local variables are initialized on each entry to the procedure.

Any variable declarations are followed by the procedure's body, which is a sequence of statements enclosed by braces (`{ }`). The body must begin with a labeled statement. There is an implicit label `Error` immediately after the body. If control ever reaches that point, then execution of either the process (multiprocess algorithm) or the complete algorithm (uniprocess algorithm) halts.

A procedure *PName* can be called by the statement

`call` *PName* `(` $expr_1$`,` ... `,` $expr_n$ `)`

Executing this call assigns the current values of the expressions $expr_i$ to the corresponding parameters $param_i$, initializes the procedure's local variables, and puts control at the beginning of the procedure body.

A `return` statement assigns to the parameters and local procedure variables their previous values—that is, the values they had before the procedure was last called—and returns control to the point immediately following the `call` statement.

The `call` and `return` statements are considered to be assignments to the procedure's parameters and local variables. In particular, they are included in the rule that a variable can be assigned a value by at most one assignment statement in a step. For example, if x is a local variable of procedure P, then a step within the body of P that (recursively) calls P cannot also assign a value to x.

For a multiprocess algorithm, the identifier `self` in the body of a procedure equals the process identifier of the process within which the procedure is executing.

The `return` statement has no argument. A PlusCal procedure does not explicitly return a value. A value can be returned by having the procedure set a global variable and having the code immediately following the `call` read that variable. For example, in a multiprocess algorithm, procedure P might use a global variable `rVal` to return a value by executing

```
rVal[self] := ... ;
return ;
```

From within a process in a process set, the code that calls P might look like this:

```
call P(17) ;   lab: x := ... rVal[self] ... ;
```

For a call from within a single process, the code would contain the process's identifier instead of `self`.

In any control path, a `return` statement must be immediately followed by a label. A `call` statement must either be followed in the control path by a label or else it must appear immediately before a `return` statement in a statement sequence.

When a `call` P statement is followed immediately by a `return`, the return from procedure P and the return performed by the `return` statement are both executed as part of a single execution step. When these statements are in the (recursive) procedure P, this combining of the two returns is essentially the standard optimization of replacing tail recursion by a loop.

3.5 Macros

A macro is like a procedure, except that a call of a macro is expanded at translation time. You can think of a macro as a procedure that is executed within the step from which it is called.

A macro definition looks much like a procedure declaration—for example:

```
macro P(s, i) { await s ≥ i ;
                s := s - i  }
```

The difference is that the body of the macro may contain no labels, no `while`, `call`, `return`, or `goto` statement. It may contain a call of a previously defined macro. Macro definitions come right after any global variable declarations and `define` section.

A macro call is like a procedure call, except with the `call` omitted—for example:

```
P(sem, y + 17) ;
```

The translation replaces the macro call with the sequence of statements obtained from the body of the macro definition by substituting the arguments of the call for the definition's parameters. Thus, this call of the `P` macro expands to:

```
await sem ≥ (y + 17) ;
sem := sem - (y + 17) ;
```

When translating a macro call, substitution is syntactic in the sense that the meaning of any symbol in the macro definition other than a parameter is the meaning it has in the context of the call. For example, if the body of the macro definition contains a symbol `q` and the macro is called within a "`with (q ∈ ...)`" statement, then the `q` in the macro expansion is the `q` introduced by the `with` statement.

When replacing a macro by its definition, the translation replaces every instance of a macro parameter *id* in an expression within the macro body by the corresponding expression. Every instance includes any uses of *id* as a bound variable, as in the expression

$$[id \in 1 \mathrel{..} N \mapsto \text{FALSE}]$$

The substitution of an expression like $y + 17$ for *id* here will cause a mysterious error when the translation is parsed. When using PlusCal, obey the TLA$^+$ convention of never assigning a new meaning to any identifier that already has a meaning.

3.6 Definitions

An algorithm's expressions can use any operators defined in the TLA$^+$ module before the "`BEGIN TRANSLATION`" line. Since the TLA$^+$ declaration of the algorithm's variables follows that line, the definitions of those operators can't mention any algorithm variables. The PlusCal `define` statement allows you to write TLA$^+$ definitions of operators that depend on the algorithm's global variables. For example, suppose the algorithm begins:

```
--algorithm Test {
  variables x ∈ 1..N ; y ;
  define { zy    ≜ y*(x+y)
           zx(a) ≜ x*(y-a) }
  ...
```

(The symbol "$\triangleq$" is typed "`==`".) The operators *zy* and *zx* can then be used in expressions anywhere in the remainder of the algorithm. Observe that there is no semicolon or other separator between the two definitions. Section 5.11 on page 55 describes how to write TLA$^+$ definitions.

The variables that may appear within the `define` statement are the ones declared in the `variable` statement that immediately precedes it and that follows the algorithm name, as well as the variable *pc* and, if there is a procedure, the variable *stack*. Local process and procedure variables may not appear in the `define` statement. The `define` statement's definitions need not mention the algorithm's variables. You might prefer to put definitions in the `define` statement even when they don't have to go there. However, remember that the `define` statement cannot mention any symbols defined or declared after the "`END TRANSLATION`" line; and the symbols it defines cannot be used before the "`BEGIN TRANSLATION`" line.

Definitions, including ones in a `define` statement, are not expanded in the PlusCal to TLA$^+$ translation. All defined symbols appear in the translation exactly as they appear in the PlusCal code.

3.7 Labels

Various rules for where labels must or may not appear have been introduced above. The complete set of rules are:

- The first statement in the body of a procedure, of a process, or of a uniprocess algorithm must be labeled.
- A `while` statement must be labeled.

- A statement S in a statement sequence must be labeled if it is preceded in that sequence by any of the following:
  - A `call` statement, if S is not a `return` or a `goto`.
  - A `return` statement.
  - A `goto` statement.
  - An `if` or `either` statement that contains a labeled statement, a `goto`, a `call`, or a `return` anywhere within it.
- A macro body and the `do` clause of a `with` statement cannot contain any labeled statements.
- In any control path, a label must come between an assignment to a variable x and any other statement that assigns a value to x. A local variable or parameter of a procedure P is set by a `call P(...)` or `return` statement in P.

The implicit labels `Done` and `Error` cannot be used as actual labels.

3.8 The Translation's Definitions and Declarations

This section lists all the identifiers declared and defined in the TLA$^+$ translation of a PlusCal algorithm. You may need to know what those identifiers are when writing invariants and liveness properties to check the algorithm. Moreover, as explained on page 20 of Section 3.1, TLA$^+$ does not allow the assignment of a new meaning to an identifier that already has a meaning. Redefining an identifier declared or defined by the translation, or using it as a bound variable, will cause a "multiply-defined identifier" error when the TLA$^+$ module is parsed by the SANY parser, which is invoked by the Toolbox.

The translation of a PlusCal algorithm declares the following TLA$^+$ variables:

- Each variable declared either globally or locally within a process or a procedure.
- *pc*
- *stack*, if the algorithm contains one or more procedures.
- Each formal parameter of a procedure.

A multiprocess PlusCal algorithm defines each of the following. For a uniprocess algorithm, the "(*self*)" argument is omitted.

- For a multiprocess algorithm, the set *ProcSet* of all process identifiers.
- The tuple *vars* of all variables.
- The initial predicate *Init*. It contains a conjunct for each variable. The conjuncts for global variables precede those for local procedure and process variables. The conjuncts for the variables declared in a single `variable` statement appear in the order in which they are declared. (This order is significant, since the initial value of a variable can depend on the initial values assigned by previous conjuncts.)
- The next-state action *Next* and the complete specification *Spec*.
- For each statement label *Lab*, an action *Lab*(*self*) if the statement is in a procedure or in a process set; otherwise, an action *Lab*. This action is the TLA$^+$ representation of the atomic operation beginning at that label. (Actions and atomic operations are discussed in Section 5.10.1 on page 52.) If the definition is of *Lab*(*self*), then this is the action describing the operation performed by a process *self*, for *self* in *ProcSet*.
- For each procedure P, an action P(*self*). It is the disjunction of all actions in the procedure executed by a process with identifier *self* in *ProcSet*.
- For each process set named P, an action P(*self*). It is the disjunction of all actions not in a procedure that are executed by a process with identifier *self* in the process set.
- For each single process named P, an action P that is the disjunction of all actions not in a procedure that are executed by the process.

Because TLA$^+$ does not allow an identifier to be declared or defined multiple times, the translation may rename some of these identifiers to produce a legal TLA$^+$ specification. For example, if the PlusCal code declares a variable `x` and also uses `x` as a label, or if it declares `x` as a local variable in two different procedures, then one of the two `x`'s must be renamed. If the translator renames identifiers, then it issues a warning and indicates, in comments placed right after the "`BEGIN TRANSLATION`" line, what renamings have been done.

Identifiers defined or declared in the translation may not be given new meanings in any TLA$^+$ definition that follows the "`END TRANSLATION`" line. For example, if the PlusCal algorithm declares a variable `j`, then a definition that follows the translated algorithm cannot contain the expression

$$\forall i, j \in 1..N \, : \, (i \neq j) \Rightarrow \neg((pc[i] = \text{"cs"}) \land (pc[j] = \text{"cs"}))$$

that redeclares the identifier j. Such a re-use of an identifier causes a "multiply-defined identifier" error when the TLA$^+$ module is parsed.

4 Checking the Algorithm

Sections 2.3–2.5 above tell you how to use the translator and TLC model checker to check an algorithm. This section explains more about the translator and TLC. Only the commonly used features of TLC are described. Consult Chapter 14 of the TLA$^+$ book for a more complete description of what TLC can do. Also, check the document *Current Versions of the TLA$^+$ Tools* on the TLA$^+$ tools web page for recently-added features. That page can be found from the main TLA$^+$ web page, a link to which is at `http://lamport.org`. The Toolbox's *Help* pages also describe many of TLC's features.

4.1 Running the Translator

Running the translator is simple; Section 2.3 on page 8 explains how to do it. Section 2.5 on page 11 describes the translator's `-termination` option. The other options you are likely to use are ones that specify fairness properties; they are described in Section 4.6 on page 38. Appendix Section C on page 67 contains a list of all translator options.

The one part of using the translator that can be tricky is understanding its messages. There are two kinds of translator error messages that can be mysterious. The first is one saying that the translator was expecting to find a certain token and didn't. For example, the missing semicolon at the end of the first line of

```
L1:  a := b + c
L2:  f[x] := c
```

produces the error message

```
-- Expected ";" but found ":="
   line ..., column ....
```

where the line and column numbers indicate the location of the second "`:=`". We might expect the translator to complain when it finds "`b + c`" followed by "`L2`", since no legal expression can begin $b + c\ L2$. However, the translator does not try to parse expressions. It leaves that task to the SANY parser, which is called by the Toolbox. Instead, upon seeing the "`:=`" in the first statement, the translator just assumes that everything until the next reserved symbol is part of the assignment statement's expression. It discovers that something is wrong when it finds the expression ended by "`:=`".

The lesson to be learned from this example is that the source of an error can come well before the location where the error is reported. If you can't find the cause of an error, try narrowing in on it by running the translator with sections of the code commented out. (You can do this by bracketing the code with `(*` and `*)`, even if it contains comments.)

The second class of error that can be mysterious is one caused by omitting a needed label. This is indicated by an error message like

```
    -- Missing label at the following location:  ...
```

Section 3.7 on page 29 gives the rules for where labels are needed. If you are mystified by this message, it may be because you've forgotten that `call` and `return` statements assign values to a procedure's parameters and local variables.

As explained above, some errors in the algorithm are not found by the translator but by SANY, the TLA$^+$ parser. The error can be either in the part of the module that you wrote or in the part written by the translator. The translator does not parse expressions, leaving it to SANY to find most errors in the algorithm's expressions. Clicking on the error message in the Toolbox jumps to and highlights the error in the translation. Executing the Toolbox's *Goto PCal Source* command jumps to the corresponding region of PlusCal. If the source of the problem is not immediately clear, you should be able to figure it knowing that the translation copies your expressions pretty much the way you typed them, except for the following changes.

- Some variables are primed.

- Variables local to a process are turned into functions (arrays) that take an additional argument. For example, in algorithm *FastMutex* of Figure 2 on page 13, each occurrence of the local variable j is replaced by $j[self]$.

- An assignment to an element of a function or record variable is rewritten as an assignment to the variable using the TLA$^+$ EXCEPT construct explained in Section 5.7 on page 49.

- Variables may be renamed, as explained in Section 3.8 on page 30.

If the parser complains that an identifier has been multiply defined, it may mean that you have redefined or used as a bound variable an identifier that is defined or declared in the algorithm's translation. This problem is discussed above in Section 3.8 on page 30.

Occasionally, it may be difficult to figure out the cause of a parsing error. In that case, try inserting a "==$\cdots$==" line to prematurely end the module in different places until you find the definition or statement that is causing the error.

4.2 Specifying the Constants

Most algorithms are written in terms of constant parameters, declared in the TLA$^+$ module with a CONSTANT statement. A model must specify the values of these constants. This is done in the *What is the model?* section of the *Model Overview* page.

There are three kinds of values you can assign to a constant:

- An ordinary value, such as 3000.
- A model value. Making a constant a model value tells TLC to treat it as an uninterpreted symbol that is unequal to any value other than itself.
- A set of model values. For example, setting the constant `Proc` equal to the set `{p1, p2, p3}` of model values tells TLC to let the value of *Proc* be the set $\{p1, p2, p3\}$, where $p1$, $p2$, and $p3$ are considered to be model values (uninterpreted symbols).

See the Toolbox help page for more information about model values.

You can also assign new meanings to defined constants and constant operators for the purpose of model checking. For example, an algorithm might contain a statement

```
with ( i ∈ Nat ) ...
```

where *Nat* is defined by the standard *Naturals* module to be the set of all natural numbers. TLC cannot check an algorithm that requires it to enumerate an infinite set like *Nat*. However, you can use the *Definition Override* section of the model's *Advanced Options* page to tell TLC to substitute a finite set of numbers for *Nat*.

Definition override can also be used to replace a definition with one that is more easily computed by TLC. For example, you might replace the definition of *gcd* on page 11 with an alternative definition that TLC can compute more efficiently. You could use *gcd* in the algorithm because its definition is easy to understand, but speed up the checking by having TLC use another definition.

4.3 Constraints

TLC tries to generate all reachable states of the algorithm. It does this by repeatedly finding all states that can be reached with a single step from a reachable state that it has already found, starting with all possible initial states. It will run forever if there are an infinite number of reachable states.

Some algorithms have infinitely many reachable states because they have counters or queues that can grow without bound. You can limit the reachable states that TLC examines by using a *constraint*, which is an arbitrary Boolean expression. If TLC finds a reachable state s that does not satisfy the constraint, then it will not look for states that can be reached from s. You can specify such a constraint in the *State Constraint* section of the model's *Advanced Options* page. For example, the constraint $x < 17$ causes TLC to find only those reachable states that are either initial states or are reachable by a sequence of states all having x less than 17.

4.4 Understanding TLC's Output

There are two kinds of errors TLC can find: (i) an `assert` statement is executed when its expression is false or some property that you asked TLC to check is not satisfied, or (ii) the algorithm is trying to evaluate a meaningless expression such as $foo.bar$ if foo does not equal a record. In the first case, TLC tells you which assertion or property is violated. In the second, it usually reports the stack of nested expressions it was executing when it found the error; but in some cases it just prints the unhelpful message "`null`".

For any error, TLC produces an error trace of states reached in the execution up to the point at which the error occurred. A state consists of an assignment of values to all the variables. TLC shows most values as ordinary TLA$^+$ expressions, as described in Section 5. However, functions are described in terms of the operators @@ and :> that are defined in the *TLC* module. The expression

$$d_1 :> e_1 \;@@\; d_2 :> e_2 \;@@\; \ldots \;@@\; d_n :> e_n$$

equals the function f with domain $\{d_1, \ldots, d_n\}$ such that $f[d_i] = e_i$ for each i in $1 \mathrel{..} n$.

It can sometimes be quite difficult to figure out the cause of an error from TLC's error message. In that case, you can debug by inserting `print` statements in the algorithm. You can also use the *Print* operator in the algorithm's expressions or in the invariants that TLC is checking. The operator *Print* is defined in the *TLC* module so $Print(pval, val)$ equals val, but TLC prints the value of $pval$ when evaluating it.

TLC reports the location of an error in the TLA$^+$ specification, and you can usually click on the error message and jump to that location. If the location is in the algorithm's translation, the Toolbox's *Goto PCal Source* command will jump to the corresponding part of the PlusCal code.

4.5 Invariance Checking

The examples in Section 2 explain how to use TLC to check invariance of a formula—meaning that the formula is true in all states reached in any execution of the algorithm. An important example of invariance is type correctness. In ordinary typed programming languages, type correctness is a syntactic condition. Because PlusCal is typeless, type correctness is a property of the algorithm, asserting that the value of each variable is an element of the proper set. For example, we say that a variable `p` has type *prime number* iff the value of `p` is always a prime number—in other words, iff the following formula is an invariant, where `Nat` is the set of natural numbers.

```
p ∈ {i ∈ Nat : ∀ j ∈ 2..(i-1) : i % j ≠ 0}
```

(If you don't understand this invariant now, you should after reading Section 5.) TLC can check if this formula is an invariant. Like type checking in ordinary programs, checking type correctness is a good way to find simple errors in a PlusCal algorithm.

For an algorithm to be type correct, the initial values of its variables must be of the right "type". If no initial value is specified for a variable, its default initial value is an unspecified constant named `defaultInitValue`. By default, the Toolbox's models set it to be a model value (see page 35). Since `defaultInitValue` is unspecified, it is not a type-correct value for the variable. The algorithm will therefore not be type correct unless the variable is properly initialized. Among the variables whose type you might want to check are the procedure parameters. An algorithm can assign initial values to a procedure's formal parameters as indicated in this example:

```
procedure  Foo (p1 = 0, p2 = {"a", "b"})
```

Like a procedure variable's declaration, the initial-value declaration of a formal parameter *p* must be of the form *p* = *expression*.

Since a procedure's formal parameters are set equal to the corresponding arguments when the procedure is called, their initial values do not affect the execution. Those initial values serve only to ensure that the corresponding variables in the TLA$^+$ specification always have values of the correct type.

4.6 Termination, Liveness, and Fairness

We saw in Section 2.5 how to check termination of a uniprocess algorithm. Termination is a special case of a general class of properties called *liveness* properties, which assert that something must eventually happen. We can use TLC to check more general liveness properties of an algorithm.

An algorithm satisfies a liveness property only under some assumptions—usually fairness assumptions on actions. In a PlusCal algorithm, there is an action corresponding to each label. Execution of that action consists of executing all code from that label to the next. An action is *enabled* iff it can be executed. Consider the following code within a process:

```
a: y := 42;
   z := y + 1;
b: await x > self;
   x := x-1;
c: ...
```

An execution of action a consists of executing the assignments to y and z. That action is enabled iff control in the process is at a. An execution of action b decrements x by 1. It is enabled iff control is at b and the value of x is greater than the process's identifier *self*.

An action like a that is enabled iff control is at that label is said to be *non-blocking.* An action like b that is not non-blocking is said to be *blocking.*

Fairness for a non-blocking action means that the process cannot stop at that action. Thus, fairness at a means that if control in the process is at a, then the process must eventually execute action a.

There are two kinds of fairness conditions for blocking actions. *Weak* fairness of an action α means that a process cannot halt at α if α remains forever enabled. For example, weak fairness of action b means that if $x > \mathit{self}$ remains true while the process is at b, then action b is eventually executed. *Strong* fairness of α means that, in addition to a process not being able to halt at α if α remains true forever, it can't halt if α keeps being disabled and subsequently enabled. For example, strong fairness of action b implies that process *self* cannot halt at b as long as $x > \mathit{self}$ either remains true or keeps becoming true, even if it also keeps becoming false.

For a non-blocking action, weak and strong fairness are equivalent, so there is only one kind of fairness.

Writing `fair process` instead of just `process` asserts that all actions of the process are, by default, weakly fair. The default fairness condition of an action of the process can be modified by adding `+` of `-` after its label.

Writing `a:+` asserts that action a is strongly fair. Writing `a:-` asserts that action a satisfies no fairness condition.

Writing `fair+ process` process assserts that all actions of the process are strongly fair by default. Adding `-` after a label in the process asserts that the action satisfies no fairness condition. Adding `+` after a label in a `fair+` process has no effect.

A process that is not a `fair` or `fair+` process is called an *unfair* process and has no fairness assumptions on its actions. Adding `+` or `-` after a label in such a process has no effect.

The following translator options affect fairness assumptions.

- `-wf` Makes any unfair process (one not preceded by `fair` or `fair+`) a `fair` process.
- `-sf` Makes any unfair process (one not preceded by `fair` or `fair+`) a `fair+` process.
- `-nof` Makes every process an unfair process.

In addition to fairness of individual algorithm actions, there is also fairness of the entire algorithm. This property asserts that the algorithm cannot halt if it has at least one enabled action. This property is asserted by beginning the algorithm with `--fair algorithm`, or by the `-wfNext` translator option.

For a uniprocess or sequential algorithm, beginning the algorithm with `--fair algorithm` or using the `-wfNext`, `-wf`, or `-sf` option are equivalent. (Without another process, if control is at a blocking action α that is not enabled, then α can never become enabled.) Weak fairness for a sequential algorithm can also be specified by writing `fair` (or `fair+`) before the `{` that begins the algorithm's body. Adding `+` or `-` after a label has no effect for a sequential algorithm.

The `-termination` option, discussed in Section 2.5 on page 11, effectively adds a `-wf` option if none of the fairness options described above is specified.

The liveness properties we want an algorithm to satisfy can be specified as temporal formulas using the TLA$^+$ temporal operators used to express fairness and liveness are described in Section 5.10 on page 51. Temporal properties are subtle and can be hard to understand. Chapter 8 of the TLA$^+$ book discusses these properties in more detail. Here, we just describe one particular operator that is quite useful: the operator $\leadsto$, typed `~>` and pronounced *leads to*.

The formula $P \leadsto Q$ asserts that if P ever becomes true, then Q is true then or will be true at some later. As an example, let's consider algorithm

FastMutex in Figure 2 on page 13. We assume that the processes are fair, so we add the keyword `fair` before the `process`. We allow a process to remain forever inside its noncritical section or its critical section. We therefore add `-` after the labels `ncs:` and `cs:`.

The liveness condition the algorithm satisfies is that, if some process is trying to enter its critical section, then some process (not necessarily the same one) is or eventually enters its critical section. A process i is trying to enter its critical section if $pc[i]$ is in the set $\{$"start", "l1", ... "l10"$\}$. It's easier to express this by saying that $pc[i]$ is *not* in the set $\{$"ncs", "cs", "l11", "l12"$\}$. So, the property we want to check is that this condition leads to some process being in its critical section. This property is writen as

$$\begin{aligned}&(\exists i \in 1 \,..\, N : pc[i] \notin \{\text{"ncs"}, \text{"cs"}, \text{"l11"}, \text{"l12"}\}) \\ &\qquad \leadsto \; (\exists i \in 1 \,..\, N : pc[i] = \text{"cs"})\end{aligned}$$

TLC can check this property for three processes ($N = 3$) in about a minute.

The language constructs and translator options described allow you to express all the fairness assumptions about an algorithm that you're likely to need. In the unlikely event that you want some other kind of fairness assumption, you can write it yourself using the TLA$^+$ temporal operators described in Section 5.10 on page 51.

Liveness checking (including termination) is slower than invariance checking, and TLC cannot check liveness on as large a model as it can check invariance.

4.7 Additional TLC Features

4.7.1 Deadlock Checking

An algorithm is *deadlocked* if it has not terminated, but it can take no further step. A process has terminated if it has reached the end of its code, so control is at the implicit `Done` label that ends its body.

The most likely way for a uniprocess algorithm to deadlock is for a procedure call to "fall off the end" without executing a `return` statement—that is, for it to reach the implicit label `Error` that ends the procedure body.

A multiprocess algorithm is deadlocked if no process can take a step, but some process has not terminated. The usual way for this to happen is for each processes to be waiting, either at an `await` statement whose expression is false or at a statement of the form "`with` $x \in S\ldots$" when S equals the empty set.

Deadlock is normally an error and is reported by TLC. However, sometimes an algorithm is supposed to halt in a state in which not all processes have reached the end of their code. To stop TLC from checking for deadlock, uncheck the *Deadlock* box on the *Model Overview* page.

4.7.2 Multithreading

TLC can execute with multiple threads to take advantage of a multiprocessor computer. You specify the number of threads it should use in the *How to run?* section of the *Model Overview* page. The number you choose should be at most equal to the number of actual processors the computer has. (Running it with more threads than there are processors can slow TLC down.) In theory, using w processors can speed up TLC's computation of the set of reachable states by a factor of almost w. In practice, the speedup depends on the quality of the Java runtime's implementation of multithreading. TLC's algorithm for checking liveness is single-threaded, so additional worker threads will not speed up that part of TLC's execution.

4.7.3 Symmetry

Many algorithms are symmetric in a set of values. For example, the fast mutual exclusion algorithm described in Section 2.6 on page 11 is symmetric in the set of process identifiers. This means that, given any possible execution of the algorithm, permuting the set of identifiers of the processes yields a possible execution. Exactly what symmetry means is explained in Section 14.3.4 on page 245 of the TLA$^+$ book.

When assigning a set of model values to a constant, we can specify that set to be a *symmetry set.* (Model values are explained on page 35.) TLC will then speed up its checking of the algorithm by ignoring any new state it finds that is the same as a state it has already found under a permutation of that set of model values.

TLC can use symmetry only for a set of model values. For TLC to take advantage of the symmetry of algorithm *FastMutex*, that algorithm would have to be rewritten to use an arbitrary set of process identifiers instead of the set of numbers $1 \,..\, N$.

When we instruct TLC to assume that an algorithm is symmetric, it does not check whether the algorithm really is symmetric. That's our responsibility.

Do not tell TLC both to assume symmetry and to check liveness. The interaction of a symmetry assumption with TLC's algorithm for checking liveness is subtle. It's hard to determine if liveness checking will produce correct results when symmetry is assumed.

5 TLA$^+$ Expressions and Definitions

We now describe the TLA$^+$ operators with which PlusCal expressions are built. They are also listed with brief explanations in Tables 1–3 on pages 70–71. Only TLA$^+$ operators that can be evaluated by TLC are given.

We show the typeset versions of all expressions. Table 5 on page 72 shows how symbols with no obvious ASCII representation are typed. TLA$^+$ keywords are typed with upper-case letters, so TRUE is typed as `TRUE`.

If you're not sure about the meaning of some construct, try it out with the Toolbox. You can have the Toolbox print the value of any constant expression by typing it into the *Evaluate Constant Expression* section of a model's *Model Checking Results* page. This will cause TLC to evaluate the expression and print the result in the *Value* box. You can tell TLC not to do any checking of the specification by selecting *No Behavior Spec* in the *What is the behavior spec?* section of the *Model Overview* page. (This will be the default selection if you create a module with no algorithm and no VARIABLE declaration.)

5.1 Numbers

Non-negative integers are typed in the usual way as strings of decimal digits. The standard module *Naturals* defines the following standard operators on integers:

$$+ \quad - \quad * \quad \hat{} \text{ (exponentiation)} \quad < \quad > \quad \leq \quad \geq \quad \% \quad \div \quad ..$$

where $-$ is subtraction, not the unary negation operator. The expression a^b is typed `a^b`. The operator "$..$" is defined so $a\,..\,b$ is the set of all integers c such that $a \leq c \leq b$. The modulus operator % and the integer division operator $\div$ are defined so that, for any integer a and positive integer b, the value of $a \% b$ is in $0\,..\,(b-1)$ and

$$a \;=\; b * (a \div b) \;+\; (a \% b)$$

The *Naturals* module also defines *Nat* to be the set of all natural numbers (non-negative integers).

The *Integers* module defines everything the *Naturals* module does plus the unary "$-$" operator and the set *Int* of all integers. You are unlikely to use *Nat* or *Int* in an algorithm, but you might very well write something like $n \in Int$ in a type-correctness invariant (Section 4.5).

5.2 Strings

Strings are enclosed in double-quotes (`"`), so the string "`abc`" is typed `"abc"`. The following pairs of characters are used to represent certain special characters in strings.

`\"`	`"`	`\t`	tab	`\f`	form feed
`\\`	`\`	`\n`	line feed	`\r`	carriage return

A string is defined in TLA$^+$ to be a sequence of characters, but TLC does not treat them as first-class sequences. TLC treats strings as a primitive data type, except that the operators "$\circ$" (concatenation) and *Len* (length) defined in the standard *Sequences* module work properly on them—for example, TLC knows that "`ab`" $\circ$ "`c`" equals "`abc`" and *Len*("`abc`") equals 3. (These operators are described in Section 5.8 on page 49 below.) Even though sequences in TLA$^+$ are functions, TLC does not regard them as such and it cannot evaluate "`abc`"[2].

Putting the TLA$^+$ comment delimiters `(*` and `*)` inside strings in a PlusCal algorithm is tricky if the algorithm appears in a comment (as it probably does). For example, suppose your algorithm contains the statement

```
x := "the *) delimiter" ;
```

Because this appears inside a comment, the TLA$^+$ parser ignores the double-quotes and considers `*)` to end a comment. To keep both the TLA$^+$ parser and the PlusCal translator happy, you have to write something like this:

```
 \* (*
x := "the *) delimiter" ;
```

A similar trick works for `(*`.

5.3 Boolean Operators

The Boolean values are written TRUE and FALSE. The set BOOLEAN contains these two values. The five propositional operators on Booleans are

$\land$	conjunction (and, typed "`/\`")	$\Rightarrow$	implication (typed "`=>`")
$\lor$	disjunction (or, typed "`\/`")	$\equiv$	equivalence (typed "`<=>`" or "`\equiv`")
$\neg$	negation (not, typed "`~`")		

The four binary operators are defined by the truth tables of Figure 3 on the next page. The operator $\neg$ is defined by

$\neg$TRUE = FALSE $\qquad$ $\neg$FALSE = TRUE

F	G	$F \land G$
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

F	G	$F \lor G$
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

F	G	$F \Rightarrow G$
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	TRUE
FALSE	FALSE	TRUE

F	G	$F \equiv G$
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	TRUE

Figure 3: Truth tables for the binary Boolean operators.

In addition to the usual binary operators $\land$ and $\lor$, TLA$^+$ also allows a bulleted-list notation for conjunctions and disjunctions. For example, the expression

```
∧ A
∧ ∨ B
  ∨ C
∧ D
```

equals $A \land (B \lor C) \land D$. Indentation is used to eliminate parentheses, which can make a complicated formula easier to read. The $\land$ or $\lor$ symbols in a bulleted-list conjunction or disjunction must line up exactly.

Universal and existential quantification over sets of values have the following forms:

$\forall x \in S : P(x)$

The expression that equals TRUE if $P(x)$ equals TRUE for all elements x in the set S, and equals FALSE if $P(x)$ equals FALSE for some x in S. Thus, $\forall n \in 1 \,..\, 3 : f[n] > y$ is equivalent to

$$(f[1] > y) \land (f[2] > y) \land (f[3] > y)$$

$\exists x \in S : P(x)$

The expression that equals TRUE if $P(x)$ equals TRUE for some x in S, and equals FALSE if $P(x)$ equals FALSE for all x in S. Thus, $\exists n \in 1 \,..\, 3 : f[n] > y$ is equivalent to

$$(f[1] > y) \lor (f[2] > y) \lor (f[3] > y)$$

In these expressions, the bound identifier x may not already be defined or declared, and it may not occur in the expression S. In the case of S equal to the empty set $\{\}$, these definitions become

$$\forall\, x \in \{\} : P(x) \;\equiv\; \text{TRUE}$$
$$\exists\, x \in \{\} : P(x) \;\equiv\; \text{FALSE}$$

for any P. TLA$^+$ allows some obvious abbreviations for nested quantifiers. For example,

$$\forall\, x \in S,\, y \in T : F \quad \text{means} \quad \forall\, x \in S : (\forall y \in T : F)$$
$$\exists\, x, y \in S : F \quad \text{means} \quad \exists\, x \in S : (\exists\, y \in S : F)$$

5.4 Sets

Enumerated finite sets are written in the usual way, $\{e_1, \ldots, e_n\}$ being the set containing the elements $e_1, \ldots, e_n$. For example, $\{1 + 1, 2 + 2, 4\}$ is the set containing the two elements 2 and 4. (Remember that an element either is or is not an element of a set; it makes no sense to talk about a set containing multiple copies of an element.) As a special case of this notation, $\{\}$ is the empty set (the set containing no elements). TLA$^+$ provides the following operators on sets.

$\in$ (membership)	$\cup$ (union)	UNION (big $\bigcup$)
$\subseteq$ (subset)	$\cap$ (intersection)	SUBSET (power set)
$\setminus$ (set difference)		

Here are their definitions:

$e \in S$	Equals TRUE if e is an element of the set S and equals FALSE otherwise.
$S \cap T$	The set of elements in both S and T.
$S \cup T$	The set of elements in S or T (or both).
$S \subseteq T$	True iff every element of S is an element of T.
$S \setminus T$	The set of elements in S that are not in T.
UNION S	The union of the elements of S. In other words, a value e is an element of UNION S iff it is an element of an element of S. Mathematicians usually write this as $\bigcup S$.

SUBSET S The set of all subsets of S. Mathematicians sometimes call this the *power set* of S and write it as $\mathcal{P}(S)$ or 2^S.

Mathematicians often describe a set as "the set of all ... such that ...". The following two constructs formalize such a description.

$\{x \in S : P(x)\}$ The subset of S consisting of all elements x satisfying property $P(x)$. For example, the set of all odd natural numbers can be written $\{n \in \mathit{Nat} \,:\, n \,\%\, 2 = 1\}$.

$\{e(x) : x \in S\}$ The set of elements of the form $e(x)$, for all x in the set S. For example, $\{2*n + 1 : n \in 1 \,..\, 100\}$ is the set $\{3, 5, 7, \ldots, 201\}$.

In these expressions, the bound identifier x may not already be defined or declared, and it may not occur in the expression S. The construct $\{e(x) : x \in S\}$ has the same generalizations as $\exists x \in S : F$. For example, $\{e(x, y) : x \in S,\ y \in T\}$ is the set of all elements of the form $e(x, y)$, for x in S and y in T.

The standard module *FiniteSets* defines *Cardinality*(S) to be the cardinality (number of elements in) the finite set S.

The expression CHOOSE $x \in S : P(x)$ is defined to equal some arbitrarily chosen value x in the set S such that $P(x)$ equals TRUE. If no such x exists, then the value of that expression is unspecified, and TLC will report an error when evaluating it. The CHOOSE operator is known to logicians as *Hilbert's* ε. This operator cannot be used to introduce nondeterminism in an algorithm. The PlusCal statement

```
n := CHOOSE i ∈ 1..7 :  TRUE
```

will assign to `n` the same value every time it is executed. That value is some single unspecified integer in the set $1 \,..\, 7$.

5.5 Functions

What programmers call an array, mathematicians call a function. Intuitively, a function f maps each element d in its domain to the value $f[d]$. If f is not a function or d is not in the domain of f, then the meaning of $f[d]$ is not specified and TLC will report an error if it tries to evaluate that expression.

A function is completely specified by its domain and the value of $f[d]$ for every d in its domain. If f is a function, then DOMAIN f is its domain. The expression $[x \in S \mapsto e(x)]$ equals the function f whose domain is S such that $f[d] = e(d)$ for every d in S. For example,

$$[i \in \{1, 2, 3\} \mapsto 2 * i]$$

is the function *twice* with domain $\{1, 2, 3\}$ such that

$$twice[1] = 2 \qquad twice[2] = 4 \qquad twice[3] = 6$$

Using the operators @@ and :> defined in the *TLC* module, this function can also be written

$$(1 :> 1) \;@@\; (2 :> 4) \;@@\; (3 :> 6)$$

For any sets S and T, the expression $[S \rightarrow T]$ is the set of all functions f with domain S such that $f[d]$ is in T for all d in S.

Functions are first-class values, so $f[d]$ can be a function. For example, the function

$$[i \in Nat \mapsto [j \in 1 \,..\, N \mapsto (2 * i) \,\%\, j]]$$

is a function f such that $f[3][x]$ equals $6 \,\%\, x$.

TLA$^+$ also allows functions of multiple arguments. For example,

$$[i \in Nat,\ j \in 1 \,..\, N \mapsto (2 * i) \,\%\, j]$$

is a function g of two arguments such that $g[3, x]$ equals $6 \,\%\, x$. A function with multiple arguments is actually a function with a single argument that is a tuple. For example, $g[3, x]$ is shorthand for $g[\langle 3, x \rangle]$. (Section 5.8 on the next page discusses tuples.)

5.6 Records

TLA$^+$ provides records that are much like the records (also called structs) of ordinary programming languages. If *exp* is a record-valued expression, then *exp.bar* is the *bar* field of that record. The expression

$$[foo \mapsto 17,\ bar \mapsto \{1, 2, 3\}]$$

equals the record r containing a *foo* field whose value is 17 and a *bar* field whose value is the set $\{1, 2, 3\}$. This record r is an element of the set

$$[foo : Nat,\ bar : \text{SUBSET}\ 1 \,..\, 13]$$

consisting of all records with a *foo* field that is an element of the set *Nat* of natural numbers and a *bar* field that is an element of the set SUBSET $1 \,..\, 13$ of all subsets of the set $1 \,..\, 13$.

In TLA$^+$, a record with fields *foo* and *bar* is actually a function whose domain is the set {"foo", "bar"}. The expression *exp.bar* is shorthand for *exp*["bar"].

5.7 The Except Construct

TLA$^+$ provides an EXCEPT construct for describing a function or record that is almost the same as a given function or record. You will probably not need to use the EXCEPT construct yourself. However, it is used extensively in the TLA$^+$ translation of PlusCal programs, so you must know how to interpret it if you want to understand the translation.

If f is a function, then $[f \text{ EXCEPT } ![c] = exp]$ equals the value of f after executing the PlusCal statement $f[c] := exp$. Thus, $[f \text{ EXCEPT } ![c] = exp]$ is the function g that is the same as f except that $g[c] = exp$. This function can also be written

$$[x \in \text{DOMAIN } f \mapsto \text{IF } x = c \text{ THEN } exp \text{ ELSE } f[x]]$$

Similarly, if r is a record, then $[r \text{ EXCEPT } !.c = exp]$ is the record that equals the value of r after executing $r.c := exp$. In other words, it is the record that is the same as r except that its c field equals exp. Since a record is a function whose domain is a set of strings, $[r \text{ EXCEPT } !.c = exp]$ is the same as $[r \text{ EXCEPT } ![\text{"c"}] = exp]$.

A "!" clause of an EXCEPT construct can be more complicated. For example $[f \text{ EXCEPT } ![c].d[e] = exp]$ is the value of f after executing the statement $f[c].d[e] := exp$. You can check that this equals

$$[f \text{ EXCEPT } ![c] = [f[c] \text{ EXCEPT } !.d = [f[c].d \text{ EXCEPT } ![e] = exp]]]$$

An EXCEPT expression can have multiple "!" clauses. For example, the expression $[f \text{ EXCEPT } ![c] = exp1,\ ![d].e = exp2]$ equals the value of f after executing the PlusCal multiple assignment statement

$$f[c] := exp1 \;||\; f[d].e := exp2$$

Remember that a multiple assignment is executed by first evaluating all the right-hand expressions, then performing the assignments from left to right. This implies that $[f \text{ EXCEPT } ![c] = exp1,\ ![d].e = exp2]$ is equal to

$$[\,[f \text{ EXCEPT } ![c] = exp1] \text{ EXCEPT } ![d].e = exp2]\,].$$

5.8 Tuples and Sequences

A finite sequence is what programmers usually call a *list*. In TLA$^+$, a sequence of length n is the same as an n-tuple, which is defined to be a function with domain $1 \,..\, n$. Finite sequences are written in angle brackets $\langle\ \rangle$. The sequence $\langle e_1, \ldots, e_n \rangle$ is the function s with domain $1 \,..\, n$ such that $s[i]$ equals e_i, for each i in $1 \,..\, n$. Thus, $\langle \text{"a"}, \text{"bc"}, \text{"de"} \rangle[3]$ equals "de".

Sets of tuples can be described with the Cartesian product operator $\times$. For example, $\mathit{Nat} \times \mathit{Int} \times \{\text{“a”}, \text{“b”}, \text{“c”}\}$ is the set of all triples $\langle x, y, z \rangle$ such that $x \in \mathit{Nat}$, $y \in \mathit{Int}$, and $z \in \{\text{“a”}, \text{“b”}, \text{“c”}\}$.

The standard module *Sequences* defines the following operators:

$\mathit{Seq}(S)$	The set of all sequences of elements of the set S. For example, $\langle 3, 7 \rangle$ is an element of $\mathit{Seq}(\mathit{Nat})$.
$\mathit{Head}(s)$	The first element of sequence s. For example, $\mathit{Head}(\langle 3, 7 \rangle)$ equals 3.
$\mathit{Tail}(s)$	The tail of sequence s, which consists of s with its head removed. For example, $\mathit{Tail}(\langle 3, 7, \text{“a”} \rangle)$ equals $\langle 7, \text{“a”} \rangle$.
$\mathit{Append}(s, e)$	The sequence obtained by appending element e to the tail of sequence s. For example, $\mathit{Append}(\langle 3, 7 \rangle, 3)$ equals $\langle 3, 7, 3 \rangle$.
$s \circ t$	The sequence obtained by concatenating the sequences s and t. For example, $\langle 3, 7 \rangle \circ \langle 3 \rangle$ equals $\langle 3, 7, 3 \rangle$.
$\mathit{Len}(s)$	The length of sequence s. For example, $\mathit{Len}(\langle 3, 7 \rangle)$ equals 2.

5.9 Miscellaneous Constructs

An IF expression has the form

$$\text{IF } \mathit{bool} \text{ THEN } \mathit{t\_expr} \text{ ELSE } \mathit{e\_expr}$$

If *bool* equals TRUE, then this expression equals *t_expr*; if *bool* equals FALSE, then it equals *e_expr*.

The CASE expression

$$\text{CASE } p_1 \rightarrow e_1 \;\Box\; \ldots \;\Box\; p_n \rightarrow e_n$$

equals some e_i for which p_i equals TRUE. If there is no such p_i, then the value of the expression is unspecified and TLC will report an error when evaluating it. For example, the value of the expression

$$\text{CASE } i \in 1 \,..\, N \rightarrow \text{“a”} \;\Box\; i \in N \,..\, 2 * N \rightarrow \text{“b”}$$

is

- “a” if i is in $1 \,..\, (N - 1)$,
- “b” if i is in $(N + 1) \,..\, 2 * N$,

- either "a" or "b" if $i = N$,
- unspecified if i is not in $1 \mathrel{..} 2 * N$, and TLC reports an error when evaluating it.

In the third case, whether the expression equals "a" or "b", it equals the same value every time it is evaluated.

The CASE expression

$$\text{CASE } p_1 \rightarrow e_1 \;\square\; \ldots \;\square\; p_n \rightarrow e_n \;\square\; \text{OTHER } e$$

is equivalent to

$$\text{CASE } p_1 \rightarrow e_1 \;\square\; \ldots \;\square\; p_n \rightarrow e_n \;\square\; \neg(p_1 \lor \ldots \lor p_n) \rightarrow e$$

Thus, its value equals e if all of the p_i equal FALSE.

A LET expression allows you to make definitions local to the expression. For example,

$$\begin{array}{lll} \text{LET} & x & \triangleq \; a + b \\ & y & \triangleq \; a - b \\ \text{IN} & \multicolumn{2}{l}{\text{IF } y > 0 \text{ THEN } x + y \text{ ELSE } x - y} \end{array}$$

equals

$$\text{IF } a - b > 0 \text{ THEN } (a + b) + (a - b) \text{ ELSE } (a + b) - (a - b)$$

Any sequence of TLA$^+$ definitions can appear between the LET and the IN. Section 5.11 on page 55 describes TLA$^+$ definitions.

5.10 Temporal Operators

A *behavior* is a nonempty sequence of states, where a state is an assignment of values to variables. A behavior of an algorithm is one that can be generated by executing the algorithm.

A *temporal formula* is a predicate on behaviors—in other words, it is true or false for any nonempty sequence of states. An algorithm satisfies a temporal formula F iff F is true of all behaviors of the algorithm.

Temporal formulas cannot appear in a PlusCal algorithm. They are used only in the fairness properties assumed of the algorithm's executions and in the properties asserted about the algorithm. Section 4.6 on page 38 explains how to assert fairness properties of the algorithm and how to tell TLC to check liveness properties. This section defines the TLA$^+$ temporal operators that are used to express these fairness and liveness properties.

The definitions are given for infinite behaviors only. They are applied to finite behaviors by considering the behavior $\langle s_1, \ldots, s_n \rangle$ to be equivalent to the infinite behavior $\langle s_1, \ldots, s_n, s_n, s_n, \ldots \rangle$ obtained by repeating the last state forever.

5.10.1 Fairness

An *atomic operation* of an algorithm consists of all control paths that start at some label l, end at a label, and do not pass through any label. For example, this code sequence

```
L1: if (x = 0) y := y + 1
     else L2: { await sem > 0 ;
                sem := sem - 1 } ;
L3: ...
```

contains the atomic operations

```
L1:  if (x = 0) y := y + 1 else {L2: } ; L3:
```

and

```
L2:  await sem > 0 ; sem := sem - 1 ; L3:
```

We name an atomic operation by the label that begins it, so the second of these atomic operations is called operation `L2`.

In TLA$^+$, an *action* is a formula describing how the state changes. More precisely, it is a formula that is true or false of a pair of states. We say that $s \rightarrow t$ is an A *step* iff action A is true of the pair $\langle s, t \rangle$ of states. An action A is said to be *enabled* in a state s iff it is possible to perform an A action in state s—that is, iff there is some state t such that $s \rightarrow t$ is an A step.

For each atomic operation `L` of a PlusCal algorithm, the TLA$^+$ translation defines an action L that represents the operation—in other words, where $s \rightarrow t$ is an L step iff executing operation `L` in state s can produce state t. We call L an *atomic action* of the algorithm. Appendix Section B on page 60 describes how atomic operations are represented as atomic actions.

Fairness assumptions about the algorithm are expressed with the following fairness assumptions about actions:

Weak Fairness of an action A means that if it remains continuously possible to execute A, then A is eventually executed. Weak fairness of A is expressed by the temporal formula $\mathrm{WF}_{vars}(A)$ (typed `WF_vars(A)`), where *vars* is the tuple of all variables of the algorithm. This formula asserts of a behavior $\langle s_1, s_2, \ldots \rangle$ that, if there is some $i > 0$ such that A is enabled in state s_j, for all $j \geq i$, then $s_j \rightarrow s_{j+1}$ is an A step, for some $j \geq i$.

Strong Fairness of an action A means that if it is repeatedly possible to execute A, even if it is repeatedly impossible to execute A, then A is eventually executed. Strong fairness of A is expressed by the formula $\mathrm{SF}_{vars}(A)$ (typed `SF_vars(A)`). This formula asserts of a behavior $\langle s_1, s_2, \ldots \rangle$ that if A is enabled in infinitely many states s_j, then $s_j \rightarrow s_{j+1}$ is an A step, for some $j > 0$.

Strong fairness of A is stronger than (implies) weak fairness of A. In other words, if $\mathrm{SF}_{vars}(A)$ is true of a behavior σ, then $\mathrm{WF}_{vars}(A)$ is also true of σ.

As an example, let $L2$ be the atomic action corresponding to the atomic operation `L2` above (on the preceding page). Weak fairness of $L2$ means that, if the process is at control point `L2` and *sem* remains positive, then eventually operation `L2` will be executed. Strong fairness of $L2$ means that, if the process is at control point `L2` and *sem* keeps being set to a positive value, even if it keeps being reset to 0, then eventually `L2` will be executed.

A process's next-state action is defined to be the disjunction of all its atomic actions. The most common fairness assumption is weak fairness of each process's next-state action. For a PlusCal algorithm, weak fairness of a process's next-state action is equivalent to the conjunction of weak fairness of each of its atomic actions. Similarly, strong fairness of a process's next-state action in a PlusCal algorithm is equivalent to strong fairness of each of its atomic actions. An algorithm's next-state action is the disjunction of all of its atomic actions. Weak fairness of the algorithm's next-state action means that the algorithm will not halt if it is possible for some process to perform an action.

5.10.2 Liveness

The temporal properties that an algorithm should satisfy are expressed with the temporal operators "$\Box$", "$\Diamond$", and "$\leadsto$", which are defined as follows:

$\Box F$ is true of a behavior σ iff the temporal formula F is true for every suffix of σ. In other words, $\Box F$ is true for a behavior $\langle s_1, s_2, \ldots \rangle$ iff F is true on the behavior $\langle s_i, s_{i+1}, \ldots \rangle$, for all $i > 0$. Hence, if $\Box F$ is true of σ, then F is true of σ. We usually think of $\Box F$ as asserting that F is *always* true.

$\Diamond F$ is true of a behavior σ iff the temporal formula F is true of some suffix of σ. Since σ is a suffix of itself, if F is true of σ then so is $\Diamond F$. We usually think of $\Diamond F$ as asserting that F is *eventually* true.

$F \leadsto G$ asserts of a behavior σ that if τ is any suffix of σ for which F is true, then there is a suffix of τ for which G is true. In other words, $F \leadsto G$ asserts that whenever F becomes true, G must be true then or at some later point in the execution. We usually read $\leadsto$ as "leads to".

Formulas expressing liveness properties are built with these operators and state predicates. A *state predicate* is a formula that is true or false of a state. For example, the state predicate $x > 0$ is true of those states in which the value of x is greater than 0. A state predicate is considered to be the temporal formula that is true of a behavior σ iff it is true of the first state of σ. If P is a state predicate, then $\Box P$ is true of a behavior σ iff P is true of the first state of all suffixes of the behavior—in other words, iff P is true of all states of σ. Hence, the temporal formula $\Box P$ asserts that P is an invariant of an algorithm. The formula $\Diamond P$ asserts of a behavior σ that P is true in some state of σ.

To check that you understand these temporal operators, you can verify that:

- $\Box\Diamond F$ is true of a behavior σ iff F is true for infinitely many suffixes of σ. In particular, if P is a state predicate, then $\Box\Diamond P$ asserts of σ that P is true for infinitely many states of σ.
- $\Diamond\Box F$ is true of a behavior $\langle s_1, s_2, \ldots\rangle$ iff there is some $i > 0$ such that F is true of $\langle s_j, s_{j+1}, \ldots\rangle$, for all $j \geq i$.
- $F \leadsto G$ is equivalent to $\Box(F \Rightarrow \Diamond G)$.

5.10.3 One Algorithm Implementing Another

Instead of just checking that an algorithm satisfies certain properties, you can check that it implements a complete higher-level specification describing what the algorithm is supposed to do. The TLA$^+$ book explains how to write such a specification in TLA$^+$. You can also write the specification as a more abstract PlusCal algorithm. In that case, you will have to show that the algorithm implements its more abstract version under an interface refinement. Interface refinement is explained in Section 10.8 of the TLA$^+$ book. In most cases, the interface refinement will be a simple data refinement.

5.11 TLA$^+$ Definitions

TLA$^+$ allows you to define operators that take zero or more arguments. Ordinary operator definitions have the two forms

$$F \;\triangleq\; expr$$

$$F(p_1, \ldots, p_n) \;\triangleq\; expr$$

(The symbol "$\triangleq$" is typed "`==`".) The identifier F and the formal parameters p_i must not already have a meaning. All identifiers, other than the p_i, that appear in the expression *expr* must already have a meaning. Hence, recursive definitions cannot be written in this way. TLA$^+$ does allow recursive *function* definitions of the form

$$f[x \in S] \;\triangleq\; expr$$

For example, you can define the factorial function by

$$fact[n \in Nat] \;\triangleq\; \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * fact[n-1]$$

TLA$^+$ also permits definitions of binary (infix) operators. For example, the following defines $\oplus$ (typed "`(+)`") to mean addition modulo N:

$$a \oplus b \;\triangleq\; (a + b) \,\%\, N$$

Table 4 on page 71 lists all user-definable operator symbols. (Table 5 on page 72 lists the non-obvious ASCII versions of those symbols.)

Definitions of operators or functions that are used in the PlusCal algorithm must appear in the module before the "`BEGIN TRANSLATION`" line, or in a `define` statement as described in Section 3.6 on page 29. Except for ones in a `define` statement, definitions that use the identifiers declared or defined in the translation must come after the "`END TRANSLATION`" line. Those identifiers are listed in Section 3.8 on page 30.

TLA$^+$ does not use a semicolon or any other delimiter to end a definition. It's customary to start each definition on a new line, but that isn't necessary. Two successive definitions can be separated by any kind of space character or characters.

References

[1] Leslie Lamport. TLA—temporal logic of actions. A web page, a link to which can be found at URL `http://lamport.org`. The page can also be found by searching the Web for the 21-letter string formed by concatenating `uid` and `lamporttlahomepage`.

[2] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

[3] Leslie Lamport. *Specifying Systems*. Addison-Wesley, Boston, 2003. Also available on the Web via a link at `http://lamport.org`.

[4] J. T. Schwartz, R. B. Dewar, E. Schonberg, and E. Dubinsky. *Programming with sets: An Introduction to SETL*. Springer-Verlag, New York, 1986.

[5] Robert Sedgewick. *Algorithms*. Addison-Wesley, 1988.

Appendix

Section A gives the BNF grammar of PlusCal, and Section B describes the TLA$^+$ translation of a PlusCal algorithm.

A The Grammar

The algorithm must appear in a file with a TLA$^+$ module, either outside the module or within a single comment. In the latter case, it will almost surely be enclosed by "`(*`" and "`*)`". Comments within an algorithm are delimited by "`\*`" or "`(*...*)`". The grammar described here is for an algorithm with every comment removed and replaced by one or more spaces.

Here is a simplified BNF grammar for the algorithm text. It does not express the restrictions on where labels must or may not occur, which are explained in Section 3.7 on page 29. It also does not express the restrictions on what statements may occur in the body of a macro—namely, that a `while`, `call`, `return`, `goto`, or macro call may not appear there. The BNF uses the following notations.

- The square brackets "$[$" and "$]$" are BNF grouping symbols (but "`[`" and "`]`" are literals).
- $a \mid b$ means a or b.
- $a^{0,1}$ means 0 or 1 instance of a.
- a^* means 0 or more instances of a.
- a^+ means 1 or more instances of a.

The grammar is:

$$
\begin{aligned}
\langle \mathit{Algorithm} \rangle \ ::= \ & [\,\texttt{--algorithm} \mid \texttt{--fair algorithm}\,]\ \langle \mathit{name} \rangle \\
& \texttt{\{}\ \langle \mathit{VarDecls} \rangle^{0,1} \\
& \quad \langle \mathit{Definitions} \rangle^{0,1} \\
& \quad \langle \mathit{Macro} \rangle^{*} \\
& \quad \langle \mathit{Procedure} \rangle^{*} \\
& \quad [\,\langle \mathit{CompoundStmt} \rangle \mid \langle \mathit{Process} \rangle^{+}\,]\ \texttt{\}}
\end{aligned}
$$

$$
\langle \mathit{Definitions} \rangle ::= \texttt{define}\ \texttt{\{}\ \langle \mathit{Defs} \rangle\ \texttt{\}}\ [\texttt{;}]^{0,1}
$$

$$\begin{aligned}\langle \mathit{Macro}\rangle \ ::=\ & \texttt{macro}\ \langle \mathit{Name}\rangle\ (\ [\langle \mathit{Variable}\rangle\ [\texttt{,}\ \langle \mathit{Variable}\rangle]^*\ ]^{0,1}\ )\\ & \langle \mathit{CompoundStmt}\rangle\ [\texttt{;}]^{0,1}\end{aligned}$$

$$\begin{aligned}\langle \mathit{Procedure}\rangle \ ::=\ & \texttt{procedure}\ \langle \mathit{Name}\rangle\ (\ [\langle \mathit{PVarDecl}\rangle\ [\texttt{,}\ \langle \mathit{PVarDecl}\rangle]^*\ ]^{0,1}\ )\\ & \langle \mathit{PVarDecls}\rangle^{0,1}\\ & \langle \mathit{CompoundStmt}\rangle\ [\texttt{;}]^{0,1}\end{aligned}$$

$$\begin{aligned}\langle \mathit{Process}\rangle \ ::=\ & [\,\texttt{fair}\ [+]^{0,1}\,]^{0,1}\ \texttt{process}\ \texttt{(}\ \langle \mathit{Name}\rangle\ [\texttt{=}\mid \texttt{\textbackslash in}]\ \langle \mathit{Expr}\rangle\ \texttt{)}\\ & \langle \mathit{VarDecls}\rangle^{0,1}\\ & \langle \mathit{CompoundStmt}\rangle\ [\texttt{;}]^{0,1}\end{aligned}$$

$$\begin{aligned}\langle \mathit{VarDecls}\rangle \ &::=\ [\texttt{variable}\mid \texttt{variables}]\ \langle \mathit{VarDecl}\rangle^{+}\\ \langle \mathit{VarDecl}\rangle \ &::=\ \langle \mathit{Variable}\rangle\ [\,[\texttt{=}\mid \texttt{\textbackslash in}]\ \langle \mathit{Expr}\rangle\,]^{0,1}\ [\texttt{;}\mid\texttt{,}]\\ \langle \mathit{PVarDecls}\rangle \ &::=\ [\texttt{variable}\mid \texttt{variables}]\ [\langle \mathit{PVarDecl}\rangle\ [\texttt{;}\mid\texttt{,}]\ ]^{+}\\ \langle \mathit{PVarDecl}\rangle \ &::=\ \langle \mathit{Variable}\rangle\ [\,\texttt{=}\ \langle \mathit{Expr}\rangle\,]^{0,1}\end{aligned}$$

$$\langle \mathit{CompoundStmt}\rangle \ ::=\ \texttt{\{}\ \langle \mathit{Stmt}\rangle\ [\texttt{;}\ \langle \mathit{Stmt}\rangle]^*\ [\texttt{;}]^{0,1}\ \texttt{\}}$$

$$\langle \mathit{Stmt}\rangle \ ::=\ [\langle \mathit{Label}\rangle\ \texttt{:}\ [\texttt{+}\mid\texttt{-}]^{0,1}\,]^{0,1}\ \ [\langle \mathit{UnlabeledStmt}\rangle\mid\langle \mathit{CompoundStmt}\rangle]$$

$$\begin{aligned}\langle \mathit{UnlabeledStmt}\rangle \ ::=\ & \langle \mathit{Assign}\rangle\mid\langle \mathit{If}\rangle\mid\langle \mathit{While}\rangle\mid\langle \mathit{Either}\rangle\mid\langle \mathit{With}\rangle\mid\\ & \quad\langle \mathit{Await}\rangle\mid\langle \mathit{Print}\rangle\mid\langle \mathit{Assert}\rangle\mid\langle \mathit{Skip}\rangle\mid\langle \mathit{Return}\rangle\mid\\ & \quad\langle \mathit{Goto}\rangle\mid\langle \mathit{Call}\rangle\mid\langle \mathit{MacroCall}\rangle\end{aligned}$$

$$\langle \mathit{Assign}\rangle \ ::=\ \langle \mathit{LHS}\rangle\ \texttt{:=}\ \langle \mathit{Expr}\rangle\ [\ \texttt{||}\ \langle \mathit{LHS}\rangle\ \texttt{:=}\ \langle \mathit{Expr}\rangle]^*$$

$$\langle \mathit{LHS}\rangle \ ::=\ \langle \mathit{Variable}\rangle\ [\texttt{[}\ \langle \mathit{Expr}\rangle\ [\texttt{,}\ \langle \mathit{Expr}\rangle]^*\ \texttt{]}\mid\texttt{.}\ \langle \mathit{Field}\rangle]^*$$

$$\langle \mathit{If}\rangle \ ::=\ \texttt{if}\ \texttt{(}\ \langle \mathit{Expr}\rangle\ \texttt{)}\ \langle \mathit{Stmt}\rangle\ [\texttt{else}\ \langle \mathit{Stmt}\rangle]^{0,1}$$

$$\langle \mathit{While}\rangle \ ::=\ \texttt{while}\ \texttt{(}\ \langle \mathit{Expr}\rangle\ \texttt{)}\ \langle \mathit{Stmt}\rangle$$

$$\langle \mathit{Either}\rangle \ ::=\ \texttt{either}\ \langle \mathit{Stmt}\rangle\ \ [\texttt{or}\ \langle \mathit{Stmt}\rangle\,]^{+}$$

$$\begin{aligned}\langle \mathit{With}\rangle \ ::=\ & \texttt{with}\ \texttt{(}\ \langle \mathit{Variable}\rangle\ \ [\,\texttt{=}\mid\texttt{\textbackslash in}\,]\ \ \langle \mathit{Expr}\rangle\\ & \quad [\,[\texttt{;}\mid\texttt{,}]\ \ \langle \mathit{Variable}\rangle\ \ [\,\texttt{=}\mid\texttt{\textbackslash in}\,]\ \ \langle \mathit{Expr}\rangle\ \ ]^*\ \ [\texttt{;}\mid\texttt{,}]^{0,1}\\ & \texttt{)}\ \langle \mathit{Stmt}\rangle\end{aligned}$$

$$\langle \mathit{Await}\rangle \ ::=\ [\,\texttt{await}\mid\texttt{when}\,]\ \langle \mathit{Expr}\rangle$$

$$\langle \mathit{Print}\rangle \ ::=\ \texttt{print}\ \langle \mathit{Expr}\rangle$$

$$\langle \mathit{Assert}\rangle \ ::=\ \texttt{assert}\ \langle \mathit{Expr}\rangle$$

$$\langle \mathit{Skip}\rangle \ ::=\ \texttt{skip}$$

$\langle\mathit{Return}\rangle$ ::= `return`

$\langle\mathit{Goto}\rangle$ ::= `goto` $\langle\mathit{Label}\rangle$

$\langle\mathit{Call}\rangle$ ::= `call` $\langle\mathit{MacroCall}\rangle$

$\langle\mathit{MacroCall}\rangle$ ::= $\langle\mathit{Name}\rangle$ `(` $[\langle\mathit{Expr}\rangle$ [`,` $\langle\mathit{Expr}\rangle]^{*}]^{0,1}$ `)`

$\langle\mathit{Variable}\rangle$ ::= A TLA$^+$ identifier that is not a PlusCal reserved word and is not *pc*, *stack*, or *self*.

$\langle\mathit{Field}\rangle$ ::= A TLA$^+$ record-component label.

$\langle\mathit{Name}\rangle$ ::= A TLA$^+$ identifier that is not a PlusCal reserved word.

$\langle\mathit{Label}\rangle$ ::= A TLA$^+$ identifier that is not a PlusCal reserved word and is not `Done` or `Error`.

$\langle\mathit{Expr}\rangle$ ::= A TLA$^+$ expression not containing a PlusCal reserved word or symbol.

$\langle\mathit{Defs}\rangle$ ::= A sequence of TLA$^+$ definitions not containing a PlusCal reserved word or symbol.

TLA$^+$ expressions and definitions are described in Section 5. A TLA$^+$ record-component label is any sequence of letters, digits, and "`_`" characters containing at least one non-digit and not equal to "`WF_`" or "`SF_`". A TLA$^+$ identifier is a record-component that is not one of the following.

```
ASSUME  ASSUMPTION  AXIOM  CASE  CHOOSE  CONSTANT
CONSTANTS  DOMAIN  ELSE  ENABLED  EXCEPT  EXTENDS  IF  IN
INSTANCE  LET  LOCAL  MODULE  OTHER  UNION  SUBSET  THEN
THEOREM  UNCHANGED  VARIABLE  VARIABLES  WITH
```

The PlusCal reserved words are

```
assert  await  begin  call  define  do  either  else  elsif
end  goto  if  macro  or  print  procedure  process  return
skip  then  variable  variables  when  while  with
```

(Some of these are keywords in the p-syntax that are not used in the c-syntax.) The PlusCal reserved symbols are "`:=`" and "`||`".

B The TLA$^+$ Translation

B.1 The *FastMutex* Algorithm

The TLA$^+$ translation is described with the example algorithm of *FastMutex* in Figure 2 on page 13. I have simplified the translation a bit to make it easier to read, but I have not altered its meaning. To help you understand the translation of your own algorithm, you can try executing the Toolbox's *Goto PCal Source* command on any region of the translation.

The translation begins

$$\textsc{constant}\ \ defaultInitValue$$

This declares *defaultInitValue* to be an unspecified constant. By default, a Toolbox model makes *defaultInitValue* a model value. (The declaration of *defaultInitValue* is omitted if every variable is explicitly initialized.)

The translation next declares the algorithm's variables and defines *vars* to be the tuple of all these variables.

$$\begin{array}{l}\textsc{variables}\ x,\ y,\ b,\ j,\ pc \\ vars\ \triangleq\ \langle x,\ y,\ b,\ j,\ pc\rangle\end{array}$$

The variable *pc* is added to describe the control state. If an algorithm contains one or more procedures, a variable *stack* is added to hold the calling stack. In addition, each formal parameter and local variable of a procedure is declared to be a variable.

Had the algorithm contained a `define` statement, the translation would have contained two VARIABLES statements, the first declaring the variables x, y, b, and pc, and the second declaring the remaining variable j. The definitions from the `define` statement would have been put between the two VARIABLES statements.

For a multiprocess program, the translation next defines the set *ProcSet* of all process identifiers.

$$ProcSet\ \triangleq\ 1\,..\,N$$

Next is the definition of the initial predicate *Init* that specifies the initial values of all the declared variables. Comments indicate if the variables are global or local to a process or procedure.

$$\begin{array}{ll}Init\ \triangleq & \text{Global variables} \\ & \land\ x = defaultInitValue \\ & \land\ y = 0\end{array}$$

$$\begin{array}{l}\land\ b = [i \in 1\,..\,N \mapsto \text{FALSE}] \\ \text{Process Proc} \\ \land\ j \ \ = [\mathit{self} \in 1\,..\,N \mapsto \{\}] \\ \land\ pc = [\mathit{self} \in \mathit{ProcSet} \mapsto \text{“ncs”}]\end{array}$$

Observe that the process-local variables and the variable pc are made functions with domain equal to the appropriate set of process identifiers.

Next come the action definitions. As explained in Section 5.10.1 on page 52, a TLA$^+$ action is a formula describing a pair of states—the state before executing the action and the state after executing it. In an action, unprimed variables refer to their values before executing the action and the primed variables refer to their values after the execution.

The translation defines an action for each atomic operation of the algorithm. As explained in Section 5.10.1, an atomic operation begins at a label that is used to name the action. The first such action definition is generated by statement *ncs*. The definition is parameterized by the identifier *self*, which represents the current process's identifier. (The EXCEPT construct is explained in Section 5.7 on page 49.)

$$\mathit{ncs}(\mathit{self}) \triangleq \begin{array}[t]{l}\land\ pc[\mathit{self}] = \text{“ncs”} \\ \land\ pc' = [pc \text{ EXCEPT } ![\mathit{self}] = \text{“start”}] \\ \land\ \text{UNCHANGED } \langle x,\ y,\ b,\ j\rangle\end{array}$$

The conjunct $pc[\mathit{self}] =$ “ncs” is an *enabling condition*, meaning that the action can be executed only when it is true. It asserts that control of process *self* is at label *ncs*. The action sets $pc[\mathit{self}]$ to “start”; the UNCHANGED conjunct asserts that the values of all other variables are not changed. The evaluation of the `while` test does not appear explicitly in the action because it equals TRUE. The `skip` statement similarly does not appear because it has no effect.

The atomic actions corresponding to the statements labeled *start* and $l1$ are analogous.

$$\mathit{start}(\mathit{self}) \triangleq \begin{array}[t]{l}\land\ pc[\mathit{self}] = \text{“start”} \\ \land\ b' = [b \text{ EXCEPT } ![\mathit{self}] = \text{TRUE}] \\ \land\ pc' = [pc \text{ EXCEPT } ![\mathit{self}] = \text{“l1”}] \\ \land\ \text{UNCHANGED } \langle x,\ y,\ j\rangle\end{array}$$

$$l1(\mathit{self}) \triangleq \begin{array}[t]{l}\land\ pc[\mathit{self}] = \text{“l1”} \\ \land\ x' = \mathit{self} \\ \land\ pc' = [pc \text{ EXCEPT } ![\mathit{self}] = \text{“l2”}] \\ \land\ \text{UNCHANGED } \langle y,\ b,\ j\rangle\end{array}$$

Action $l2$ performs the `if` statement's test and the subsequent transfer of control.

$$\begin{array}{lll} l2(\mathit{self}) & \triangleq & \land\, pc[\mathit{self}] = \text{“l2”} \\ & & \land\, \text{IF } y \neq 0 \text{ THEN } pc' = [pc \text{ EXCEPT } ![\mathit{self}] = \text{“l3”}] \\ & & \qquad\qquad\quad \text{ELSE } pc' = [pc \text{ EXCEPT } ![\mathit{self}] = \text{“l5”}] \\ & & \land\, \text{UNCHANGED } \langle x,\, y,\, b,\, j \rangle \end{array}$$

The body of the `then` clause of statement $l2$ consists of two atomic actions.

$$\begin{array}{lll} l3(\mathit{self}) & \triangleq & \land\, pc[\mathit{self}] = \text{“l3”} \\ & & \land\, b' = [b \text{ EXCEPT } ![\mathit{self}] = \text{FALSE}] \\ & & \land\, pc' = [pc \text{ EXCEPT } ![\mathit{self}] = \text{“l4”}] \\ & & \land\, \text{UNCHANGED } \langle x,\, y,\, j \rangle \end{array}$$

$$\begin{array}{lll} l4(\mathit{self}) & \triangleq & \land\, pc[\mathit{self}] = \text{“l4”} \\ & & \land\, y = 0 \\ & & \land\, pc' = [pc \text{ EXCEPT } ![\mathit{self}] = \text{“start”}] \\ & & \land\, \text{UNCHANGED } \langle x,\, y,\, b,\, j \rangle \end{array}$$

The expression $y = 0$ of the `await` statement is an enabling condition of action $l4$. (Recall that this means the action can be executed only when $y = 0$ is true. The conjunct $pc[\mathit{self}] = \text{“l4”}$ is the other enabling condition of this action.)

Actions $l5$ and $l6$ introduce nothing new and their definitions are omitted here. Action $l7$ shows that the process's local variable j has been turned into an array indexed by self.

$$\begin{array}{lll} l7(\mathit{self}) & \triangleq & \land\, pc[\mathit{self}] = \text{“l7”} \\ & & \land\, b' = [b \text{ EXCEPT } ![\mathit{self}] = \text{FALSE}] \\ & & \land\, j' = [j \text{ EXCEPT } ![\mathit{self}] = 1] \\ & & \land\, pc' = [pc \text{ EXCEPT } ![\mathit{self}] = \text{“l8”}] \\ & & \land\, \text{UNCHANGED } \langle x,\, y \rangle \end{array}$$

Action $l8$ shows how a `while` statement whose test is not identically true is translated.

$$\begin{array}{lll} l8(\mathit{self}) & \triangleq & \land\, pc[\mathit{self}] = \text{“l8”} \\ & & \land\, \text{IF } j[\mathit{self}] \leq N \\ & & \qquad \text{THEN } \land\, \neg b[j[\mathit{self}]] \\ & & \qquad\qquad\quad \land\, j' = [j \text{ EXCEPT } ![\mathit{self}] = j[\mathit{self}] + 1] \\ & & \qquad\qquad\quad \land\, pc' = [pc \text{ EXCEPT } ![\mathit{self}] = \text{“l8”}] \\ & & \qquad \text{ELSE } \;\; \land\, pc' = [pc \text{ EXCEPT } ![\mathit{self}] = \text{“l9”}] \end{array}$$

$$\begin{aligned}&\qquad\qquad \land \text{UNCHANGED } j\\ &\quad \land \text{UNCHANGED } \langle x,\ y,\ b\rangle\end{aligned}$$

Actions $l9$, cs, and $l11$ are obtained in a similar manner and are omitted. Actions $l10$ and $l12$ show the translation of an explicit `goto` and the transfer of control at the end of the `while` loop.

$$\begin{aligned}l10(\mathit{self}) \triangleq\ & \land pc[\mathit{self}] = \text{“l10”}\\ & \land y = 0\\ & \land pc' = [pc \text{ EXCEPT } ![\mathit{self}] = \text{“start”}]\\ & \land \text{UNCHANGED } \langle x,\ y,\ b,\ j\rangle\end{aligned}$$

$$\begin{aligned}l12(\mathit{self}) \triangleq\ & \land pc[\mathit{self}] = \text{“l12”}\\ & \land b' = [b \text{ EXCEPT } ![\mathit{self}] = \text{FALSE}]\\ & \land pc' = [pc \text{ EXCEPT } ![\mathit{self}] = \text{“ncs”}]\\ & \land \text{UNCHANGED } \langle x,\ y,\ j\rangle\end{aligned}$$

The translation next defines $\mathit{Proc}(\mathit{self})$ to be the next-state action of process self of process set Proc, which is the disjunction of all the atomic actions of the process. (The name of a process or process set is used only to name the process's next-state action.) A step of the process is one that satisfies formula $\mathit{Proc}(\mathit{self})$.

$$\begin{aligned}\mathit{Proc}(\mathit{self}) \triangleq\ & \mathit{ncs}(\mathit{self}) \lor \mathit{start}(\mathit{self}) \lor l1(\mathit{self}) \lor l2(\mathit{self}) \lor l3(\mathit{self})\\ &\quad \lor l4(\mathit{self}) \lor l5(\mathit{self}) \lor l6(\mathit{self}) \lor l7(\mathit{self}) \lor l8(\mathit{self})\\ &\quad \lor l9(\mathit{self}) \lor l10(\mathit{self}) \lor cs(\mathit{self}) \lor l11(\mathit{self})\\ &\quad \lor l12(\mathit{self})\end{aligned}$$

The action Next is defined to be the next-state action of the entire algorithm. It is the disjunction of the next-state actions of all the processes. (The existential quantification is equivalent to the disjunction $\mathit{Proc}(1) \lor \ldots \lor \mathit{Proc}(N)$.)

$$\begin{aligned}\mathit{Next} \triangleq\ & \lor \exists\, \mathit{self} \in 1 \,..\, N : \mathit{Proc}(\mathit{self}))\\ & \lor\ \text{Disjunct to prevent deadlock on termination}\\ & \quad \land \forall\, \mathit{self} \in \mathit{ProcSet} : pc[\mathit{self}] = \text{“Done”}\\ & \quad \land \text{UNCHANGED } \mathit{vars}\end{aligned}$$

The last disjunct of Next is added for TLC's benefit. TLC has no way of distinguishing deadlock from termination, which is simply a desired form of deadlock. The translation therefore adds a disjunction to Next that allows a terminated algorithm to perform a step that does nothing. This transforms termination into an infinite no-op loop, so it is not reported as deadlock by

TLC.[3] Since the *FastMutex* algorithm cannot terminate, the disjunct serves no function in this case. However, the translator is not clever enough to notice that the disjunct is unnecessary. The `-noDoneDisjunct` translator option always suppresses this extra disjunct.

The translator next defines formula *Spec* to be the complete TLA specification of the algorithm. The formula will mean nothing to you if you don't know TLA, but that doesn't matter. You don't need to understand TLA to use PlusCal. If no liveness or termination option is specified, the definition of *Spec* is

$$Spec \triangleq Init \land \Box[Next]_{vars}$$

Had we used a `fair process` statement or the `-wf` option to specify weak fairness of the processes' actions, the definition of *Spec* would have the additional conjunct

$$\forall \, self \in 1 \,..\, N : \mathrm{WF}_{vars}(Proc(self))$$

Had we added `-` after the labels `ncs:` and `cs:`, this conjunct would have become

$$\forall \, self \in 1 \,..\, N : \mathrm{WF}_{vars}((pc[self] \notin \{\text{"ncs"}, \text{"cs"}\}) \land Proc(self))$$

Adding a `+` after the label `l4:` would have changed this conjunct to

$$\forall \, self \in 1 \,..\, N : \begin{array}{l} \land \mathrm{WF}_{vars}((pc[self] \notin \{\text{"ncs"}, \text{"cs"}\}) \land Proc(self)) \\ \land \mathrm{SF}_{vars}(l4(self)) \end{array}$$

If the algorithm had procedures, then $Proc(self)$ would have been defined to be the disjunction only of the atomic actions in the process body. For each procedure P called by the process, the translation would have defined action $P(self)$ to be the disjunction of the atomic actions in the body of the procedure, and it would have conjoined to *Spec* the fairness property

$$\forall self \in 1 \,..\, N \, : \, \mathrm{WF}_{vars}(P(self))$$

Any `+` and `-` label modifiers in the procedure would be handled analogously to the way they are handled in the body of the process.

Had we specified strong fairness of the process, either with a `fair+ process` statement or the `-sf` translator option, then the translation would have been the same except with WF replaced everywhere by SF, and the (now redundant) conjunct $\mathrm{SF}_{vars}(l4(self))$ eliminated.

Finally, the translation defines the temporal formula *Termination* that asserts the property that the algorithm eventually terminates:

[3] If you are familiar with TLA, then you will realize that adding this disjunct does not change the meaning of the specification, just the way TLC checks it.

```
--algorithm Procedures {
  procedure P(pA = 11, pB = 12)
    variables pv = 0 ;
    { LP1 : pv := 1 ;
            call Q("ProcP") ;
      LP2 : return }

  procedure Q(qA = 13)
    variables qv1 = 1; qv2 = 2;
    { LQ1: if (qA = "Mn") { qv1 := 9 ;
                            call P("a", "b") }
           else print stack ;
      LQ2: return }

  { LM: call Q("Mn") } }
```

Figure 4: An algorithm illustrating procedure calls.

$$Termination \triangleq \Diamond(\forall\, self \in ProcSet : pc[self] = \text{“Done”})$$

B.2 Procedures

The *FastMutex* algorithm does not show how procedure calls and returns are translated. Their translation models a standard implementation using a call stack that is represented by the variable *stack*. For a multiprocess algorithm, the value of *stack* is an array of individual stacks, indexed by process identifier. You probably don't care exactly how procedure calls and returns are represented in TLA$^+$; if you do, you can just look at the translation of an algorithm containing them. However, you may need to understand how to read the value of *stack* when debugging your algorithm. This is explained with the sample algorithm of Figure 4 on this page.

An execution of the algorithm calls procedure Q. The execution of Q calls procedure P, and that execution of P calls Q. The execution of Q following the last call prints the value of *stack* and returns. The other two procedure executions then return and the algorithm terminates.

The value of *stack* is thus printed in an execution of Q inside an execution of P inside an execution of Q. Executing the algorithm prints the following value of *stack*. (The notation for writing records is explained in Section 5.6 on page 5.6.)

$$\begin{aligned}
\langle\, &[qA \mapsto \text{“Mn”},\ qv1 \mapsto 9,\ qv2 \mapsto 2, pc \mapsto \text{“LP2”},\ \ \mathit{procedure} \mapsto \text{“Q”}]\,,\\
&[pA \mapsto 11,\ pB \mapsto 12,\ pv \mapsto 0,\qquad pc \mapsto \text{“LQ2”},\ \ \mathit{procedure} \mapsto \text{“P”}]\,,\\
&[qA \mapsto 13,\qquad qv1 \mapsto 1,\ qv2 \mapsto 2, pc \mapsto \text{“Done”}, \mathit{procedure} \mapsto \text{“Q”}]\ \rangle
\end{aligned}$$

The value is a sequence of three records, one for each procedure being executed. The innermost procedure execution produced the first of these records. The *procedure* field shows that the algorithm is executing a call of Q, and the pc field shows that this execution will return to the statement labeled $LP2$. The remaining components show the values of the procedure's parameter qA and its local variables $qv1$ and $qv2$ when the procedure was called. The corresponding variables will be restored to those values upon the next return from procedure Q.

The second record in the sequence *stack* contains the same information for the call of procedure P. Since this was the first call of P in the call stack, the parameters pA and pB and the local variable pv contained their initial values.

C Translator Options

Some translator options can be specified in the module file with an `options` statement. It has the form

`PlusCal options (` *list of options* `)`

where the items in the list may be (but need not be) separated by commas. In the `options` statement, the "`-`" in the option name may be omitted. Any options may be specified in the Toolbox as follows. In the Spec Explorer (which can be raised from the *File* menu), right-click on the specification and select *Properties*. Put the list of options, with "`-`" in the option names and without commas, in the *PlusCal call arguments* field.

Options That May Appear in an `options` Statement

`-wf` Change all unfair processes to weakly fair (`fair`) processes.

`-sf` Change all unfair processes to strongly fair (`fair+`) processes.

`-wfNext` Conjoin to specification *Spec* weak fairness of the algorithm's entire next-state action

`-nof` Conjoin no fairness assumption to specification *Spec*.

`-termination` If no fairness option is selected, it is equivalent to selecting the `-wf` option. When it appears in an `option` statement in a specification's root module, it causes new Toolbox models to specify termination checking. When used in command-line mode (see below), it adds to the *.cfg* file a command to cause TLC to check for termination.

`-noDoneDisjunct` Suppress the extra disjunct that the translation normally adds to the next-state action *Next* to prevent TLC from thinking that the algorithm has deadlocked when it terminates.

`-label` Tells the translator to add missing labels. The translator will add the minimum set of labels needed to satisfy the labeling rules given in Section 3.7 on page 29. This is the default for a uniprocess algorithm in which the user has typed no labels.

`-labelRoot` *name* If the translator adds missing labels, this causes it to use the prefix *name* for the added labels, so the labels will be *name*1, *name*2, etc. The default prefix is *Lbl_*.

`-lineWidth` The translator tries to perform the translation so lines have this maximum width. (It will often fail.) The default value is 78; the minimum value is 60.

Options That Can be Specified Using the *Spec Explorer*

`-spec` *name* This option is used to have the translation performed by executing a TLA$^+$ specification instead of by the translator itself. It causes the translator to write to the file *AST.tla* in the current directory a TLA$^+$ module that defines *ast* to equal a TLA$^+$ representation of the algorithm's abstract syntax tree (AST) and *fairness* to equal the fairness option. (The current directory is the one from which the translator is run.) The translator then copies the files *name.tla* and *name.cfg* from its own internal directory to the current directory, runs TLC on *name.tla*, and uses TLC's output to produce the translation.

The TLA$^+$ representation of an algorithm's AST does not capture formatting information, so this translation will not work on any algorithm containing an expression with bulleted lists of conjuncts or disjuncts. Also, the translator does not attempt to format the output, so the translation is difficult to read. This option is mainly of use to people interested in the formal specification of PlusCal. However, if you suspect that an error is caused by a bug in the translator, you might try using this option to see if the error occurs when the translation is performed by the TLA$^+$ specification.

Currently, there is only a single specification of the translator available, called by letting *name* be *PlusCal*.

`-myspec` *name* Like `-spec`, except the translator uses the files *names.tla* and *names.cfg* in the current directory, instead of copying them from its own directory.

`-writeAST` This causes the translator to write the file *AST.tla* as in the `-spec` option, but not to perform the translation.

Options for Command-Line Use Only

The PlusCal translator can be run outside the Toolbox, from a command line. It then produces a configuration (`.cfg`) file that can be used to run TLC from the command line. The following options make sense only when the translator is run from a command line.

`-unixEOL` When you view the files written by the translator in a text editor, you may find a `^M` at the end of every line. This option will force the translator to use the Unix end-of-line convention, which should remove those `^M`s.

`-help` Type a help file instead of doing a translation.

`-nocfg` Suppress writing of the *.cfg* file.

`-reportLabels` Tells the translator to print the names and locations of all labels it adds. Like `-label`, it causes the translator to add missing labels.

`-debug` Runs the translator in debugging mode. Currently, this does nothing useful.

Logic

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

TRUE FALSE BOOLEAN [the set $\{$TRUE, FALSE$\}$]

$\forall x \in S : p$ (1) $\exists x \in S : p$ (1)

CHOOSE $x \in S : p$ [An x in S satisfying p]

Sets

$= \quad \neq \quad \in \quad \notin \quad \cup \quad \cap \quad \subseteq \quad \setminus$ [set difference]

$\{e_1, \ldots, e_n\}$	[Set consisting of elements e_i]
$\{x \in S : p\}$ (2)	[Set of elements x in S satisfying p]
$\{e : x \in S\}$ (1)	[Set of elements e such that x in S]
SUBSET S	[Set of subsets of S]
UNION S	[Union of all elements of S]

Functions

$f[e]$	[Function application]
DOMAIN f	[Domain of function f]
$[x \in S \mapsto e]$ (1)	[Function f such that $f[x] = e$ for $x \in S$]
$[S \rightarrow T]$	[Set of functions f with $f[x] \in T$ for $x \in S$]
$[f$ EXCEPT $![e_1] = e_2]$ (3)	[Function $\widehat{f}$ equal to f except $\widehat{f}[e_1] = e_2$]

Records

$e.h$	[The h-field of record e]
$[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$	[The record whose h_i field is e_i]
$[h_1 : S_1, \ldots, h_n : S_n]$	[Set of all records with h_i field in S_i]
$[r$ EXCEPT $!.h = e]$ (3)	[Record $\widehat{r}$ equal to r except $\widehat{r}.h = e$]

Tuples

$e[i]$	[The i^{th} component of tuple e]
$\langle e_1, \ldots, e_n \rangle$	[The n-tuple whose i^{th} component is e_i]
$S_1 \times \ldots \times S_n$	[The set of all n-tuples with i^{th} component in S_i]

(1) $x \in S$ may be replaced by a comma-separated list of items $v \in S$, where v is either a comma-separated list or a tuple of identifiers.

(2) x may be an identifier or tuple of identifiers.

(3) $![e_1]$ or $!.h$ may be replaced by a comma separated list of items $!a_1 \cdots a_n$, where each a_i is $[e_i]$ or $.h_i$.

Table 1: The operators of TLA$^+$.

IF p THEN e_1 ELSE e_2	[e_1 if p true, else e_2]
CASE $p_1 \rightarrow e_1 \;\Box\; \ldots \;\Box\; p_n \rightarrow e_n$	[Some e_i such that p_i true]
CASE $p_1 \rightarrow e_1 \;\Box\; \ldots \;\Box\; p_n \rightarrow e_n \;\Box$ OTHER $\rightarrow e$	[Some e_i such that p_i true, or e if all p_i are false]
LET $d_1 \triangleq e_1 \;\ldots\; d_n \triangleq e_n$ IN e	[e in the context of the definitions]

$$\begin{array}{ll} \land\; p_1 & \text{[the conjunction } p_1 \land \ldots \land p_n\text{]} \\ \quad\vdots & \\ \land\; p_n & \end{array} \qquad \begin{array}{ll} \lor\; p_1 & \text{[the disjunction } p_1 \lor \ldots \lor p_n\text{]} \\ \quad\vdots & \\ \lor\; p_n & \end{array}$$

Table 2: Miscellaneous constructs.

$\Box F$	[F is always true]
$\Diamond F$	[F is eventually true]
$\mathrm{WF}_e(A)$	[Weak fairness for action A]
$\mathrm{SF}_e(A)$	[Strong fairness for action A]
$F \leadsto G$	[F leads to G]

Table 3: Temporal operators.

$+$ (1)	$-$ (1)	$*$ (1)	$/$ (2)	$\circ$ (3)	$++$
$\div$ (1)	$\%$ (1)	$\hat{}$ (1,4)	$..$ (1)	$\ldots$	$--$
$\oplus$ (5)	$\ominus$ (5)	$\otimes$	$\oslash$	$\odot$	$**$
$<$ (1)	$>$ (1)	$\leq$ (1)	$\geq$ (1)	$\sqcap$	$//$
$\prec$	$\succ$	$\preceq$	$\succeq$	$\sqcup$	$\hat{}\hat{}$
$\ll$	$\gg$	$<:$	$:>$ (6)	$\&$	$\&\&$
$\sqsubset$	$\sqsupset$	$\sqsubseteq$ (5)	$\sqsupseteq$	$\mid$	$\%\%$
$\subset$	$\supset$		$\supseteq$	$\star$	$@@$ (6)
$\vdash$	$\dashv$	$\models$	$=\!\mid$	$\bullet$	$\#\#$
$\sim$	$\simeq$	$\approx$	$\cong$	$\$$	$\$\$$
$\bigcirc$	$::=$	$\asymp$	$\doteq$	$??$	$!!$
$\propto$	$\wr$	$\uplus$			

(1) Defined by the *Naturals*, *Integers*, and *Reals* modules.
(2) Defined by the *Reals* module.
(3) Defined by the *Sequences* module.
(4) $x\hat{}y$ is printed as x^y.
(5) Defined by the *Bags* module.
(6) Defined by the *TLC* module.

Table 4: User-definable operator symbols.

$\land$	`/\` or `\land`	$\lor$	`\/` or `\lor`	$\Rightarrow$	`=>`
$\neg$	`~` or `\lnot` or `\neg`	$\equiv$	`<=>` or `\equiv`	$\triangleq$	`==`
$\in$	`\in`	$\notin$	`\notin`	$\neq$	`#` or `/=`
$\langle$	`<<`	$\rangle$	`>>`	$\Box$	`[]`
$<$	`<`	$>$	`>`	$\Diamond$	`<>`
$\leq$	`\leq` or `=<` or `<=`	$\geq$	`\geq` or `>=`	$\leadsto$	`~>`
$\ll$	`\ll`	$\gg$	`\gg`	$\overset{+}{\rhd}$	`-+->`
$\prec$	`\prec`	$\succ$	`\succ`	$\mapsto$	`|->`
$\preceq$	`\preceq`	$\succeq$	`\succeq`	$\div$	`\div`
$\subseteq$	`\subseteq`	$\supseteq$	`\supseteq`	$\cdot$	`\cdot`
$\subset$	`\subset`	$\supset$	`\supset`	$\circ$	`\o` or `\circ`
$\sqsubset$	`\sqsubset`	$\sqsupset$	`\sqsupset`	$\bullet$	`\bullet`
$\sqsubseteq$	`\sqsubseteq`	$\sqsupseteq$	`\sqsupseteq`	$\star$	`\star`
$\vdash$	`|-`	$\dashv$	`-|`	$\bigcirc$	`\bigcirc`
$\models$	`|=`	$=\!\mid$	`=|`	$\sim$	`\sim`
$\rightarrow$	`->`	$\leftarrow$	`<-`	$\simeq$	`\simeq`
$\cap$	`\cap` or `\intersect`	$\cup$	`\cup` or `\union`	$\asymp$	`\asymp`
$\sqcap$	`\sqcap`	$\sqcup$	`\sqcup`	$\approx$	`\approx`
$\oplus$	`(+)` or `\oplus`	$\uplus$	`\uplus`	$\cong$	`\cong`
$\ominus$	`(-)` or `\ominus`	$\times$	`\X` or `\times`	$\doteq$	`\doteq`
$\odot$	`(.)` or `\odot`	$\wr$	`\wr`	x^y	`x^y` (2)
$\otimes$	`(\X)` or `\otimes`	$\propto$	`\propto`	x^+	`x^+` (2)
$\oslash$	`(/)` or `\oslash`	“s”	`"s"` (1)	x^*	`x^*` (2)
$\exists$	`\E`	$\forall$	`\A`	$X^\#$	`x^#` (2)
$\boldsymbol{\exists}$	`\EE`	$\boldsymbol{\forall}$	`\AA`	$'$	`'`
$]_v$	`]_v`	$\rangle_v$	`>>_v`		
WF_v	`WF_v`	SF_v	`SF_v`		
┌────────	`--------` (3)	────────┐	`--------` (3)		
├────────┤	`--------` (3)	└────────┘	`========` (3)		

(1) s is a sequence of characters.

(2) x and y are any expressions.

(3) a sequence of four or more `-` or `=` characters.

Table 5: The ASCII representations of typeset symbols.

Index