# The PlusCal Algorithm Language

Leslie Lamport
Microsoft Research

2 January 2009

**Abstract**

Algorithms are different from programs and should not be described with programming languages. The only simple alternative to programming languages has been pseudo-code. PlusCal is an algorithm language that can be used right now to replace pseudo-code, for both sequential and concurrent algorithms. It is based on the TLA$^+$ specification language, and a PlusCal algorithm is automatically translated to a TLA$^+$ specification that can be checked with the TLC model checker and reasoned about formally.

# Contents

# 1 Introduction

PlusCal is a language for writing algorithms, including concurrent algorithms. While there is no formal distinction between an algorithm and a program, we know that an algorithm like Newton's method for approximating the zeros of a real-valued function is different from a program that implements it. The difference is perhaps best described by paraphrasing the title of Wirth's classic book [19]: a program is an algorithm plus an implementation of its data operations.

The data manipulated by algorithms are mathematical objects like numbers and graphs. Programming languages can represent these mathematical objects only by programming-language objects like bit strings and pointers, introducing implementation details that are irrelevant to the algorithm. The customary way to eliminate these irrelevant details is to use pseudo-code. There are two obvious problems with pseudo-code: it has no precise meaning, and it can be checked only by hand—a notoriously unreliable method of finding errors.

PlusCal is designed to replace pseudo-code for describing algorithms. A PlusCal algorithm is translated to a TLA$^+$ specification [12]. That specification can be debugged (and occasionally even completely verified) with the TLC model checker [20]. A TLA$^+$ specification is a formula of TLA, a logic invented expressly for proving properties of systems, so properties of an algorithm can be proved by reasoning about its translation.

There are other languages that might be satisfactory replacements for pseudo-code in a Utopian world where everyone has studied the language. A researcher can use PlusCal in his next paper; a professor can use it in her next lecture. PlusCal code is simple enough that explaining it is almost as easy as explaining the pseudo-code that it replaces. I know of no other language that can plausibly make this claim and has the expressive power to replace pseudo-code for both sequential and concurrent algorithms. Other languages used to describe algorithms are discussed in the conclusion.

PlusCal's simplicity comes from its simple, familiar programming language constructs that make it resemble a typical toy language. For example, here is the "Hello World" program:

```
--algorithm HelloWorld
  begin  print "Hello, world."
  end algorithm
```

PlusCal has the expressive power to replace pseudo-code because of its rich expression language. A PlusCal expression can be any expression of TLA$^+$,

which means it can be anything expressible in set theory and first-order logic. This gives PlusCal's expression language all the power of ordinary mathematics, making it infinitely more powerful than the expression language of any programming language.

Programming languages have two other deficiencies that make them unsuitable as algorithm languages:

- They describe just one way to compute something. An algorithm might require that a certain operation be executed for all values of $i$ from 1 to $N$; most programming languages must specify in which order those executions are performed. PlusCal provides two simple constructs for expressing nondeterminism.

- 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, and defining a concurrent algorithm requires specifying what constitutes a single (atomic) step. Programming languages provide no well-defined notion of a program step. PlusCal uses labels to describe an algorithm's steps.

  Describing the grain of atomicity is crucial for concurrent algorithms, but is often unimportant for sequential algorithms. Labels can therefore be omitted and the translator instructed to choose the steps, which it makes as large possible to facilitate model checking.

PlusCal combines five important features: simple conventional program constructs, extremely powerful expressions, nondeterminism, a convenient way to describe the grain of atomicity, and model checking. The only novel aspect of any of these features is the particular method of using labels to indicate atomic actions. While the individual features are not new, their combination is. PlusCal is the only language I know of that has them all. This combination of features makes it ideal for writing algorithms.

PlusCal can be used not only in publications and in the classroom, but also in programming. Although most programming involves simple data manipulation, a program sometimes contains a nontrivial algorithm. It is more efficient to debug the algorithm by itself, rather than debugging it and its implementation at the same time. Writing the algorithm in PlusCal and debugging it with TLC before implementing it is a good way to do this.

Being easy to read does not necessarily make PlusCal easy to write. Like any powerful language, PlusCal has rules and restrictions that are not immediately obvious. Because of its inherent simplicity, the basic language should not be hard to learn. What many programmers and computer scientists will

find hard is learning to take advantage of the power of the expression language. TLA$^+$ expressions use only basic math—that is, predicate logic, sets, and functions (which include tuples and records). However, many computer scientists would have difficulty describing even something as simple as a graph in terms of sets and functions. With PlusCal, the writer of an algorithm can reveal to the reader as much or as little of the underlying math as she wishes.

PlusCal's features imply its limitations. Programming languages are complex because of constructs like objects and variable scoping that are useful for writing large programs. PlusCal's simplicity limits the length of the algorithms it can conveniently describe. The largest algorithm I have written in it is about 500 lines. I expect that PlusCal would not work well for algorithms of more than one or two thousand lines. (However, a one-line PlusCal assignment statement can express what in a programming language requires a multi-line loop or the call of a complicated procedure.) Programming languages are inexpressive because they must yield efficient code. While it is possible to restrict PlusCal so it can be compiled into efficient code, any such restriction would reduce its utility for writing algorithms. PlusCal is for writing algorithms, not for writing large specifications or efficient programs.

The semantics of PlusCal is specified formally by its translation to TLA$^+$. A TLA$^+$ specification of the translation is included in the PlusCal distribution, which is available on the Web [8]. (The translator, which is written in Java, has the option of performing the translation by executing this specification with TLC.) The translation is described in Section 4. However, except for its expressions, PlusCal is so simple and most of its constructs so banal that there is no need to give a rigorous semantics here. Instead, the language is explained in Section 2 by a series of examples. Section 3 describes the few features not contained in the examples, and Section 5 completes the language description by explaining the constraints on where labels may and may not appear. To convince the reader that nothing is being hidden, a grammar of the full language (excluding its expressions) appears in the appendix. A language manual is available on the PlusCal Web site.

No attempt is made here to describe the complete language of TLA$^+$ expressions. The TLA$^+$ notation used in the examples is explained only where it does not correspond to standard mathematical usage. The PlusCal language manual briefly explains TLA$^+$ and its expressions. The semantics of TLA$^+$ expressions is trivial in the sense that a semantics consists of a translation to ordinary mathematics, and TLA$^+$ expressions are expressions of ordinary mathematics. A precise explanation of all the TLA$^+$ operators

that can appear in a PlusCal expression is given in Section 16.1 of the TLA$^+$ manual [12].

# 2 Some Examples

A PlusCal algorithm can be written in either of two syntaxes—the clearer but longer p-syntax (*p* for *prolix*), or the more compact c-syntax that will look familiar to most programmers. The first two examples use the p-syntax; the next two use the c-syntax. The grammar given in the appendix is for the c-syntax.

## 2.1 Euclid's Algorithm

The first example is a simple version of Euclid's algorithm from Sedgewick's textbook [18, page 8]. The algorithm computes the GCD of two natural numbers $m$ and $n$ by setting $u$ to $m$ and $v$ to $n$ and executing the following pseudo-code.

**while** $u \neq 0$ **do**
  **if** $u < v$ **then** swap $u$ and $v$ **end if**;
  $u := u - v$
**end while**;

Upon termination, $v$ equals the GCD of $m$ and $n$. The PlusCal version appears in Figure 1 on this page. (Symbols are actually typed as ASCII strings—for example, "$\in$" is typed "`\in`".) The variable declarations assert that the initial values of $m$ and $n$ are in the set $1..K$ of integers from 1 through $K$, and that $u$ and $v$ initially equal $m$ and $n$, respectively. (We will see later where $K$ is declared.) Assignment statements separated by || form a

**--algorithm** *EuclidSedgewick*
**variables** $m \in 1..K,\ n \in 1..K,\ u = m,\ v = n$
**begin while** $u \neq 0$ **do**
    **if** $u < v$ **then** $u := v \ ||\ v := u$ **end if**;
    $u := u - v$
  **end while**;
  **assert** *IsGCD*$(v, m, n)$
**end algorithm**

Figure 1: Euclid's algorithm in PlusCal.

```
MODULE Euclid
EXTENDS Naturals, TLC
CONSTANT K
Divides(i,j)  ≜  ∃ k ∈ 0..j : j = i * k
IsGCD(i,j,k)  ≜    Divides(i,j)
                 ∧ Divides(i,k)
                 ∧ ∀ r ∈ 0..j ∪ 0..k :
                       Divides(r,j) ∧ Divides(r,k) ⇒ Divides(r,i)
(* --algorithm EuclidSedgewick
   ...
   end algorithm *)
\* BEGIN TRANSLATION
    Translator puts TLA+ specification here
\* END TRANSLATION
```

Figure 2: The module containing the PlusCal code for Euclid's algorithm.

multi-assignment, executed by first evaluating all the right-hand expressions and then performing all the assignments. The **assert** statement checks the correctness of the algorithm, where *IsGCD*($v$, $m$, $n$) will be defined to be true iff $v$ is the GCD of $m$ and $n$, for natural numbers $v$, $m$, and $n$.

The algorithm appears in a comment in a TLA$^+$ module, as shown in Figure 2 on this page. The module's EXTENDS statement imports the *Naturals* module, which defines arithmetic operators like subtraction and "..", and a special *TLC* module that is needed because of the algorithm's **assert** statement. The CONSTANT declaration declares the algorithm parameter $K$. The module next defines *Divides*($i$, $j$) to be true for natural numbers $i$ and $j$ iff $i$ divides $j$, and it uses *Divides* to define *IsGCD*.

The translator inserts the algorithm's translation, which is a TLA$^+$ specification, between the BEGIN and END TRANSLATION comment lines, replacing any previous version. The translator also writes a configuration file that controls the TLC model checker. We must add to that file a command that specifies the value of $K$. TLC checks that the assertion is satisfied and that execution terminates for all $K^2$ possible choices of the variables' initial values. For $K = 50$, this takes about 25 seconds. (All execution times are for a 2.4 GHz personal computer.)

**Remarks**

The operation of swapping $u$ and $v$ can of course be expressed without a multiple assignment by declaring an additional variable $t$ and writing:

$$t := u;\ u := v;\ v := t$$

It can also be written as follows.

$$\textbf{with}\ \ t = u\ \ \textbf{do}\ \ u := v;\ v := t\ \ \textbf{end with}$$

The **with** statement declares $t$ to be local to the **do** clause.

Instead of restricting $m$ and $n$ to lie in the range $1\,..\,K$, it would be more natural to allow them to be any positive integers. We do this by replacing $1\,..\,K$ with the set of positive integers; here are three ways to express that set in TLA$^+$, where *Nat* is defined in the *Naturals* module to be the set of all natural numbers:

$$Nat \setminus \{0\} \qquad \{i \in Nat : i > 0\} \qquad \{i + 1 : i \in Nat\}$$

To check the resulting algorithm, we would tell TLC to substitute a finite set of numbers for *Nat*.

As this example shows, PlusCal is untyped. Type correctness is an invariance property of an algorithm asserting that, throughout any execution, the values of the variables belong to certain sets. A type invariant for algorithm *EuclidSedgewick* is that the values of $u$ and $v$ are integers. For a type invariant like this whose proof is trivial, a typed language allows type correctness to be verified by type checking. If the proof is not completely trivial, as for the type invariant that $u$ and $v$ are natural numbers, type correctness cannot be verified by ordinary type checking. (If *natural number* is a type, then type checking is undecidable for a Turing complete language with subtraction.) These type invariants are easily checked by TLC.

## 2.2 The Quicksort Partition Operation

What most distinguishes the version of Euclid's algorithm given above from a program in an ordinary language is the expression $IsGCD(v, m, n)$. It hints at the expressive power that PlusCal obtains by using TLA$^+$ as its expression language. I now present a more compelling example of this: the *partition* operation of the quicksort algorithm [4].

Consider a version of quicksort that sorts an array $A[1], \ldots, A[N]$ of numbers. It uses the operation $Partition(lo, hi)$ that chooses a value *pivot* in $lo\,..\,(hi-1)$ and permutes the array elements $A[lo], \ldots, A[hi]$ to make

$A[i] \leq A[j]$ for all $i$ in $lo\,..\,pivot$ and $j$ in $(pivot+1)\,..\,hi$. It is easy to describe a particular implementation of this operation with a programming language. The following PlusCal statement describes what the operation $Partition(lo, hi)$ is supposed to do, not how it is implemented. The code assumes that $Perms(A)$ is defined to be the set of permutations of $A$.

```
with piv ∈ lo..(hi−1),
     B ∈ { C ∈ Perms(A) :
                (∀ i ∈ 1..(lo − 1) ∪ (hi + 1)..N : C[i] = A[i] )
              ∧ (∀ i ∈ lo..piv, j ∈ (piv + 1)..hi : C[i] ≤ C[j] ) }
  do pivot := piv ;
     A := B
end with
```

This **with** statement is executed by nondeterministically choosing values of $piv$ and $B$ from the indicated sets and then executing the **do** clause. TLC will check the algorithm with all possible executions of this statement.

The operator $Perms$ is defined in TLA$^+$ as follows, using local definitions of $Auto(S)$ to be the set of automorphisms of $S$, if $S$ is a finite set, and of $\star$ to be function composition. (Arrays are what mathematicians call functions. In TLA$^+$, $[A \to B]$ is the set of functions with domain $A$ and range a subset of $B$, and DOMAIN $F$ is the domain of $F$ if $F$ is a function.)

$$
\begin{array}{l}
Perms(B) \triangleq \\
\quad \text{LET } Auto(S) \triangleq \{f \in [S \to S] : \forall\, y \in S : \exists\, x \in S : f[x] = y\} \\
\qquad\quad f \star g \triangleq [x \in \text{DOMAIN } g \mapsto f[g[x]]] \\
\quad \text{IN } \{B \star f : f \in Auto(\text{DOMAIN } B)\}
\end{array}
$$

Using the description above of the *partition* operation and this definition of *Perms*, TLC will check partial correctness and termination of the usual recursive version of quicksort for all 4-element arrays $A$ with values in a set of 4 numbers in about 100 seconds.

### Remarks

This example is not typical. It was chosen to illustrate two things: how nondeterminism can be conveniently expressed by means of the **with** statement, and the enormous expressive power that PlusCal achieves by its use of ordinary mathematical expressions. The definition of *Perms* is the TLA$^+$ statement of one that many mathematicians would write, but few computer

```
 ncs: noncritical section;
start: ⟨b[i] := true⟩;
       ⟨x := i⟩;
       if ⟨y ≠ 0⟩ then ⟨b[i] := false⟩;
                       await ⟨y = 0⟩;
                       goto start   fi;
       ⟨y := i⟩;
       if ⟨x ≠ i⟩ then ⟨b[i] := false⟩;
                       for j := 1 to N do await ⟨¬b[j]⟩ od;
                       if ⟨y ≠ i⟩ then await ⟨y = 0⟩;
                                       goto start   fi   fi;
       critical section;
       ⟨y := 0⟩;
       ⟨b[i] := false⟩;
       goto ncs
```

Figure 3: Process $i$ of the Fast Mutual Exclusion Algorithm, based on the original description. It assumes that initially $x = y = 0$ and $b[i] = \mathit{false}$ for all $i$ in $1\,..\,N$.

scientists would. Almost all computer scientists would define $\mathit{Perms}(B)$ by recursion on the number of elements in $B$, the way it would be computed in most programming languages. (Such a definition can also be written in TLA$^+$.) To appreciate the power of ordinary mathematics, the reader should try to write a recursive definition of $\mathit{Perms}$.

A standard computer science education does not provide the familiarity with simple math needed to make the definition of $\mathit{Perms}$ easy to understand. A textbook writer therefore might not want to include it in a description of quicksort. Because the definition is external to the PlusCal code, the writer has the option of omitting it and informally explaining the meaning of $\mathit{Perms}(B)$. On the other hand, a professor might want to take advantage of the opportunity it provides for teaching students some math.

### 2.3 The Fast Mutual Exclusion Algorithm

An example of a multiprocess algorithm is provided by the Fast Mutual Exclusion Algorithm [10]. The algorithm has $N$ processes, numbered from 1 through $N$. Figure 3 on this page is the original description of process number $i$, except with the noncritical section and the outer infinite loop made explicit. Angle brackets enclose atomic operations (steps). For example, the

```
--algorithm FastMutex
 { variables x = 0, y = 0, b = [i ∈ 1..N ↦ FALSE] ;
   process (Proc ∈ 1..N)
    variable j ;
     { ncs: 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 ;
              goto ncs }}
```

Figure 4: The Fast Mutual Exclusion Algorithm in PlusCal.

evaluation of the expression $y \neq 0$ in the first **if** statement is performed as a single step. If that expression equals *true*, then 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.)

A PlusCal version of the algorithm appears in Figure 4 on this page. The preceding examples use PlusCal's p-syntax; this example is written in PlusCal's alternative c-syntax. The PlusCal version differs from the original pseudo-code in the following nontrivial ways.

- It explicitly declares the global variables $x$, $y$, and $b$ and their initial values, as well as the process-local variable $j$, whose initial value is not specified. (The TLA$^+$ expression $[v \in S \mapsto e]$ is the function $F$ with domain $S$ such that $F[v] = e$ for all $v$ in $S$.)

- It declares a set of processes with identifiers in the set $1..N$ (one

process for each identifier). Within the body of the **process** statement, *self* denotes the identifier of the process.

- The critical and noncritical sections are represented by atomic **skip** instructions. (Because TLA specifications are closed under stuttering steps [9, 12], this algorithm actually describes nonatomic critical and noncritical sections that can do anything except modify the variables $x$, $y$, $b$, and $j$ or jump to a different part of the process.)

- The grain of atomicity is expressed by labels. A single atomic step consists of an execution starting at a label and ending at the next label. 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 $l4$.

- A **while** loop implements the original's **for** statement.

As this example shows, a PlusCal **await** statement can occur within a larger atomic action. A step containing the statement "**await** $P$" can be executed only when $P$ evaluates to TRUE. This statement is equivalent to the dynamic logic statement "$P$**?**" [16].

For this algorithm, mutual exclusion means that no two processes are simultaneously at control point $cs$. The translation introduces a variable $pc$ to represent the control state, where control in process $p$ is at $cs$ iff $cs[p]$ equals "cs". Mutual exclusion is therefore asserted by the invariance of:

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

TLC can check mutual exclusion and the absence of deadlock for all executions in about 15 seconds for $N = 3$ and 15 minutes for $N = 4$. It takes TLC about 5 times as long to check the absence of livelock as well, assuming weak fairness of each process's actions. (Fairness is discussed in Section 4.3.)

### Remarks

Observe how similar the PlusCal version is to the pseudo-code, presented almost exactly as previously published. The 15 lines of pseudo-code are expressed in PlusCal with 17 lines of statements plus 4 lines of declarations. Those declarations include specifications of the initial values of variables, which are not present in the pseudo-code and are expressed by accompanying text. The extra two lines of PlusCal statements arise from converting a **for** to a **while**. (For simplicity, TLA$^+$ has no **for** or **until** statement.)

Readers who had never seen PlusCal would need the following explanation of the code in Figure 4.

> The **process** declaration asserts that there are $N$ processes, numbered from 1 through $N$, and gives the code for process *self*. Execution from one label to the next is an atomic action, and an **await** $P$ statement can be executed only when $P$ is true. Variable declarations specify the initial value of variables, $b$ being initially equal to an array with $b[i] = \text{FALSE}$ for each process $i$.

Compare this with the following explanation that would be needed by readers of the pseudo-code in Figure 3.

> The algorithm has $N$ processes, numbered from 1 through $N$; the code of process $i$ is given. Angle brackets enclose atomic operations, and an **await** $P$ statement can be executed only when $P$ is true. Variables $x$ and $y$ are initially equal to 0, and $b[i]$ is initially equal to *false* for each process $i$.

Instead of asserting mutual exclusion by a separate invariant, we can replace the critical section's **skip** statement by the following assertion that no other process is in its critical section.

> **assert** $\forall\, p \in 1\,..\,N \setminus \{self\} \;:\; pc[p] \neq$ "cs"

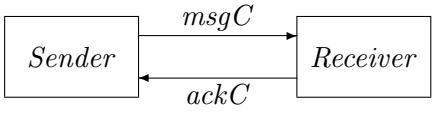
Correctness of the algorithm does not depend on the order in which a process examines other processes' variables. The published version of the algorithm used a **for** loop to examine them in one particular order because there was no simple standard construct for examining them in an arbitrarily chosen order. To allow the iterations of the loop body to be performed in any order, we just replace the corresponding PlusCal code of Figure 4 with the following.

```
    j := 1..N;
l8: while (j ≠ {}) { with (e ∈ j) { await ¬b[e];
                                    j := j \ {e}  } };
```

Weak fairness of each process's actions prevents a process from remaining forever in its noncritical section—something that a mutual exclusion algorithm must allow. Absence of livelock should be checked under the assumption of weak fairness for each process's actions other than the noncritical section action. Section 4.3 explains how such a fairness assumption is asserted.

## 2.4 The Alternating Bit Protocol

Our final example is the alternating bit protocol, which is a distributed message-passing algorithm [14, Section 22.3]. A sender and a receiver process communicate over lossy FIFO channels, as pictured here.




To send a message $m$, the sender repeatedly sends the pair $\langle m, sbit \rangle$ on channel $msgC$, where $sbit$ equals 0 or 1. The sender acknowledges receipt of the message by repeatedly sending $sbit$ on channel $ackC$. Upon receipt of the acknowledgement, the sender complements $sbit$ and begins sending the next message.

The PlusCal version of the algorithm appears in Figure 5 on the next page. To understand it, you must know how finite sequences are represented in TLA$^+$'s standard *Sequences* module. A sequence $\sigma$ of length $N$ is a function (array) whose domain (index set) is $1 \,..\, N$, where $\sigma[i]$ is the $i^{\text{th}}$ element of the sequence. The *Head* and *Tail* operators are defined as usual, $Len(\sigma)$ is the length of sequence $\sigma$, and $Append(\sigma, e)$ is the sequence obtained by appending the element $e$ to the tail of $\sigma$. Tuples are just finite sequences, so the pair $\langle a, b \rangle$ is a two-element sequence and $\langle a, b \rangle[2]$ equals $b$.

The algorithm assumes that the set *Msg* of possible messages is defined or declared and that $Remove(i, \sigma)$ is the sequence obtained by removing the $i^{\text{th}}$ element of $\sigma$ if $1 \leq i \leq Len(\sigma)$. It can be defined in the TLA$^+$ module by

$$Remove(i, seq) \triangleq [\, j \in 1 \,..\, (Len(seq) - 1) \mapsto \text{IF } j < i \text{ THEN } seq[j] \text{ ELSE } seq[j+1] \,]$$

The channels $msgC$ and $ackC$ are represented by variables whose values are finite sequences, initially equal to the empty sequence $\langle\,\rangle$. The variable *input* is the finite sequence of messages that the sender has decided to send and the variable *output* is the sequence of messages received by the receiver; initially both equal the empty sequence.

The operations of sending and receiving a message on a channel are represented by the macros *Send* and *Rcv*. Macros are expanded syntactically. For example, the statement $Send(rbit, ackC)$ is replaced by

$$ackC := Append(ackC, rbit)$$

```
--algorithm ABProtocol
{ variables input = ⟨⟩; output = ⟨⟩; msgC = ⟨⟩; ackC = ⟨⟩;
  macro Send(m, chan) { chan := Append(chan, m) }
  macro Rcv(v, chan) { await chan ≠ ⟨⟩;
                       v := Head(chan);
                       chan := Tail(chan) }
  process (Sender = "S")
   variables next = 1; sbit = 0; ack;
   { s: while (TRUE) {
          either with (m ∈ Msg) { input := Append(input, m) }
          or       { await next ≤ Len(input);
                     Send(⟨input[next], sbit⟩, msgC) }
          or       { Rcv(ack, ackC);
                     if (ack = sbit) { next := next + 1;
                                       sbit := (sbit + 1) % 2 }}}}
  process (Receiver = "R")
   variables rbit = 1; msg;
   { r: while (TRUE) {
          either Send(rbit, ackC)
          or       { Rcv(msg, msgC);
                     if (msg[2] ≠ rbit) { rbit := (rbit + 1) % 2
                                          output := Append(output, msg[1])}}}}
  process (LoseMsg = "L")
   { l: while (TRUE) {
          either with (i ∈ 1..Len(msgC)) { msgC := Remove(i, msgC)}
          or       with (i ∈ 1..Len(ackC)) { ackC := Remove(i, ackC) }}}
}
```

Figure 5: The Alternating Bit Protocol in PlusCal.

which appends *rbit* to the sequence *ackC*. If $v$ and *chan* are variables and *chan* equals a finite sequence, then the operation $Rcv(v, chan)$ can be executed iff *chan* is non-empty, in which case it sets $v$ to the first element of *chan* and removes that element from *chan*.

There are three processes: the sender, the receiver, and a *LoseMsg* process that models the lossiness of the channels by nondeterministically deleting messages from them. The process declaration *Sender* = "S" indicates that there is a single *Sender* process with identifier the string "S"; it is equivalent to the declaration *Sender* $\in$ {"S"}. The only new PlusCal construct in the processes' code is

**either** $S_1$ **or** $S_2$ ... **or** $S_n$

which executes $S_i$ for a nondeterministically chosen $i$.

The three processes run forever. The presence of just one label in each process means that the execution of one iteration of its **while** statement's body is a single atomic action. The sender can either choose a new message to send and append it to *input*, send the current message $input[next]$, or receive an acknowledgement (if *ackC* is non-empty). The receiver can either receive a message and, if the message has not already been received, append it to *output*; or it can send an acknowledgement. A single step of the *LoseMsg* process removes an arbitrarily chosen message from either *msgC* or *ackC*. If *msgC* is the empty sequence, then $1\,..\,Len(msgC)$ is the empty set and only the **or** clause of the *LoseMsg* process can be executed. If both *msgC* and *ackC* equal the empty sequence, then the *LoseMsg* process is not enabled and can perform no step. (See Section 4.2 below for an explanation of why this is the meaning of the process's code.)

The important safety property satisfied by the algorithm is that the receiver never receives an incorrect message. This means that the sequence *output* of received messages is an initial subsequence of the sequence *input* of messages chosen to be sent. This condition is asserted by the predicate $output \sqsubseteq input$, where $\sqsubseteq$ is defined by:

$$s \sqsubseteq t \;\triangleq\; (Len(s) \leq Len(t)) \land (\forall\, i \in 1..Len(s) \,:\, s[i] = t[i])$$

Section 4.3 discusses the desired liveness property, that every chosen message is eventually received.

Algorithm *ABProtocol* has an infinite number of reachable states. The sequence *input* can become arbitrarily long and, even if the sender puts only a single message in *input*, the sequences *msgC* and *argC* can become arbitrarily long. TLC will run forever on an algorithm with an infinite set

of reachable states unless it finds an error. (TLC will eventually exceed the capacity of some data structure and halt with an error, but that could take many years because it keeps on disk the information about what states it has found.) We can bound the computation by telling TLC to stop any execution of the algorithm when it reaches a state not satisfying a specified constraint. For example, the constraint

$$(Len(input) < 4) \land (Len(msgC) < 5) \land (Len(ackC) < 5)$$

stops an execution when *input* has 4 messages or one of the channels has 5 messages. With this constraint and a set *Msg* containing 3 elements, TLC model checks the algorithm in 7.5 seconds.

**Remarks**

It may appear that, by introducing the *LoseMsg* process, we are forcing the channels to lose messages. This is not the case. As discussed in Section 4.3 below, an algorithm's code describes only what steps *may* be executed; it says nothing about what steps *must* be executed. Algorithm *ABProtocol*'s code does not require the *LoseMsg* process ever to delete a message, or the *Sender* process ever to send one. Section 4.3 explains how to specify what the algorithm must do.

Each process of the algorithm consists of an infinite loop whose body nondeterministically chooses one atomic action to execute. This structure is typical of high-level versions of distributed algorithms.

This example shows that PlusCal can easily describe a distributed message-passing algorithm, even though it has no special constructs for sending and receiving messages. Adding such constructs could eliminate the four lines of macros. However, what operations should they specify? Are messages broadcast or sent on point-to-point channels? Are they always delivered in order? Can they be lost? Can the same message be received twice? Different distributed algorithms make different assumptions about message passing, and I know of no simple construct that covers all possibilities. Any particular kind of message passing that is easy to explain should be easy to describe in PlusCal.

# 3 The Complete Language

We have seen almost all the PlusCal language constructs. The major omissions are the following (written in the p-syntax).

- TLA$^+$ has notation for records, where a record is a function whose domain is a finite set of strings and $a.b$ is syntactic sugar for $a[\text{“b”}]$. PlusCal allows the usual assignment to fields of a record, as in

  $$v.a := 0;\ A[0].b := 42;$$

  TLC will report an error if it tries to execute this code when $v$ is not a record with an $a$ component or $A$ is not an array with $A[0]$ a record having a $b$ component. This usually implies that $v$ and $A$ must be initialized to values of the correct “type”.

- The **if** statement has optional **elsif** clauses (only in the p-syntax) followed by an optional **else** clause.

- PlusCal has procedure declarations and **call** and **return** statements. Since **call** is a statement, it does not return a value. The customary approach of making procedure calls part of expression evaluation would make specifying steps problematic, and allowing return values would complicate the translation. Procedures can easily return values by setting global variables (or process-local variables for multiprocess algorithms).

- PlusCal has an optional **define** statement for inserting TLA$^+$ definitions. It goes immediately after the declarations of the algorithm's global variables and permits operators defined in terms of those variables to be used in the algorithm's expressions.

The description of the language is completed in Section 5, which explains where labels are forbidden or required.

# 4 The TLA$^+$ Translation

## 4.1 An Example

A TLA$^+$ specification describes a set of possible behaviors, where a behavior is a sequence of states and a state is an assignment of values to variables. The heart of a TLA$^+$ specification consists of an initial predicate and a next-state action. The initial predicate specifies the possible initial states, and the next-state action specifies the possible state transitions. An action is a formula containing primed and unprimed variables, where unprimed variables refer to the old state and primed variables refer to the new state. For example, the action $x' = x + y'$ specifies all transitions in which the

```
--algorithm EuclidSedgewick
variables m ∈ 1..K, n ∈ 1..K, u = m, v = n
begin L1: while u ≠ 0 do
              if u < v then u := v || v := u end if;
          L2: u := u − v
            end while;
            assert IsGCD(v, m, n)
    Done:
end algorithm
```

Figure 6: Euclid's algorithm, showing labels $L1$ and $L2$ implicitly added by the translator and the implicit label *Done*.

value of $x$ in the new state equals the sum of its value in the old state and the value of $y$ in the new state.

The translation from PlusCal to TLA$^+$ is illustrated with the version of Euclid's algorithm from Section 2.1. The algorithm is shown in Figure 6 on this page with the two labels, $L1$ and $L2$, implicitly added by the translator. Also shown is the implicit label *Done* that represents the control point at the end of the algorithm.

The translation appears in Figure 7 on the next page. It uses the TLA$^+$ notation that a list of formulas bulleted with $\land$ or $\lor$ symbols denotes their conjunction or disjunction. Indentation is significant and is used to eliminate parentheses. (This notation makes large formulas easier to read, and engineers generally like it; but it confuses many computer scientists. The notation can be used in PlusCal expressions.)

The important parts of the translation are the definitions of the initial predicate *Init* and the next-state action *Next*. The predicate *Init* is obtained in the obvious way from the variable declaration, with the variable *pc* that represents the control state initialized to the initial control point—that is, to the string "L1".

Actions $L1$ and $L2$ specify the transitions representing execution steps starting at the corresponding control points. The conjunct $pc$ = "L1" of action $L1$ asserts that a transition can occur only in a starting state in which the value of the variable *pc* is "L1". (A conjunct containing no primed variables is an enabling condition.) The expression UNCHANGED $f$ is an abbreviation for $f' = f$, so the conjunct UNCHANGED $\langle u, v \rangle$ asserts that the values of $u$ and $v$ are left unchanged by the transition. The imported *TLC* module defines $Assert(A, B)$ to equal $A$, but TLC halts and prints the value $B$ and a trace of the current execution if it evaluates the expression when

$$
\begin{array}{l}
\mathit{Init} \triangleq \land\ m \in 0\,..\,K \\
\qquad\qquad \land\ n \in 1\,..\,K \\
\qquad\qquad \land\ u = m \\
\qquad\qquad \land\ v = n \\
\qquad\qquad \land\ pc = \text{“L1”}
\end{array}
$$

$$
\begin{array}{l}
L1 \triangleq \land\ pc = \text{“L1”} \\
\qquad\qquad \land\ \text{IF}\ \ u \neq 0\ \ \text{THEN}\ \ \land\ \text{IF}\ \ u < v\ \ \text{THEN}\ \ \land\ u' = v \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ v' = u \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{ELSE}\ \ \text{UNCHANGED}\ \langle u, v \rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ pc' = \text{“L2”} \\
\qquad\qquad\qquad\qquad\qquad\quad \text{ELSE}\ \ \land\ \mathit{Assert}(\mathit{IsGCD}(v, m, n), \text{“Failure of assertion at. . .”}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ pc' = \text{“Done”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ \text{UNCHANGED}\ \langle u, v \rangle \\
\qquad\qquad \land\ \text{UNCHANGED}\ \langle m, n \rangle
\end{array}
$$

$$
\begin{array}{l}
L2 \triangleq \land\ pc = \text{“L2”} \\
\qquad\qquad \land\ u' = u - v \\
\qquad\qquad \land\ pc' = \text{“L1”} \\
\qquad\qquad \land\ \text{UNCHANGED}\ \langle m, n, v \rangle
\end{array}
$$

$$\mathit{vars} \triangleq \langle m, n, u, v, pc \rangle$$

$$\mathit{Next} \triangleq L1 \lor L2 \lor (pc = \text{“Done”} \land \text{UNCHANGED}\ \mathit{vars})$$

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

Figure 7: The translation of Euclid's algorithm.

$A$ equals FALSE.

The next-state action *Next* allows all transitions that are allowed by $L1$ or $L2$, or that leave the tuple *vars* of all the algorithm variables unchanged (are stuttering steps [9, 12]) when a terminated state has been reached. This last disjunct keeps TLC from reporting deadlock when the algorithm terminates. (An algorithm deadlocks when no further step is possible; termination is just deadlock we want to occur.) Since every TLA specification allows stuttering steps, this disjunct does not change the meaning of the specification, just the way TLC checks it.

Finally, *Spec* is defined to be the TLA formula that describes the safety part of the algorithm's complete specification. Proving that the algorithm satisfies a safety property expressed by a temporal formula $P$ means proving $\mathit{Spec} \Rightarrow P$. Most PlusCal users can ignore *Spec*.

### 4.2 Translation as Semantics

A classic way of stating that a programming language is poorly defined is to say that its semantics is specified by the compiler. A goal of PlusCal was to make an algorithm's translation so easy to understand that it is a useful specification of the algorithm's meaning. To achieve this goal, the following principles were maintained:

T1. The only TLA$^+$ variables used in the translation are the ones declared in the algorithm plus *pc*. (Algorithms with procedures also use a variable *stack* for saving return locations and values of local procedure variables.)

T2. All identifiers declared or defined in the translation (including bound variables) are taken from the algorithm text, except for a few standard ones like *Init* and *Next*. ("Algorithm text" includes labels implicitly added by the translator.)

T3. There is a one-to-one correspondence between expressions in the translation and expressions in the algorithm. (The only exceptions are the expressions for pushing and popping items on the stack in the translation of procedure **call** and **return** statements.)

It may seem that PlusCal is so simple that its semantics is obvious. However, a naive user might be puzzled by what the following statement in a multiprocess algorithm does when $x$ equals 0:

$L1$: $x := x - 1$; **await** $x \geq 0$; $y := x$;
$L2$: ...

Is $x$ decremented but $y$ left unchanged? Is the execution aborted and the original value of $x$ restored? The statement's translation is:

$$\begin{aligned} L1 \triangleq\ & \land\ pc = \text{“}\mathsf{L1}\text{”} \\ & \land\ x' = x - 1 \\ & \land\ x' \geq 0 \\ & \land\ y' = x' \\ & \land\ \text{UNCHANGED}\ \ldots \end{aligned}$$

Action $L1$ equals FALSE when $x = 0$, which is satisfied by no step, so the statement cannot be executed while $x$ is less than 1. Statement $L1$ is equivalent to

**await** $x > 0$; $x := x - 1$; $y := x$;

because the two statements' translations are mathematically equivalent. Realizing this might help users think in terms of what a computation does rather than how it does it.

Even a fairly sophisticated user may have trouble understanding this statement:

$L1$: **with** $i \in \{1, 2\}$ **do** **await** $i = 2$
**end with**;
$L2$: ...

Is it possible for an execution to deadlock because the **with** statement selects $i = 1$ and the **await** statement then waits forever for $i$ to equal 2? The answer is probably not obvious to readers unfamiliar with dynamic logic. The translation of statement $L1$ is:

$$\begin{aligned} L1 \triangleq\ & \land\ pc = \text{“L1”} \\ & \land\ \exists\, i \in \{1, 2\} : i = 2 \\ & \land\ pc' = \text{“L2”} \\ & \land\ \text{UNCHANGED}\ \langle \ldots \rangle \end{aligned}$$

It should be clear to anyone who understands simple predicate logic that the second conjunct equals TRUE, so statement $L1$ is equivalent to **skip**.

These two examples are contrived. The first will not occur in practice because no one will put an **await** statement after an assignment within a single step, but the second abstracts a situation that occurs in real examples. Consider the *LoseMsg* process in the alternating bit protocol of Figure 5. It may not be clear what the **either**/**or** statement means if one or both channels are empty. Examining the TLA$^+$ translation reveals that the disjunct of the next-state action that describes steps of this process is:

$$\begin{aligned} & \land\ pc[\text{“L”}] = \text{“l”} \\ & \land\ \lor\ \land\ \exists\, i \in 1\,..\,Len(msgC) : msgC' = Remove(i, msgC) \\ & \qquad\ \land\ \text{UNCHANGED}\ ackC \\ & \quad\ \lor\ \land\ \exists\, i \in 1\,..\,Len(ackC) : ackC' = Remove(i, ackC) \\ & \qquad\ \land\ \text{UNCHANGED}\ msgC \\ & \land\ pc' = [pc\ \text{EXCEPT}\ ![\text{“L”}] = \text{“l”}] \\ & \land\ \text{UNCHANGED}\ \langle input, output, next, sbit, ack, rbit, msg \rangle \end{aligned}$$

(The reader should be able to deduce the meaning of the EXCEPT construct and, being smarter than the translator, should realize that the action's first conjunct implies that its third conjunct is a complicated way of asserting $pc' = pc$.) If $msgC$ is the empty sequence, then $Len(msgC) = 0$, so

$1 \,..\, Len(msgC)$ equals the empty set. Since $\exists\, i \in \{\} : \ldots$ equals FALSE, this action's second conjunct is equal to the conjunct's second disjunct. Hence, when $msgC$ equals the empty sequence, a step of the *LoseMsg* process can only be one that removes a message from $ackC$. If $ackC$ also equals the empty sequence, then the entire action equals FALSE, so in this case the process can do nothing.

It is not uncommon to specify the semantics of a programming language by a translation to another language. However, the TLA$^+$ translation can explain to ordinary users the meanings of their programs. The translation is written in the same module as the algorithm. The use of labels to name actions makes it easy to see the correspondence between the algorithm's code and disjuncts of the next-state action. (The translator can be directed to report the names and locations in the code of all labels that it adds.)

The semantics of PlusCal is defined formally by a TLA$^+$ specification of the translator as a mapping from an algorithm's abstract syntax tree to the sequence of tokens that form its TLA$^+$ specification [8]. The part of the specification that actually describes the translation is about 700 lines long (excluding comments). This specification is itself executable by TLC. The translator has a mode in which it parses the algorithm, writes a module containing the TLA$^+$ representation of the abstract syntax tree, calls TLC to execute the translation's specification for that syntax tree, and uses TLC's output to produce the algorithm's TLA$^+$ translation. (The abstract syntax tree does not preserve the formatting of expressions, so this translation may be incorrect for algorithms with expressions that use the TLA$^+$ bulleted conjunction/disjunction list notation.)

### 4.3 Liveness

An algorithm's code specifies the steps that may be taken; it does not require any steps to be taken. In other words, the code specifies the safety properties of the algorithm. To deduce liveness properties, which assert that something does eventually happen, we have to add liveness assumptions to assert when steps must be taken. These assumptions are usually specified as fairness assumptions about actions [3]. The two common types of fairness assumption are weak and strong fairness of an action. Weak fairness of action $A$ asserts that an $A$ step must occur if $A$ remains continuously enabled. Strong fairness asserts that an $A$ step must occur if $A$ keeps being enabled, even if it is also repeatedly disabled.

For almost all sequential (uniprocess) algorithms, the only liveness requirement is termination. It must be satisfied under the assumption that

the algorithm keeps taking steps as long as it can, which means under the assumption of weak fairness of the entire next-state action. (Since there is no other process to disable an action, weak fairness is equivalent to strong fairness for sequential algorithms.) The PlusCal translator can be directed to create the appropriate TLA$^+$ translation and TLC configuration file to check for termination.

For multiprocess algorithms, there is an endless variety of liveness requirements. Any requirement other than termination must be defined by the user in the TLA$^+$ module as a temporal-logic formula, and the TLC configuration file must be modified to direct TLC to check that it is satisfied. The three most common fairness assumptions are weak and strong fairness of each process's next-state action and weak fairness of the entire next-state action—the latter meaning that the algorithm does not halt if any process can take a step, but individual processes may be starved. The PlusCal translator can be directed to add one of these three fairness assumptions to the algorithm's TLA$^+$ translation. However, there is a wide variety of other fairness assumptions made by algorithms. These must be written by the user as temporal-logic formulas.

As an example, let us return to algorithm *ABProtocol* of Section 2.4. A liveness property we might want to require is that every message that is chosen is eventually delivered. Since the safety property implies that incorrect messages are not delivered, it suffices to check that enough message are delivered. This is expressed by the following temporal logic formula, which asserts that for any $i$, if *input* ever contains $i$ elements then *output* will eventually contain $i$ elements:

$$\forall\, i \in Nat \,:\, (Len(input) = i) \rightsquigarrow (Len(output) = i)$$

The algorithm satisfies this property under the assumption of strong fairness of the following operations:

- The sender's first **or** clause, which can send a message
- The sender's second **or** clause, which can receive an acknowledgement.
- The receiver's **either** clause, which can send an acknowledgement.
- The receiver's **or** clause, which can receive a message.

The translation defines the formula *Sender* to be the sender's next-state action. It is the disjunction of three formulas that describe the three clauses of the **either**/**or** statement. The first **or** clause is the only one that can

modify $msgC$, so the action describing that clause is $Sender \land (msgC' \neq msgC)$. Similarly, the sender's last **or** clause is described by the action $Sender \land (ackC' \neq ackC)$. The relevant receiver actions are defined similarly. The complete TLA$^+$ specification of the algorithm, with these four strong fairness conditions, is the following formula:

$$
\begin{aligned}
&\land\ Spec \\
&\land\ \mathrm{SF}_{vars}(Sender \land (ackC' \neq ackC)) \\
&\land\ \mathrm{SF}_{vars}(Sender \land (msgC' \neq msgC)) \\
&\land\ \mathrm{SF}_{vars}(Receiver \land (ackC' \neq ackC)) \\
&\land\ \mathrm{SF}_{vars}(Receiver \land (msgC' \neq msgC))
\end{aligned}
$$

This specification makes no fairness assumption on the sender's operation of choosing a message to send or on the *LoseMsg* process's operation of deleting a message. Those operations need never be executed.

To check the liveness property $\forall\, i \in Nat \ldots$, we must tell TLC to substitute a finite set for $Nat$. With the constraint described in Section 2.4, it suffices to substitute $0 \,..\, 4$ for $Nat$. It then takes TLC about 3.5 minutes to check that the algorithm satisfies the liveness property, about 30 times as long as the 7.5 seconds taken to check safety. This ratio of 30 is unusually large for such a small example; it arises because the liveness property being checked is essentially the conjunction of five formulas that are checked separately—one for each value of $i$. For a single value of $i$, the ratio of liveness to safety checking is about the same factor of 5 as for the Fast Mutual Exclusion Algorithm.

Fairness is subtle. Many readers may not understand why these four fairness assumptions are sufficient to ensure that all messages are received, or why strong fairness of the complete next-state actions of the sender and receiver are not. The ability to mechanically check liveness properties is quite useful. Unfortunately, checking liveness is inherently slower than checking safety and cannot be done on as large an instance of an algorithm. Fortunately, liveness errors tend to be less subtle than safety errors and can usually be caught on rather small instances.

# 5 Labeling Constraints

PlusCal puts a number of restrictions on where labels can and must appear. They are added to keep the TLA$^+$ translation simple—in particular, to achieve the principles T1–T3 described in Section 4.2. Here are the re-

strictions. (They can be stated more succinctly, but I have split apart some rules when different cases have different rationales.)

*A* **while** *statement must be labeled.*

Programming languages need loops to describe simple computations; PlusCal does not. For example, it is easy to write a single PlusCal assignment statement that sets $x[i]$ to the $i^{\text{th}}$ prime, for all $i$ in the domain of $x$. In PlusCal, a loop is a sequence of repeated steps. Eliminating this restriction would require an impossibly complicated translation.

*In any control path, there must be a label between two assignments to the same variable. However, a single multi-assignment statement may assign values to multiple components of the same (array- or record-valued) variable.*

This is at worst a minor nuisance. Multiple assignments to a variable within a step can be eliminated by using a **with** statement—for example, replacing

$$x := f(x);\ \ldots ;\ x := g(x, y)$$

by

$$\textbf{with}\ \ temp = f(x)\ \ \textbf{do}\ \ldots ;\ x := g(temp, y)\ \ \textbf{end with}$$

A translation could perform such a rewriting, but that would require violating T2.

*A statement must be labeled if it is immediately preceded by an* **if** *or* **either** *statement that contains a* **goto***,* **call***,* **return***, or labeled statement within it.*

Without this restriction, the translation would have to either duplicate expressions, violating T3, or else avoid such duplication by giving expressions names, violating T2.

*The first statement of a process or of a uniprocess algorithm must be labeled.*

This is a natural requirement, since a step is an execution from one label to the next.

*The* **do** *clause of a* **with** *statement cannot contain any labeled statements.*

Allowing labels within a **with** statement would require the **with** variables to become TLA$^+$ variables, violating T1.

*A statement other than a* **return** *must be labeled if it is immediately preceded by a* **call***; and a procedure's first statement must be labeled.*

This means that executing a procedure body requires at least one complete step. There is no need for intra-step procedure executions in PlusCal; anything they could compute can be described by operators defined in the TLA$^+$ module.

*A statement that follows a* **goto** *or* **return** *must be labeled.*

This just rules out unreachable statements.

*A macro body cannot contain any labeled statements.*

A macro can be used multiple times within a single process, where it makes no sense for the same label to appear more than once. Related to this constraint is the restriction that a macro body cannot contain a **while**, **call**, **return**, or **goto** statement.

# 6 Conclusion

PlusCal is a language for writing algorithms. It is designed not to replace programming languages, but to replace pseudo-code. Why replace pseudo-code? No formal language can be as powerful or easy to write. Nothing can beat the convenience of inventing new constructs as needed and letting the reader try to deduce their meaning from informal explanations.

The major problem with pseudo-code is that it cannot be tested, and untested code is usually incorrect. In August of 2004, I did a Google search for *quick sort* and tested the first ten actual algorithms on the pages it found. Of those ten, four were written in pseudo-code; they were all incorrect. The only correct versions were written in executable code; they were undoubtedly correct only because they had been debugged.

Algorithms written in PlusCal can be tested with TLC—either by complete model checking or by repeated execution, making nondeterministic choices randomly. It takes effort to write an incorrect sorting algorithm that correctly sorts all arrays of length at most 4 with elements in $1\,..\,4$. An example of an incorrect published concurrent algorithm and how its error could have been found by using PlusCal appears elsewhere [13].

Another advantage of an algorithm written in PlusCal is that it has a precise meaning that is specified by its TLA$^+$ translation. The translation can be a practical aid to understanding the meaning of the code. Since the translation is a formula of TLA, a logic with well-defined semantics and

proof rules [11], it can be used to reason about the algorithm with any desired degree of rigor.

We can use anything when writing pseudo-code, including PlusCal. Pseudo-code is therefore, in principle, more expressive than PlusCal. In practice, it isn't. All pseudo-code I have encountered is easily translated to PlusCal. The Fast Mutual Exclusion Algorithm of Section 2.3 is typical. The PlusCal code looks very much like the pseudo-code and is just a little longer, mostly because of variable declarations. Those declarations specify the initial values of variables, which are usually missing from the pseudo-code and are explained in accompanying text. What is not typical about the Fast Mutual Exclusion example is that the pseudo-code describes the grain of atomicity. When multiprocess algorithms are described with pseudo-code, what constitutes an atomic action is usually either described in the text or else not mentioned, leaving the algorithm essentially unspecified. PlusCal forces the user to make explicit the grain of atomicity. She must explicitly tell the translator if she wants it to insert labels, which yields the largest atomic actions that PlusCal permits.

As dramatically illustrated by the quicksort *partition* example, PlusCal makes it easy to write algorithms not usually expressed in pseudo-code. The alternating bit protocol is another algorithm that is not easily written in ordinary pseudo-code. Of the first ten descriptions of the protocol found in January of 2008 by a Google search for *alternating bit protocol*, five were only in English, four were in different formal languages, and one described the processes in a pictorial finite-state machine language and the channels in English. None used pseudo-code. Of these five formal languages, all but finite-state machines were inscrutable to the casual reader. (Finite-state machines are simple, but too inexpressive to be used as an algorithm language.)

PlusCal is a language with simple program structures and arbitrary mathematical expressions. The existing programming language that most closely resembles it is SETL [17]. The SETL language provides many of the set-theoretic primitives of TLA$^+$, but it lacks the ability to define new operators mathematically; they must be described by procedures for computing them. Moreover, SETL cannot conveniently express concurrency or nondeterminism.

There are quite a few specification languages that can be used to describe and mechanically check algorithms. Many of them, including Alloy [7] and TLA$^+$ itself, lack simple programming-language constructs like semicolon and **while** that are invaluable for expressing algorithms clearly and simply. Some are more complicated than PlusCal because they are designed for sys-

tem specifications that are larger and more complicated than algorithms. Others, such as Spin [6] and SMV [15], are primarily input languages for model checkers and are little better than programming languages at describing mathematical operators. Furthermore, many of these specification methods cannot express fairness, which is an important aspect of concurrent algorithms. I know of no specification language that combines the expressiveness and simplicity of PlusCal.

The one formal language I know of that has the replacement of pseudo-code as a stated goal is AsmL, the abstract state machine language of Gurevich et al. [**?**]. It is a reasonable language for writing sequential algorithms, though its use of types and objects make it more complicated and somewhat less expressive than PlusCal. However, while AsmL has ordinary control statements like **while**, they can appear only within an atomic step. This makes AsmL unsuitable for replacing pseudo-code for multiprocess algorithms. Also, it cannot be used to express fairness.

There are a number of toy programming languages that might be used for writing algorithms. All the ones I know of that can be compiled and executed allow only the simple expressions typical of programming languages. We could look to paper languages for better constructs than PlusCal's. Perhaps the most popular proposals for novel language constructs are Dijkstra's guarded commands [2], Hoare's CSP [5], and functional languages. Guarded command constructs are easily expressed with **either/or** and **with** statements, which provide more flexibility in specifying the grain of atomicity; the lack of shared variables and dependence on a particular interprocess communication mechanism make it difficult to write algorithms like Fast Mutual Exclusion and the Alternating Bit Protocol in CSP; and I have never seen a published concurrent or distributed synchronization algorithm described functionally. As the basis for an easy-to-understand algorithm language, it is hard to justify alternatives to the familiar constructs like assignment, **if/then**, and **while** that have been used for decades and appear in the most popular programming languages.

If simplicity is the goal, why add the **await**, **with**, and **either/or** constructs that were shown in Section 4.2 to be subtle? These constructs are needed to express interprocess synchronization and nondeterminism, and there are no standard ones that can be used instead. The subtlety of these constructs comes from the inherent subtlety of the concepts they express.

Finally, one might want to use a different expression language than TLA$^+$. To achieve expressiveness and familiarity, the language should be based on ordinary mathematics—the kind taught in introductory math classes. A number of languages have been designed for expressing math-

ematics formally. I obviously prefer TLA$^+$, but others may have different preferences. A replacement for TLA$^+$ should be suitable not just as an expression language, but as a target language for a translator and as a language for expressing liveness properties, including fairness. It should also permit model checking of algorithms.

Upon being shown PlusCal, people often ask if it can be used as a programming language. One can undoubtedly define subsets of the expression language that permit compilation into reasonably efficient code. However, it is not clear if there is any good reason to do so. The features that make programming languages ill-suited to writing algorithms are there for a reason. For example, strong typing is important in a programming language; but one reason PlusCal is good for writing algorithms is the simplicity that comes from its being untyped.

PlusCal is meant to replace pseudo-code. It combines the best features of pseudo-code with the ability to catch errors by model checking. It is suitable for use in books, in articles, and in the classroom. It can also be used by programmers to debug their algorithms before implementing them.

## References


[1] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[2] Nissim Francez. *Fairness*. Texts and Monographs in Computer Science. Springer-Verlag, New York, Berlin, Heidelberg, Tokyo, 1986.

[3] Yuri Gurevich. Can abstract state machines be useful in language theory? *Theoretical Computer Science*, 376(1–2):17–29, 2007.

[4] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, July 1961.

[5] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[6] Gerard J. Holzmann. *The Spin Model Checker*. Addison-Wesley, Boston, 2004.

[7] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, April 2002.

[8] Leslie Lamport. The pluscal algorithm language. URL `http://research.microsoft.com/users/lamport/tla/pluscal.html`. The page can also be found by searching the Web for the 25-letter string obtained by removing the "-" from `uid-lamportpluscalhomepage`.

[9] Leslie Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 657–668, Paris, September 1983. IFIP, North-Holland.

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

[11] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

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

[13] Leslie Lamport. Checking a multithreaded algorithm with $^+$cal. In Shlomi Dolev, editor, *Distributed Computing: 20th International Conference, DISC 2006*, volume 4167 of *Lecture Notes in Computer Science*, pages 151–163, Berlin, Heidelberg, 2006. Springer-Verlag.

[14] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Mateo, California, 1995.

[15] K. L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[16] Vaughan R. Pratt. Semantical considerations on Floyd-Hoare logic. In *17th Symposium on Foundations of Computer Science*, pages 109–121. IEEE, October 1976.

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

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

[19] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1975.

[20] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA$^+$ specifications. In Laurence Pierre and Thomas Kropf, editors,

*Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66, Berlin, Heidelberg, New York, September 1999. Springer-Verlag. 10th IFIP wg 10.5 Advanced Research Working Conference, CHARME '99.

## Appendix: The C-Syntax Grammar

Here is a simplified BNF grammar for PlusCal's c-syntax. Terminals like `begin` are distinguished by font and are sometimes quoted like "`(`" to avoid ambiguity. The grammar omits restrictions on where labels may or must not occur, on what statements may occur in the body of a macro, and on the use of reserved tokens like `if` and `:=` in identifiers and expressions.

*Algorithm* ::= `--algorithm` *Id*
`{` [*VarDecls*] [*Definitions*] *Macro*$^*$
*Procedure*$^*$ (*CompoundStmt* | *Process*$^+$) `}`

*Definitions* ::= `define {` *Defs* `}` [`;`]

*Macro* ::= `macro` *Id* "`(`" [*Id* (`,` *Id*)$^*$] "`)`" *CompoundStmt* [`;`]

*Procedure* ::= `procedure` *Id* "`(`" [*PVarDecl* (`,` *PVarDecl*)$^*$] "`)`"
[*PVarDecls*] *CompoundStmt* [`;`]

*Process* ::= `process` "`(`" *Id* (`=` | `\in`) *Expr* "`)`"
[*VarDecls*] *CompoundStmt* [`;`]

*PVarDecls* ::= `variable`[`s`] ( *Id* [ `=` *Expr* ] (`;`|`,`) )$^+$

*VarDecls* ::= `variable`[`s`] ( *Id* [ (`=` | `\in`) *Expr* ] (`;`|`,`) )$^+$

*CompoundStmt* ::= `{` *Stmt* [`;` *Stmt*]$^*$ [`;`] `}`

*Stmt* ::= [*Id* :] (*UnlabeledStmt* | *CompoundStmt*)

*UnlabeledStmt* ::= *Assign* | *If* | *While* | *Either* | *With* | | *Await* | *Print* |
*Assert* | `skip` | `return` | *Goto* | [`call`] *Call*

*Assign* ::= *LHS* `:=` *Expr* ( "`||`" *LHS* `:=` *Expr*)$^*$

*LHS* ::= *Id* ("`[`" *Expr* (`,` *Expr*)$^*$ "`]`" | "`.`" *Id*)$^*$

*If* ::= `if` "`(`" *Expr* "`)`" *Stmt* [`else` *Stmt*]

*While* ::= `while` "`(`" *Expr* "`)`" *Stmt*

*Either* ::= `either` *Stmt* (`or` *Stmt* )$^+$

*With* ::= `with` "`(`" *Id* ( `=` | `\in` ) *Expr*
( ( `;` | `,` ) *Id* ( `=` | `\in` ) *Expr* )* [ `;` | `,` ] "`)`" *Stmt*

*Await* ::= ( `await` | `when` ) *Expr*

*Print* ::= `print` *Expr*

*Assert* ::= `assert` *Expr*

*Goto* ::= `goto` *Id*

*Call* ::= *Id* "`(`" [ *Expr* ( `,` *Expr*)* ] "`)`"

*Id* ::= A TLA$^+$ identifier (string of letters, digits, and "_"s not all digits).

*expr* ::= A TLA$^+$ expression.

*Defs* ::= A sequence of TLA$^+$ definitions.