An Extension of PlusCal for Modeling Distributed Algorithms

Heba Alkayed, Horatiu Cirstea, Stephan Merz

An Extension of PlusCal for Modeling Distributed Algorithms

Heba Alkayed, Horatiu Cirstea, Stephan Merz
University of Lorraine, CNRS, Inria, Nancy, France

1 Motivations

The PlusCal language [3, 4] combines the expressive power of TLA$^+$ [2] with the "look and feel" of imperative pseudo-code in order to allow users to express algorithms at a high level of abstraction. PlusCal algorithms are translated to TLA$^+$ specifications and can be formally verified using the TLA$^+$ Toolbox. We propose a small extension of PlusCal, tentatively called Distributed PlusCal [1], intended for simplifying the presentation of distributed algorithms in PlusCal.

Distributed systems consist of nodes that communicate by message passing. It is convenient to model a node as running several threads that share local memory. For example, one thread may execute the main algorithm, while a separate thread listens for incoming messages. Although PlusCal offers *processes*, they have a single thread of execution. Different threads of the same node must therefore be modeled as individual processes, and variables representing the local memory of a node must be declared as global variables, obscuring the structure of the code. Our first extension allows a PlusCal process to have several code blocks that execute in parallel. Besides, Distributed PlusCal explicitly identifies variables representing communication channels and introduces associated send and receive operations. In contrast to using ordinary variables and writing macros or operator definitions for channel operations, making channels part of the language gives us some more flexibility in the TLA$^+$ translation.

2 Distributed PlusCal Algorithms

Distributed PlusCal extends the syntax of PlusCal in two places, as shown in Figure 1. In addition to *variables*, the declaration section may contain *channel* and *fifo* declarations. These represent (arrays of) communication channels, with the second kind of channels guaranteeing FIFO communication. Moreover, a process may have several sub-processes. Each sub-process contains statements (a *CompoundStmt* according to the PlusCal BNF syntax), they are executed in parallel and may refer to the variables declared in the process.

We added an option `-distpcal` to the PlusCal translator in order to switch between regular and Distributed PlusCal.

```
(* --algorithm <algorithm name>
(* Declaration section *)
variables <variable declarations>
channels <channel declarations>
fifos <fifo declarations>
(* ... *)
(* Processes section *)
process (<name> [=|\in] <Expr>))
  variables <variable declarations>
  <subprocesses>
*)
```

Figure 1: Syntactic extensions introduced by Distributed PlusCal.

3 Communication Channels

The syntax for a channel declaration, introduced with the keyword **channel** or **channels**, is shown below.

$$\textbf{channel}\ \langle id \rangle [\langle Expr_1 \rangle, \ldots, \langle Expr_N \rangle];$$

This declaration introduces an N-dimensional matrix of unordered channels indexed by the sets $\langle Expr_i \rangle$, which may be omitted for a simple channel. It gives rise to the following conjunct in the initial condition of the corresponding TLA$^+$ specification

$$id = [x1 \in Expr_1, \ldots, xN \in Expr_N \mapsto \{\}];$$

or just $id = \{\}$ for a simple channel. A FIFO channel is similarly declared with the keyword **fifo** or **fifos** and is initialized to a matrix of empty sequences.

Distributed PlusCal supports the following operations on (unordered or FIFO) channels: $send(ch, e)$ sends a single value e on a channel, $receive(ch, var)$ is enabled when ch is non-empty and receives a message into variable var, $clear(ch)$ empties the channel, and

$$broadcast(ch, [x \in S \mapsto e(x)]) \quad \text{and} \quad multicast(ch, [x \in S \mapsto e(x)])$$

send messages along several channels in an array. For the latter two operations, if ch is a (one-dimensional) array of channels, S is expected to be the domain of the array for broadcast and a subset of the domain for multicast.

4 Subprocesses

A process can have multiple sub-processes. In the C-Syntax, each sub-process appears within a pair of curly braces, whereas in the P-Syntax, sub-processes are enclosed by `begin subprocess` and `end subprocess`. Since a process may have several threads of execution, the *pc* variable is represented as a two-dimensional array indexed by process identity and sub-process number. For example, the translation of the statement labeled `exit` of the mutual-exclusion algorithm of Figure 2 is shown below.

```
exit(self) ==
  /\ pc[self][1] = "exit"
  /\ clock' = [clock EXCEPT ![self] = clock[self] + 1]
  /\ network' = [<<slf, n>> \in DOMAIN network |->
                  IF slf = self /\ n \in Nodes \ { self }
                  THEN Append(network[slf, n], Release(clock'[self]))
                  ELSE network[slf, n]]
  /\ pc' = [pc EXCEPT ![self][1] = "ncs"]]
  /\ UNCHANGED << req, ack, sndr, msg >>
```

Moreover, the translation of a procedure call stores the identity of the sub-process on the call stack so that control returns to the appropriate sub-process.

5 Evaluation

Distributed PlusCal is designed to remain backward compatible with regular PlusCal: the translation of a regular PlusCal algorithm gives rise to a TLA$^+$ specification that is equivalent with the one produced by the existing translator.

Our version of Lamport's mutual-exclusion algorithm shown in Figure 2 illustrates the representation of distributed algorithms in Distributed PlusCal. We believe that the possibility of declaring several threads per process makes expressing such algorithms more natural. Distributed algorithms employ many kinds of communication channels beyond unordered and FIFO channels, and we envisage providing different semantics through standard TLA$^+$ modules that can be instantiated, rather than baking two kinds of channels into the language.

Beyond writing a fixed number of sub-processes, one could envisage extending PlusCal by identical sub-processes indexed by a parameter set. This could perhaps be useful for modeling a node containing several CPU and GPU cores.

References

[1] Heba Al-kayed. Distributed PlusCal. `https://github.com/hibaalkayed/DistPcalTranslate`.

[2] Leslie Lamport. *Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, USA, 2002.

[3] Leslie Lamport. The PlusCal algorithm language. In M. Leucker and C. Morgan, editors, *6th Intl. Coll. Theor. Asp. Comp. (ICTAC 2009)*, volume 5684 of *LNCS*, pages 36–60, Kuala Lumpur, Malaysia, 2009. Springer.

[4] Hillel Wayne. *Practical TLA*. Apress, 2018. `https://learntla.com/`.

```
------------------------ MODULE LamportMutex -------------------------
EXTENDS Naturals, Sequences, TLC
CONSTANT N
ASSUME N \in Nat
Nodes == 1 .. N
(* PlusCal options (-distpcal) *)
(**--algorithm LamportMutex {
   fifos network[Nodes, Nodes];
   define {
     Max(c,d) == IF c > d THEN c ELSE d
     beats(a,b) == \/ req[b] = 0
                   \/ req[a] < req[b] \/ (req[a] = req[b] /\ a < b)
     \* messages used in the algorithm
     Request(c) == [type |-> "request", clock |-> c]
     Release(c) == [type |-> "release", clock |-> c]
     Acknowledge(c) == [type |-> "ack", clock |-> c]
   }
   process(n \in Nodes)
     variables clock = 0, req = [n \in Nodes |-> 0],
               ack = {}, sndr, msg;
   { \* thread executing the main algorithm
ncs: while (TRUE) {
       skip;  \* non-critical section
try:   clock := clock + 1; req[self] := clock; ack := {self};
       multicast(network, [self, nd \in Nodes |-> Request(clock)]);
enter: await (ack = Nodes /\ \A n \in Nodes \ {self} : beats(self, n));
cs:    skip;  \* critical section
exit:  clock := clock + 1;
       multicast(network, [self, n \in Nodes \ {self} |->
                           Release(clock)]);
     } \* end while
  }  { \* message handling thread
rcv:   while (TRUE) { with (n \in Nodes) {
           receive(network[n,self], msg); sndr := n;
           clock := Max(clock, msg.clock) + 1
        };
handle: if (msg.type = "request") {
           req[sndr] := msg.clock;
           send(network[self, sndr], Acknowledge(clock))
        }
        else if (msg.type = "ack") { ack := ack \cup {sndr}; }
        else if (msg.type = "release") { req[sndr] := 0; }
     }  \* end while
   } \* end message handling thread
}  **)
======================================================================
```

Figure 2: Lamport's mutual-exclusion algorithm.