A High-Level Language for Modeling Algorithms and their Properties

Sabina Akhtar, Stephan Merz, Martin Quinson
{Sabina.Akhtar,Stephan.Merz,Martin.Quinson}@loria.fr

LORIA – INRIA Nancy Grand Est and Nancy University, Nancy, France

Abstract. Designers of concurrent and distributed algorithms usually express them using pseudo-code. In contrast, most verification techniques are based on more mathematically-oriented formalisms such as state transition systems. This conceptual gap contributes to hinder the use of formal verification techniques. Leslie Lamport introduced PlusCal, a high-level algorithmic language that has the "look and feel" of pseudo-code, but is equipped with a precise semantics and includes a high-level expression language based on set theory. PlusCal models can be compiled to TLA$^+$ and verified using the model checker tlc. However, in practice the use of PlusCal requires good knowledge of TLA$^+$ and of the translation from PlusCal to TLA$^+$. In particular, the user needs to annotate the generated TLA$^+$ model in order to define the properties to be verified and to introduce fairness hypotheses. Moreover, the PlusCal language enforces certain restrictions that often make it difficult to express distributed algorithms in a natural way. We propose a new version of PlusCal with the aim of overcoming these limitations, and of providing a language in which algorithms and their properties can be expressed naturally. We have implemented a compiler of our language to TLA$^+$, supporting the verification of algorithms by finite-state model checking.

1 Introduction

Algorithms for concurrent and distributed systems [11] are notoriously hard to design, due to the number of interleavings of their constituent processes that must communicate and synchronize properly in order to achieve the desired function. It is all too easy to overlook corner cases, and hard to generate or reproduce particular behaviors during testing. Formal verification of such algorithms is therefore essential, and model checking in particular has been applied with great success in this context. However, there is a conceptual gap between the languages algorithm designers use to convey their ideas and the input languages of model checking tools. While the former emphasize high levels of abstraction in order to present the algorithmic ideas, their semantics is not precisely defined. Languages for model checkers come with a more precise (at least operational) semantics but tend to make compromises in terms of the available data types in order to enable compact state representations and the efficient computation

of operations such as the computation of successor (or predecessor) states. Most model checkers, in particular symbolic ones, support only low-level data types such as fixed-size integers and records. TLC [14], the model checker for the specification language TLA$^+$ [8], accepts a significant fragment of TLA$^+$, which is based on set theory; it thus provides one of the most expressive and high-level input languages for model checking. However, TLA$^+$ models encode transition systems via logical formulas, losing much of the (control) structure that is present in code.

Recently, Lamport introduced the PlusCal algorithm language [9] (originally called +Cal). While retaining the high level of abstraction of TLA$^+$ expressions, it provides familiar constructs of imperative programming languages for describing algorithms, such as processes, assignments, and control flow. The PlusCal compiler generates a TLA$^+$ model corresponding to the PlusCal algorithm, which is then verified using TLC. PlusCal is a high-level language that features set-based abstractions, non-determinism, and user-specified grain of atomicity; it emphasizes the analysis, not the efficient execution of algorithms and aims at bridging the gap that we described above.

Unfortunately, as we discuss in more detail in section 2, use of Lamport's PlusCal requires good knowledge of TLA$^+$, and even of the translation of PlusCal to TLA$^+$. Aiming at a simple translation in order to make the resulting TLA$^+$ model human readable, Lamport imposed some limitations on the language that can make it difficult or unnatural to express distributed algorithms. After initial attempts to extend the original language and its compiler, these limitations motivated us to develop a new version of PlusCal that retains the basic ideas of Lamport's language but overcomes the shortcomings that we identified. At the same time, we aim at a translation that enables the use of reduction techniques and hence more efficient verification.

2 Evaluation of PlusCal

TLA$^+$ is a very expressive specification language that emphasizes the use of high-level constructs such as sets and functions for expressing algorithms. A TLA$^+$ module contains a list of declarations, assertions, and definitions. In particular, an algorithm is specified as a formula of temporal logic that describes which executions are permitted. PlusCal retains the logical basis and the expression language of TLA$^+$, but otherwise resembles a (pseudo) programming language for multi-process programs, extended by non-deterministic constructs useful for modeling algorithms. In our experience we found that thinking in terms of sets is a strong point of PlusCal, as it can make the description of algorithms much more perspicuous. However, as we explain now, we also identified a number of shortcomings when trying to use PlusCal for modeling distributed algorithms.

Need to understand TLA$^+$ and the compilation. PlusCal models are not fully self-contained: the algorithm is described in the PlusCal language, but PlusCal can express neither the correctness properties that should be verified nor

fairness assumptions assumed about the algorithm's execution, which underly the proof of liveness properties. Rather, the user must add these as temporal logic formulas to the module generated by the PlusCal compiler. It is therefore necessary to understand not only TLA$^+$, but also the translation of PlusCal to TLA$^+$. An effort was made in the design of the PlusCal language to keep the translation simple. For example, the compiler tries to preserve the names of PlusCal variables in the TLA$^+$ specification. However, this is not always possible, for example if variables of the same name are declared in different procedures. Also, local variables of processes are represented as arrays in TLA$^+$, and the user must be aware of this when annotating the TLA$^+$ model.

Lack of process hierarchy and of scoping. Another serious restriction motivated by the need for a simple translation is that PlusCal processes can only be declared at top level, without any nesting. As we will illustrate in section 3 using Lamport's distributed mutual exclusion algorithm, many distributed algorithms are more naturally expressed using hierarchies of processes.

A related issue is the lack of scoping rules in PlusCal. Although variables may be declared locally to processes, scoping is not enforced and local variables can in fact be accessed throughout the program. Beyond being a possible cause of errors, the lack of a proper hierarchy of processes and of scoped local variables makes it much more difficult to implement optimizations for verification, such as partial-order reduction.

Restrictions in specifying atomicity. An important concern in modeling concurrent and distributed algorithms is the specification of the proper unit of atomicity: which (blocks of) statements can be considered to be executed without interleaving with statements of other processes? Whereas too coarse-grained atomicity may hide errors that arise in the implementation due to unexpected interleavings, too fine-grained atomicity introduces unnecessary details and causes state space explosion in verification. PlusCal uses a simple but powerful idea for expressing atomicity: the user may decorate statements with labels, and interleaving is only allowed at labeled statements. However, the user is not entirely free where labels may or may not be placed, as these are also serve for internal purposes of compilation. Typically, more labels must be introduced than would be necessary, hence aggravating state space explosion.

Technical limitations. Lamport's PlusCal imposes a number of other limitations, again motivated by the desire to keep the translation simple. For example, although sets are the basic construct for representing data, PlusCal does not contain a primitive for iterating over the elements of a set. The programmer has to introduce an auxiliary variable for iteration and keep track of the elements that have already been handled. Without special care, these auxiliary variables will again lead to state space explosion during model checking. Another technical restriction enforced in PlusCal is to disallow multiple assignments to the same variable within an atomic step.

3 Introducing a New Version of PlusCal

We now present in some more detail the main features of our version of PlusCal and describe its compilation to TLA$^+$. From now on, PlusCal will denote our version, except if explicitly stated otherwise.

3.1 Structure of an Algorithm

Figures 1 and 2 show a model of Lamport's mutual exlusion algorithm [6] in PlusCal. This is a basic distributed algorithm and shows some of the main ingredients of the language.

The *header section* indicates the name of the algorithm and lists any TLA$^+$ modules to be imported ("extended"). These modules contain definitions of operators that are used within the algorithm. Algorithm LamportMutex imports the modules Naturals and Sequences from the TLA$^+$ standard library. Global constant parameters (N and maxClock in our example) are also declared in the header section; these will later be instantiated to obtain a finite-state instance for verification.

The following *declaration section* contains declarations of global variables, procedures, and definitions. As in TLA$^+$, variables are untyped. A variable declaration may provide an initial value. In our example, we declare a global variable network as a two-dimensional array indexed by elements of the site Site, which contains the identities of the processes of type Site, declared below. The variable network represents the communication network; more precisely, network[from][to] is a queue of messages sent from site from to site to, initially empty ($\langle\rangle$ denotes the empty sequence in TLA$^+$). The operators send and broadcast, defined next, model point-to-point and broadcast communication over the network. Specifically, send computes the network obtained by sending a single message between two processes[1], and broadcast computes the state of the network after site from sends a message to all sites.

The main part of a PlusCal program consists of the *process section*, which introduces the processes that participate in the algorithm. Programs can declare any number of process types, and processes can be nested. Since we are mainly interested in finite-state model checking, all process instances are created at initialization time and we do not provide a mechanism for process activation at run time. In the example, we declare that N processes of type Site will be run, each containing one instance of process Communicator. Processes may be declared as being *fair*; for example, we assume (weak) fairness for each instance of process Communicator. Each process contains declarations of local variables, procedures or definitions analogously to the global declaration section. These declarations are properly scoped and visible only within the enclosing process. In our example, process Site declares variables clock, reqQ, and acks that represent the value of its logical clock, the sequence of requests it has received (which will be ordered by timestamp), and the set of acknowledgements it has received for

[1] The short-hand @ denotes the current value of the array cell being assigned to.

```
 1 algorithm LamportMutex
 2 extends Naturals, Sequences          (* standard modules *)
 3 constants N, maxClock
 4
 5 variable network = [from ∈ Site ↦ [to ∈ Site ↦ ⟨⟩]]
 6 definition send(from, to, msg) ≜
 7    [network EXCEPT ![from][to] = Append(@, msg)]
 8 definition broadcast(from, msg) ≜
 9    [network EXCEPT ![from] = [to ∈ Site ↦ Append(network[from][to], msg)]]
10
11 process Site[N]
12    variables
13       clock = 1,     (* logical clock of this site *)
14       reqQ = ⟨⟩,     (* queue of pending requests, ordered by clock values *)
15       acks = {}      (* set of acknowledgements received for own request *)
16    definition beats(rq1, rq2) ≜
17       ∨ rq1.clk < rq2.clk
18       ∨ rq1.clk = rq2.clk ∧ rq1.site < rq2.site
19    definition insertRequest(from, c) ≜
20       LET entry ≜ [site ↦ from, clk ↦ c]
21           len ≜ Len(reqQ)
22           pos ≜ CHOOSE i ∈ 1 .. len+1 :
23                     ∧ ∀ j ∈ 1 .. i-1 : beats(reqQ[j], entry)
24                     ∧ i = len+1 ∨ beats(entry, reqQ[i])
25       IN  SubSeq(reqQ, 1, pos-1) ∘ ⟨entry⟩ ∘ SubSeq(reqQ, pos, len)
26    definition removeRequest(from) ≜
27       LET len ≜ Len(reqQ)
28           pos ≜ CHOOSE i ∈ 1 .. len : reqQ[i].site = from
29       IN  SubSeq(reqQ, 1, pos-1) ∘ SubSeq(reqQ, pos+1, len)
30    definition max(x,y) ≜ IF x<y THEN y ELSE x
31
32    fair process Communicator[1]
33    begin
34       loop
35    rcv:  with from ∈ {s ∈ Site : Len(network[s][super]) > 0},
36                msg = Head(network[from][super])
37          do network[from][super] := Tail(@);
38             if msg.kind = “request”
39             then reqQ := insertRequest(from, msg.clk);
40                  clock := max(clock, msg.clk) + 1;
41                  network := send(super, from, [kind ↦ “ack”]);
42             elsif msg.kind = “ack”
43             then acks := acks ∪ {from};
44             elsif msg.kind = “free”
45             then reqQ := removeRequest(from);
46             end if;
47          end with;
48      end loop;
49    end process  (* Communicator *)
```

Fig. 1. Lamport's mutual-exclusion algorithm in extended PlusCal (part 1).

```
 1    begin (* process Site *)
 2      loop
 3    ncrit:    skip;
 4    try:      network := broadcast(self, [kind ↦ "request", clk ↦ clock]);
 5              acks := {};
 6    +enter:   when Len(reqQ) > 0 ∧ Head(reqQ.proc) = self ∧ acks = Sites;
 7    +crit:    skip;
 8    +exit:    network := broadcast(self, [kind ↦ "free"]);
 9      end loop;
10    end process (* Site *)
11  end algorithm
12
13  invariant ∀ s, t ∈ Site : Site[s]@crit ∧ Site[t]@crit ⇒ s=t
14  invariant
15    ∧ (* each queue holds at most one request per site *)
16      ∀ s ∈ Site : ∀ i ∈ 1 .. Len(reqQ[s]) : ∀ j ∈ i+1 .. Len(reqQ[s]) :
17          reqQ[s][j].site ≠ reqQ[s][i].site
18    ∧ (* requests stay in queue until "free" message received *)
19      ∀ s, t ∈ Site :
20          (∃ i ∈ 1 .. Len(network[s][t]) : network[s][t][i].kind = "free")
21          ⇒ ∃ j ∈ 1 .. Len(reqQ[t]) : reqQ[t][j].site = s
22    ∧ (* site is in critical section only if at the head of every request queue *)
23      ∀ s ∈ Site : Site[s]@crit ⇒ ∀ t ∈ Site : Head(reqQ[t]).site = s
24  temporal ∀ s ∈ Site : Site[s]@enter ⇝ Site[s]@crit
25
26  (* Finite instance for model checking *)
27  constants N = 3, maxclock = 5
28  constraint ∀ s ∈ Site : Site[s].clock ≤ maxClock
```

Fig. 2. Lamport's mutual-exclusion algorithm (part 2).

its own request (if any). Elements of the request queue are records with fields site and clock indicating the requesting site and the timestamp of the request. The definitions beats, insertRequest, and removeRequest formalize the priorities between two requests and the insertion and removal of a request in the priority queue.

The code of a process is given in the *code section* between the keywords **begin** and **end process**. We describe the statements of PlusCal in more detail in Section 3.2.

The process section is optionally followed by a code section for the main algorithm, which executes in parallel to the processes. No such code section is required for algorithm LamportMutex.

The description of the algorithm itself is followed by the *property* and *instance* sections, which state the properties (invariants and general temporal logic properties) to be verified. For our example, we state two invariants and one temporal

(liveness) property. The first invariant expresses mutual exclusion between all sites by asserting that no two sites are simultaneously at label crit. The second invariant states further safety properties of the algorithm that give more insight in its functioning. The temporal property asserts that whenever some site s is at the statement labeled enter, it will eventually access its critical section.[2] Local properties of single processes may also be stated after the code section of processes; they are verified for every instance of the corresponding process type.

The instance section of a PlusCal program determines the finite instance of the model that will be verified by the model checker TLC. In particular, the user must specify concrete values for all constants that have been declared in the header section. In our example, we instantiate the constant parameters N and maxClock by 3 and 5. This section may also contain other declarations that are interpreted by TLC. In particular, we define a constraint that bounds the state space for model checking by considering only such states where no clock value exceeds maxClock.

3.2 PlusCal Statements

The syntax of PlusCal resembles that of a standard imperative programming language, but adds non-deterministic constructs, which are useful for modeling. The expressions of PlusCal are just TLA$^+$ expressions. By focusing on high-level abstractions such as sets and functions, users are encouraged not to commit to any particular implementation early. We have found that algorithm designers quickly learn how to write TLA$^+$ expressions. This section introduces the key statements of PlusCal.

Basic statements include assignments and the **skip** statement, which has no effect. Statements can be labeled (cf. the labels ncrit, try etc. in Fig. 2). Lamport's PlusCal introduced the key idea that labels define the atomic unit of execution of a PlusCal model: a group of statements appearing between two labels is executed atomically, without interleaving by other processes. For example, reception and processing of a message is modeled as being atomic in process Communicator of Fig. 1. However, the compiler sometimes introduces additional labels when translating to TLA$^+$. For example, the first statement appearing inside a loop or statements following a procedure call must be labeled. Our compiler adds any required labels, but since every label creates an additional point of interleaving, we have added an **atomic** statement to PlusCal, which was not present in Lamport's language.

atomic B **end atomic**

The statements B in this form are executed (pseudo-)atomically, even if they contain labels. TLC can be used to find deadlocks caused by statements inside an atomic block being non-executable.

[2] The TLA$^+$ formula $P \leadsto Q$ asserts that every state satisfying predicate P will eventually be followed by a state satisfying predicate Q.

Case distinction is expressed by a standard **if** statement. There is also a **when** statement that blocks until the specified condition becomes true. Less conventionally, the statement

either B_1 **or** ... **or** B_n **end either**

can be used to express non-deterministic choice between n possible branches. In fact, the **if**, **when**, and **either** constructs are just special cases of the primitive form

branch
  P_1 **then** B_1
  P_2 **then** B_2
  ...
  P_n **then** B_n
  else B
end branch

inspired by Dijkstra's guarded commands [3]. The first n branches consist of a condition P_i and a block of statements B_i, the final **else** branch is optional. The effect of a **branch** statement is to non-deterministically choose some i such that P_i evaluates to true and execute the corresponding block B_i. If no P_i is true then the **else** branch is executed if it is present, otherwise execution blocks (and another process may be executed).

Non-deterministic choice over the elements of a set (rather than a fixed number of alternatives) is expressed by the statement

with $x \in S$ **do** B **end with**

which executes the statements B for some element of the set S, and blocks if S is empty. Since the expression S may contain program variables, executing the code of other processes may unblock a **with** statement.

PlusCal offers three iteration constructs. Beyond the standard **while** loop

while P **do** B **end while**

which is already present in Lamport's PlusCal, we added the statement

loop B **end loop**

for expressing infinite loops; this simply abbreviates **while** TRUE **do** B. More importantly, we added a loop of the form

for $x \in S$ **do** B **end for**

for iterating over the elements of a set S. (In contrast, the **with** statement mentioned above executes its body for a single, nondeterministically chosen element of S.) The order in which the elements of S are processed is unspecified but

fixed. Exploring only one iteration order helps mitigate state space explosion, but the correctness of the algorithm should not depend on any particular order.[3]

We extended PlusCal by the ability to express fairness assumptions for algorithms inside the language rather than through TLA$^+$ formulas that the user must add to the generated model. Fairness assumptions are fundamental for the verification of liveness properties. We have already mentioned (weak) fairness annotations for processes in section 3.1; these ensure that whenever a process instance is continually executable, it will eventually proceed. A **strongly fair** process is guaranteed to make progress if it is repeatedly (though not necessarily continually) executable. PlusCal also supports fairness annotations attached to individual statements (i.e., labels) rather than to entire processes. For example, the leading "+" signs decorating the labels enter, crit, and exit indicate weak fairness assumptions for the corresponding statements.

3.3 The PlusCal Compiler

The basic idea of the translation of PlusCal to TLA$^+$ is to represent each group of statements between two labels, and hence executed atomically, by a TLA$^+$ action formula. (An action formula contains unprimed and primed copies of state variables, which represent the values of these variables before and after the state transition.) Control flow is made explicit by adding a variable containing the values of the program counters of all process instances and of the main program (if present). Technically, the PlusCal compiler proceeds in three phases, as illustrated in Fig. 3.

Parser. The PlusCal parser is generated from a JavaCC grammar. Besides analyzing the algorithm for syntactic errors, the parser also constructs a symbol table, maintaining information about declared identifiers and checking that their use respects the scoping rules. This phase generates an abstract syntax tree (AST) that represents the PlusCal algorithm.

Translation to intermediate format. For clarity and modularity, we split the compilation into two phases. The first phase makes the control flow explicit and converts the AST to a list of labeled guarded commands (**branch** statements) whose branches contain only assignments. Each branch ends with an explicit assignment to the program counter of the entity executing the statements. For example, the statement

```
λ:  while P
    do B
μ:  ...
    end while
ν:  ...
```

[3] We plan to add a compile-time option to a future version of the PlusCal compiler that will cause the model checker to explore all possible iteration orders.

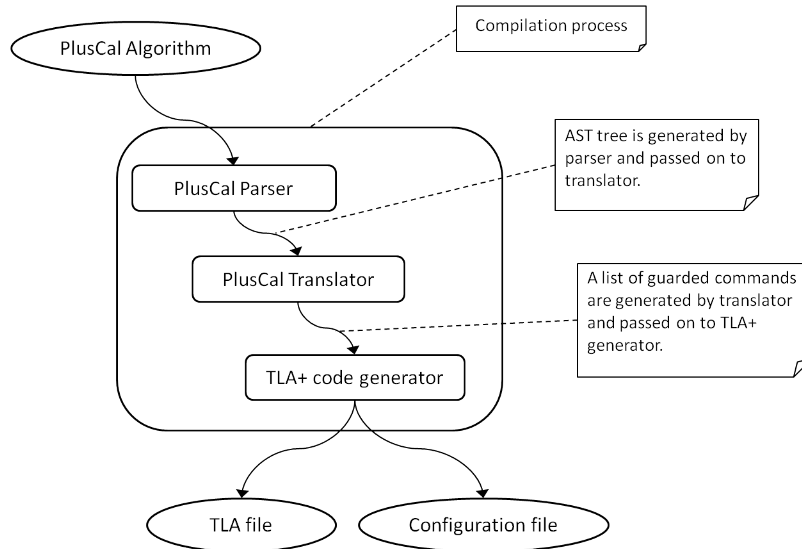

Fig. 3. The compilation phases for the new version of PlusCal.

(where there are no labels within the group of statements B) would first generate the guarded command

$\underline{\lambda}$: **branch**
  P **then** $\overline{B}$; pc[self] := μ;
  $\neg P$ **then** pc[self] := ν;
end branch

where $\overline{B}$ denotes the guarded command corresponding to B. Any nested guarded commands resulting from this translation are subsequently flattened. Procedure calls and returns are handled using an explicit run-time stack.

This translation may require additional labels, in particular for translating loops and returns from procedure calls, and the translator adds those as necessary, and signals labels added in this way to the user.

Generation of TLA$^+$ code. The final phase of compilation generates the actual TLA$^+$ model from the list of guarded commands obtained from the preceding phase. A guarded command

$\underline{\lambda}$: **branch**
  P_1 **then** x := t; y[self] := u; pc[self] := μ
  ...
  P_n **then** ...
end branch

is essentially translated to the TLA$^+$ action

$$
\begin{array}{lll}
\lambda(self) & \triangleq & \land\ pc[self] = \lambda \\
& & \land \lor \land\ P_1 \\
& & \quad\;\; \land\ x' = t \\
& & \quad\;\; \land\ y' = [y \text{ EXCEPT } ![self] = u] \\
& & \quad\;\; \land\ pc' = [pc \text{ EXCEPT } ![self] = \mu] \\
& & \quad\;\; \land \text{ UNCHANGED } vars \setminus \{x, y, pc\} \\
& & \quad \lor \ldots \\
& & \quad \lor \land\ P_n \\
& & \quad\;\; \land \ldots
\end{array}
$$

where *vars* contains all state variables. Multiple assignments to the same variable within a group of statements are handled by introducing intermediate LET-bound constants.

User-defined atomic blocks are implemented using a system-wide lock variable that is acquired at the begin of the block and released at the end. The guards of actions are strengthened appropriately: actions corresponding to statements within an atomic block test whether the lock is held by the current process, the other actions verify that the lock is free. In case some statement within an atomic block may become non-executable during the execution of the algorithm, TLC will signal a deadlock during verification.

After generating all actions corresponding to the individual transitions of the PlusCal model we define the transition relation of a process as the disjunction of the actions it may execute (including actions occurring within procedures), and the overall next-state relation as the disjunction of the transition relations for all process instances, and for the main code section if present. For the example of Figs. 1 and 2 we obtain

$$
\begin{array}{lll}
Next & \triangleq & \lor\ \exists self \in Site : _Site(self) \\
& & \lor\ \exists self \in Communicator : _Communicator(self)
\end{array}
$$

where *Site* and *Communicator* are sets containing the process identifiers of processes of type Site and Communicator, and *_Site* and *_Communicator* are the actions representing the transitions of these processes. Fairness conditions are generated from the fairness annotations present in the PlusCal model, e.g.

$$
\begin{array}{lll}
Fairness & \triangleq & \land\ \forall self \in Communicator : \mathrm{WF}_{vars}(_Communicator(self)) \\
& & \land\ \forall self \in Site : \land\ \mathrm{WF}_{vars}(enter(self)) \\
& & \qquad\qquad\qquad\quad\; \land\ \mathrm{WF}_{vars}(crit(self)) \\
& & \qquad\qquad\qquad\quad\; \land\ \mathrm{WF}_{vars}(exit(self))
\end{array}
$$

and the overall specification is obtained as

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

Finally, the properties and the instance sections of the PlusCal model are processed in order to generate the configuration file, which defines the finite-state instance and indicates the properties to be verified with TLC.

3.4 Comparison with Lamport's PlusCal

The language that we have presented in this paper was inspired by Lamport's PlusCal, to which it remains close in spirit, but it attempts to overcome some of the deficiencies that we have identified in section 2. We briefly comment on what we believe are the main advantages of our language.

Self-contained models. Models written for the original PlusCal language can express only the algorithm. All additional information, such as fairness assumptions, correctness properties or model checking constraints have to be manually inserted into the TLA$^+$ model generated by the compiler, requiring the user to understand not only TLA$^+$ but also the translation. We do not expect users to understand our compiler in any detail, or even to read the generated TLA$^+$ file.

Nested processes and scoped declarations. We allow for nested process declarations, and this leads to a clearer representation of the (communication) structure of algorithms. In our running example, we were able to declare the variables of each site as local variables, with two threads (the main Site process and the Communicator) accessing them. In the original PlusCal, one would either have two top-level process types that need to communicate via global variables (which then must be declared explicitly as arrays by the user) or insert the message-handling code between all transitions of the Site process. In either case, the model becomes hard to understand, contradicting the purpose of a high-level modeling language.

Unlike the original PlusCal, our compiler enforces proper scoping of variables, procedures, and definitions, avoiding potential errors by inadvertently accessing the variables of a different process. In future work, we plan to take advantage of this locality information in order to implement partial-order reductions for optimizing model checking.

More flexibility. As discussed in Section 3.2, the basic idea in PlusCal is to specify atomicity via labels. We managed to lift some of the restrictions on label placement that were present in the original PlusCal language, and our compiler will add labels when they are required. The user can now enforce atomicity of code blocks containing labels using the new **atomic** statement.

We also introduced several extensions, such as the **for** statement for iterating over a set, or the possibility to have several assignments to the same variable within a group of statements. On the technical side, we strived for better modularity of translation so that it becomes easier to experiment with new language primitives.

While our PlusCal variant retains most of the "look and feel" of the original PlusCal language, it does not guarantee backward compatibility. For example, programs that modify variables that are not currently in scope will be rejected by the new PlusCal compiler. The current version also does not provide macros that exist in Lamport's PlusCal.

Table 1. Number of states for some algorithms.

Algorithm	# proc.	original PlusCal	our PlusCal
Peterson	2	37	23
FastMutex	2	2679	2679
Naimi-Trehel	3	111749	53905

There are many features that we deliberately did not implement. For example, PlusCal does not provide primitives for message passing between processes. Distributed algorithms use many different forms of message passing (synchronous or not, lossy, duplicating, order preserving, ...), and these are better defined in a standard library of procedures or definitions than hard-wired into the language.

4 Experiments

We have tested our language and implementation by modeling several concurrent and distributed algorithms in it and verifying them using TLC. Our experience so far has been quite satisfactory: we found that we could represent the algorithms in a natural way and never had to touch the generated TLA$^+$ models. Table 1 shows the number of (distinct) states generated by TLC for the original PlusCal and the new PlusCal models of three algorithms: Peterson's algorithm [13], a model of which is included in the original PlusCal distribution, Lamport's FastMutex algorithm [7], a model of which appears in the PlusCal reference manual [10], and the distributed mutual-exclusion algorithm due to Naimi and Trehel [12], which is a refinement of Lamport's algorithm shown in Figs. 1 and 2. Models of all but the most trivial algorithms, and in particular of distributed algorithms, tend to be much clearer in our version of PlusCal. The numbers in Table 1 indicate that the added expressiveness does not come at the expense of lost efficiency in verification, as the state spaces generated from the new PlusCal models are not bigger than those for the same algorithm written in the original PlusCal.[4] In some cases, we obtain smaller numbers of states because of lifted labeling restrictions. The running times of TLC for these small examples never exceeded a few seconds. In future work, we believe that we can achieve significant improvements by exploiting the information about locality in PlusCal for implementing partial-order reductions.

The simplicity of translation to TLA$^+$ was an important design objective for the original PlusCal language. In particular, it is tolerable that counter-examples generated by the model checker are displayed in terms of the generated TLA$^+$ model, which the user has to understand anyway. Some of our extensions, and in particular nested processes with local variables, complicate the translation and the interpretation of counter-examples. We intend to implement a filter for displaying counter-examples in terms of the original PlusCal model.

[4] Moreover, handwritten TLA$^+$ models of these algorithms that have comparable "grain of atomicity" generate similar numbers of states.

5 Related Work

There are many languages for modeling concurrent and distributed algorithms. Promela [4] is the modeling language for verification of distributed systems using the Spin model checker. A Promela model consists of processes, channels and variables. Promela does not support nested processes, has fixed primitives for communication, and rather low-level representations of data (fixed-width subsets of integers, records, and channels). It is therefore better suited to lower-level descriptions of algorithms and protocols. On the other hand, Spin offers more efficient verification techniques than tlc.

LOTOS [1] (Language of Temporal Ordering Specifications) is a formalism for specifying distributed systems, specifically related to Open Systems Interconnection (OSI) computer network architecture. Estelle [2] is another formal description technique for writing specification for concurrent and distributed information processing systems. It is based on Extended State Transistion systems and is supported by industrial-strength tools. Both languages are similar in purpose to Promela, whereas we are aiming at obtaining higher-level descriptions of algorithms, for which abstract data representations in terms of sets and functions are more useful.

There exist many other languages that are closer to the programming languages rather than formal specification languages. They serve as inputs to verification tools and/or for generating executable code. For example, Mace [5] is a language for building distributed systems. It is a C++ language extension that translates distributed system specifications into a C++ implementation. Model checking can be performed at a higher level using the MaceMC model checker. In contrast, PlusCal is intended as a language that algorithm designers use to communicate (and validate) their ideas, not for generating efficient implementations.

6 Conclusion

PlusCal is a high-level language that aims at natural expression of algorithms; it makes formal verification easily accessible to algorithm designers. We have identified certain limitations of the original language and have defined a new version that tries to overcome them. In particular, we have strived at making algorithm descriptions entirely self-contained, so that knowledge of TLA$^+$, and in particular of the PlusCal compiler, is no longer a prerequisite for using PlusCal. We have also made the language more uniform, removing some limitations and adding features such as nested processes, scoped declarations, and user-defined grain of atomicity. We believe that the new version significantly simplifies the representation of algorithms in PlusCal and that it will be more accessible to users who are not experts in formal methods.

In future work, we are planning to address reduction techniques for mitigating the effect of state space explosion. In particular, we plan to implement partial-order reduction for verifying models written in PlusCal. In concurrent and

distributed systems, the execution of independent events in an arbitrary order results in the same overall system state, and it is therefore enough to consider only one interleaving of such events. Efficiently verifying that two events are independent is, however, non-trivial, and adding locality to PlusCal was an important first step in identifying sufficient conditions for two statements being independent.

On a more technical level, it would be beneficial to translate counter-examples produced by TLC back to the level of PlusCal programs in order to make them easier to understand for PlusCal users. We also plan to integrate our PlusCal language into the TLA$^+$ toolbox that has recently been released.[5]

Acknowledgement. We are grateful to Leslie Lamport for discussions on the design of our variant of PlusCal and for his encouragement of our project.

References

1. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks*, 14, 1987.
2. S. Budkowski and P. Dembinski. An introduction to Estelle: A specification language for distributed systems. *Comput. Netw. ISDN Syst.*, 14(1):3–23, 1987.
3. E. W. Dijkstra. Guarded commands, non-determinacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.
4. G. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, 2004.
5. C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. Vahdat. Mace: Language Support for Building Distributed Systems. In *PLDI*, pages 179–188, 2007.
6. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
7. L. Lamport. A fast mutual-exclusion algorithm. *ACM Trans. Computer Systems*, 5(1):1–11, 1987.
8. L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.
9. L. Lamport. Checking a multithreaded algorithm with +CAL. In S. Dolev, editor, *20th Intl. Symp. Distributed Computing (DISC 2006)*, volume 4167 of *LNCS*, pages 151–163, Stockholm, Sweden, 2006. Springer.
10. L. Lamport. A +CAL user's manual. `http://research.microsoft.com/en-us/um/people/lamport/tla/pluscal.html`, 2007.
11. N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.
12. M. Naimi, M. Trehel, and A. Arnold. A log(n) distributed mutual exclusion algorithm based on path reversal. *J. Parallel Distrib. Comput.*, 34(1):1–13, 1996.
13. G. L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.
14. Y. Yu, P. Manolios, and L. Lamport. Model checking TLA+ specifications. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *LNCS*, pages 54–66, Bad Herrenalb, Germany, 1999. Springer.

[5] `http://www.tlaplus.net/`