Extending PlusCal: A Language for Describing Concurrent and Distributed Algorithms

Sabina Akhtar, Stephan Merz, Martin Quinson

Extending PlusCal: A Language for Describing Concurrent and Distributed Algorithms

Sabina Akhtar, Stephan Merz, Martin Quinson

LORIA – INRIA Nancy Grand Est & Nancy University, Nancy, France
{Sabina.Akhtar,Stephan.Merz,Martin.Quinson}@loria.fr

1 Context

The design of correct concurrent and distributed algorithms is notoriously difficult, and they are prone to errors such as deadlocks and race conditions. In fact, it is often non-trivial to precisely state the assumptions under which the algorithms are expected to operate and the exact properties they are expected to guarantee. Model checking [1] is one of the most successful formal techniques in this area: properties (usually expressed in temporal logic) can be verified automatically over instances of algorithms whose state space is finite (or finitely representable). In case the property does not hold, the model checker produces a counter-example, whose inspection assists in finding the cause of the error.

Model checkers expect that algorithms be expressed in a formal modeling language. For example, TLC [5] accepts models written in TLA$^+$ [3], a specification language based on mathematical set theory and the Temporal Logic of Actions. Such languages have a very different flavor from the (pseudo-)programming languages that algorithm designers typically use to express algorithms, and that mismatch creates a barrier for the routine use of model checking. Recognizing this problem, Lamport introduced PlusCal [4], a high-level language for describing concurrent and distributed algorithms, from which TLA$^+$ models are generated and then analyzed using TLC.

Unfortunately, PlusCal suffers from a number of limitations that restricts its usefulness as an abstraction layer for the underlying TLA$^+$ formalism. Most importantly, assumptions (such as fairness) and correctness properties cannot be expressed in PlusCal but have to be stated in terms of the TLA$^+$ model that results from the translation. Hence, the user not only has to be knowledgeable in TLA$^+$, but also has to understand the translation of PlusCal to TLA$^+$. In turn, the translation has to be relatively straightforward so that its result remains human-readable, and this motivates several restrictions of PlusCal. For example, PlusCal algorithms are restricted to only top-level processes, whereas many distributed algorithms are naturally expressed in terms of nodes communicating by message passing, each of which contains several threads that access shared memory. PlusCal does not enforce variable scoping, which may result in uncaught modeling errors such as nodes inadvertently accessing variables of distant nodes.

One of the fundamental concepts of parallelism is to indicate the unit of atomicity. PlusCal employs a simple, but powerful idea: statements can be labeled, and all code appearing between two labels is assumed to be executed atomically. However, labels are also introduced for the purposes of compilation, for example when translating loops or procedure calls, and this leads to rather complex rules governing where labels can and must be placed.

We propose a variant of PlusCal that preserves the basic objectives of the language while overcoming the restrictions mentioned above. In the remainder of this contribution we sketch the main design decisions for our language extensions.

2 Our Contributions

Our language, just like PlusCal, is based on TLA$^+$. In particular, the objects and data structures that algorithms manipulate are represented by TLA$^+$ expressions. We found that working with set-theoretical abstractions helps clarifying the fundamental concepts underlying concurrent and distributed algorithms. The resulting expressive power is incomparably higher than that afforded by conventional modeling languages such as Promela [2]. Unlike PlusCal, representations of algorithms using our system are en-

tirely self-contained.[1] Fairness hypotheses are indicated by appropriate annotations of processes or labeled statements, and the compiler generates corresponding formulas in the TLA$^+$ specification. Also, our system allows models to contain an *instance section* that defines the finite instance to be verified by TLC, and from which a configuration file is generated. Third, correctness properties can be stated at the level of individual processes (verified for each instance of the process) and for the entire algorithm.

Processes can be arbitrarily nested in our language, and the compiler enforces static scoping of variable or procedure declarations, as well as of TLA$^+$ operator definitions appearing in processes. (Global variables and procedures are accessible throughout an algorithm.) Hierarchical processes more naturally reflect the structures of actual algorithms; static scoping helps limiting modeling errors and opens the way to optimizations during verification such as partial-order reduction. These changes complicate the translation and make the resulting TLA$^+$ model harder to read and understand. Because models using our language extensions are self-contained, we do not expect users to read or edit the generated TLA$^+$ specification.

Our language extension retains the basic idea of indicating atomicity via labels. Whenever additional labels are required for the purposes of translation, the compiler adds them and informs the user. We have added an explicit *atomic* construct that makes a group of statements execute without interruption by other processes, even in the presence of intervening labels.

We introduced a number of relatively minor differences to PLUSCAL. For example, we allow several assignments to the same variable to appear within a group of statements without an intervening label. Also, we have added a *for* statement for iterating over the elements of a finite set in an unspecified order. Our language is not entirely backward-compatible with PLUSCAL: for example, accesses to variables that are not in static scope are rejected. We have also replaced the general, but unscoped macro facility of PLUSCAL by a more restrictive concept of locally scoped *definitions*. Nevertheless, we have found that existing PLUSCAL algorithms can be adapted with minor effort.

We have implemented a compiler that translates algorithms written in our language to TLA$^+$ and have successfully encoded and verified a number of representative algorithms. We found it easy to describe algorithms and specify their properties using our language extensions. The fact that fairness annotations appear within the algorithms gives particular expressive power to designers and lets them explore the liveness properties guaranteed by algorithms.

In case a property fails to hold, the counter-example is currently presented by TLC in terms of the TLA$^+$ model that results from translation. Understanding counter-examples could already be difficult for the original PLUSCAL, it is even more so for our extensions, because the compilation is more involved. We plan to translate counter-examples back to the PLUSCAL level and to present them within the GUI that is currently being developed for the TLA$^+$ tools. We also interact with the authors of PLUSCAL to integrate our changes into their public distribution. Another important problem we are working on is addressing the problem of state space explosion based on partial order reduction techniques. Static scoping of variables local to (nested) processes is a prerequisite here because it helps us to detect locality and independence of TLA$^+$ actions that represent a block of PLUSCAL statements.

References

1. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, Mass., 1999.
2. G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.
3. L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, 1987.
4. L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
5. 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.
6. 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.

[1] A complete example of a model using our language extensions appears in the appendix.

Example: Representation of the FastMutex algorithm

The following is a model of Lamport's fast mutual exclusion algorithm [**?**] in our extension of PLUSCAL. Note in particular the use of fairness annotations for processes and the fact that properties are given within the model.

```
algorithm FastMutex
extends Naturals                    (* Standard module of TLA+ *)
constants
   N   (* Number of processes *)
variables
      x = 0,
      y = 0,
      b = [id ∈Peer ↦ FALSE]
fair process Peer[N]
   begin
      ncs:  loop
                skip ;              (* Non-critical Section *)
      start:    b[self] := TRUE ;
      l1:       x := self ;
      l2:       if y # 0 then
      l3:           b[self] := FALSE ;
      l4:           when y = 0;
                    goto start;
                end if;
      l5:       y := self ;
      l6:       if x # self then
      l7:           b[self] := FALSE ;
      l8:           for j ∈ 1 .. N
                        when ¬b[j];
                    end for;
      l9:           if y ≠ self then
      l10:              when y = 0; goto start;
                    end if;
                end if;
      cs:       skip;               (* Critical Section *)
      l11:      y := 0 ;
      l12:      b[self] := FALSE
            end loop;
   end process
end algorithm
(* Assert: No two processes simultaneously execute the critical section *)
invariant ∀ i, k ∈ 1..N: i ≠ k ⇒ ¬(Peer[i]@cs ∧ Peer[k]@cs)
(* Liveness: Each request for the lock is eventually granted *)
temporal ∀ p ∈ Peer: □◊¬b[p]
(* Instantiating the model for 3 processes. *)
constants
      N = 3
```