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 [?, ?] combines the expressive power of TLA$^+$ [?] 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 [?], 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 Introduction

Distributed systems are based on continuous interactions among components, these interactions produce bugs that are difficult to find by testing as they tend to be non-reproducible or not covered by test-cases. In order to test failure conditions such as deadlocks and race conditions formal verification methods have been employed successfully.

Model Checking [?] is a model-based verification technique. Model checking starts with a model described by the user and checks if the properties asserted by the user are valid on that model.

TLA$^+$ is a formal language used to describe algorithms, it provides a flexibility and an expressiveness that enables it to specify and verify complicated algorithms concisely. One of the popular modern examples of incorporating TLA$^+$ to verify distributed algorithms is its usage at Amazon Web Services [**?**].

Once the user has modeled the system using the TLA$^+$ specification language, the user can then use the Temporal Logic Checker(TLC) to check both the safety and liveness properties. TLC can further speed-up model checking by running on virtual machines using cloud-based distributed TLC [**?**]. The TLC model checker is offered with the TLA$^+$ Toolbox.

In addition, the TLA$^+$ Toolbox also provides a translator from the algorithmic language PlusCal to TLA$^+$. The PlusCal language was designed to provides simple pseudo-code like interface for the user to express concurrent systems. It maintains the expressiveness of TLA$^+$ while providing the user with a more familiar syntax.

Although PlusCal is very useful, when it comes to distributed algorithms it may enforce some limitation that make it difficult to express them in a natural way. As an example, distributed algorithms usually need to model several subprocesses coexisting and communicating as a part of a distributed node, PlusCal processes must all be declared at top level and cannot be easily used to create what the algorithm is aiming for.

3 Model checking using TLA$^+$

3.1 TLA$^+$ Specification Language

TLA$^+$ is a formal specification language in which algorithms and systems can be described at a high level of abstraction and can be formally verified using the model checker TLC or the interactive proof assistant TLAPS. TLA$^+$ is based on mathematical set theory for describing data structures in terms of sets and functions, and on the Temporal Logic of Actions TLA for specifying their executions as state machines. TLA$^+$ specifications usually have the form

$$Init \wedge \Box[Next]_{vars} \wedge L$$

where $Init$ is a predicate describing the possible initial states, $Next$ is a predicate that constrains the possible state transitions, $vars$ is the tuple of all state variables that appear in the specification, and L is a liveness or fairness property expressed as a formula of temporal logic. Transition formulas such as $Next$, also called *actions*, are at the core of TLA$^+$, and represent instantaneous state changes. They contain unprimed state variables denoting the value of the variable before the transition as well as primed state variables that denote the value after the transition.

For example, Figure 1 shows a TLA$^+$ specification of a simple memory. It declares four constant parameters `Address, Value, InitValue,` and `NoValue`, and states a hypothesis on the values that these parameters can be instantiated with. The state space of the specification is represented by the two variables

chan and `mem`. Intuitively, `mem` holds the current memory, whereas `chan` is an output channel that reflects the result of the preceding operation.

The remainder of the TLA$^+$ module contains operator definitions that represent parts of the specification and of correctness properties. The state predicate `Init` fixes the initial values of the two variables. The actions `Read(a)` and `Write(a,v)` represent reading the value at memory address `a` and writing value `v` to memory address `a`, respectively. In this specification, the memory is modeled as a function mapping addresses to values. The TLA$^+$ expression $[x \in S \mapsto e]$ denotes the function with domain `S` such that every element `x` of `S` is mapped to `e`. This is reminiscent of a λ-expression but also makes explicit the domain of the function. Function application `f[x]` is written using square brackets. Finally, the expression $[f \text{ EXCEPT } ![x] = e]$ denotes the function that is similar to f, except that argument `x` is mapped to `e`, one can think of it as a function overwrite.

The action `Next` defines the possible state transitions as the disjunction of `Read` and `Write` actions, and `Spec` represents the overall specification of the memory.

TLA$^+$ is an untyped language. Type correctness can be verified as a property of the specification. For our example, the predicate `TypeOK` indicates the possible values that the variables `chan` and `mem` are expected to hold at any state of the specification. Formally, the implication `Spec` $\Rightarrow \Box$ `TypeOK` can be established as a theorem.

For More details on the syntax and grammar of TLA$^+$, see [?] [?].

3.2 PlusCal Algorithmic Langage

PlusCal [?] was designed as an algorithm language with a more familiar syntax that can be translated into TLA$^+$ specifications and then be verified using the familiar TLA$^+$ tools.

PlusCal is an algorithm language that describes both concurrent and sequential algorithms, it maintains the expressiveness of TLA$^+$ as well as representing atomicity conveniently.

The TLA$^+$ Toolbox provides a platform where algorithm designers can model their algorithms using PlusCal, translate them to the corresponding TLA$^+$ specifications and check for the algorithm's correctness through the TLC model checker.

A PlusCal algorithm is located in a comment statement within the TLA$^+$ module. The general structure of a PlusCal algorithm is shown in Figure 2.

The **Declaration section** is where the user declares global variables that are shared among all the components of the algorithm. The **Definition section** allows the user to write TLA$^+$ definitions of operators that may refer to the algorithm's global variables. The **Macro section** holds macros whose bodies are expanded at translation time incorporating the parameters passed from the calling statement, similar to the expansion of C pre-processing macros. A **procedure** in PlusCal take a number of arguments, can declare local variables,

and can modify the global variables; it does not return a result. A **process** begins in one of two ways:

$$process(ProcName \in IdSet)$$
$$process(ProcName = Id)$$

The first form declares a set of processes, the second an individual process. these statements are optionally followed by declarations of local variables. The process body is a sequence of statements, within the body of a process set, $self$ equals the current process's identifier.

Procedure and process bodies may contain labels. All PlusCal statements appearing between two labels are executed atomically, and certain rules determine where labels must and may not appear. PlusCal enforces a strict ordering of its blocks. The define block has to come before any macros, which has to come before any procedures, which has to come before any processes. The full grammar of the PlusCal algorithm language can be found in appendix A of the PlusCal manual [**?**].

Figure 3 shows the modeling of a semaphore mutex example in PlusCal, Semaphores are integer variables that are used to solve the critical section problem.

```
---------------------------- MODULE SimpleMemory ----------------------------
CONSTANTS Address, Value, InitValue, NoValue

ASSUME
  /\ InitValue \in Value
  /\ NoValue \notin Value

VARIABLES chan, mem

\* initial condition
Init ==
  /\ chan = NoValue
  /\ mem = [a \in Address |-> InitValue]

\* transitions: reading and writing
Read(a) ==
  /\ chan' = mem[a]
  /\ mem' = mem

Write(a,v) ==
  /\ mem' = [mem EXCEPT ![a] = v]
  /\ chan' = NoValue

Next ==
  \/ \E a \in Address : Read(a)
  \/ \E a \in Address, v \in Value : Write(a,v)

\* overall specification
Spec == Init /\ [][Next]_<<chan,mem>>

\* predicate specifying type correctness
TypeOK ==
  /\ chan \in Value \cup {NoValue}
  /\ mem \in [Address -> Value]
==========================================================================
```

Figure 1: A memory specification in TLA$^+$.

```
(** algorithm <algorithm name>

(* Declaration section *)
variables <variable declarations>

(* Definition section *)
define <definition name> == <definition description>

(* Macro section *)
macro <name>(var1, ...)
<macro body of statements>

(* Procedure section *)
procedure <name>(arg1, ...)
 variables <local variable declarations>
 <procedure body of statements>

(* Processes section *)
process (<name> [=|\in] <expr>))
  variables <variable declarations>
   <process body of statements>

**)
```

Figure 2: General structure of a PlusCal algorithm

```
(*
--algorithm SemaphoreMutex {
variables sem = 1;

  process(p \in 1..N)
  {
  start : while (TRUE){
  enter :    when (sem > 0);
                  sem := sem - 1;
  cs :       skip ;
  exit :     sem := sem + 1 ;
          }
  }
}
*)
```

Figure 3: Semaphore mutex example in PlusCal

3.3 Translation to TLA$^+$.

The PlusCal translator expects as input a PlusCal algorithm following the structure described previously.

The translator parses the PlusCal algorithm and generates the corresponding TLA$^+$ specification in roughly the following steps.

1. Generate all the definitions and variables regardless of their scope within the PlusCal algorithm, as well as *vars* the tuple of all variables. The `pc` variable is introduced by the translator to track the control flow of processes.

```
\* BEGIN TRANSLATION
VARIABLES sem, pc

vars == << sem, pc >>
```

2. Generate `ProcSet` which is a set that contains all the process identifiers.

```
ProcSet == (1..N)
```

3. Generate `Init`, the initial predicate that specifies the initial values of all the declared variables. Comments indicate if the variables are global or local to a process or procedure. The variable `pc` is defined and used as a program control variable, it's a function whose domain is `ProcSet` such that each element is mapped to the entry label of the process.

```
Init == (* Global variables *)
        /\ sem = 1
        /\ pc = [self \in ProcSet |-> "start"]
```

4. For each PlusCal label, generate a TLA$^+$ action that represents the atomic operation beginning at that label. In the produced actions unprimed variables refer to their values before executing the action and the primed variables refer to their values after the execution. The definition is parameterized by the identifier `self`, which represents the identifier for the current process. For example, the following action is generated for label `enter` of the semaphore algorithm.

```
enter(self) == /\ pc[self] = "enter"
               /\ (sem > 0)
               /\ sem' = sem - 1
               /\ pc' = [pc EXCEPT ![self] = "cs" ]
```

   Moreover, the PlusCal translator generates an action that corresponds to the disjunction of the actions for the individual labels and that represents the transition relation of a process.

```
p(self) == start(self) \/ enter(self) \/ cs(self) \/ exit(self)
```

5. Generate the next-state action *Next* and the complete specification *Spec*

```
Next == (\E self \in 1..N: p(self))

Spec == Init /\ [][Next]_vars
```

In practice, to use PlusCal the user must understand the generated TLA$^+$ specifications, in order to write the properties in terms of the TLA$^+$ variables introduced by the compiler.

A more detailed description of the translation strategy can be found in [?].

4 Distributed PlusCal Algorithms

Distributed PlusCal extends the syntax of PlusCal in two places, as shown in Figure 4. 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.

The `declarations section` allows the user to declare primitive constructs such as non-ordered channels and FIFO based channels in addition to PlusCal variables. More details on the communication channels are available in Section 4.1.

In the `Process Section` each process can hold multiple sub-processes each with its own body of statements. More details on the sub-processes are available in Section 4.2.

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 4: Syntactic extensions introduced by Distributed PlusCal.

In the sections that follow we will be doing a walk-through on Lamport's distributed mutual exclusion algorithm, the algorithm uses timestamps to order critical section requests.The full implementation is included in appendix **??**.

4.1 Communication Channels

Distributed algorithms composed of multiple processes communicate by exchanging messages through channels. Channels provide communication and coordination for distributed applications. They are classified by the way they handle the addition and removal of messages from their collections.

Channels in Distributed PlusCal are unbounded, meaning that there is no maximum capacity for the number of messages they can hold.

The sections to follow explain how to define and use the channels as well as their TLA$^+$ translations.

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.

In our Lamport Mutex example,

4.2 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 5 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 5 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.

```
----------------------- 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 5: Lamport's mutual-exclusion algorithm.