Extending PlusCal for Modeling Distributed Algorithms

Horatiu Cirstea, Stephan Merz

Extending PlusCal for Modeling Distributed Algorithms*

Horatiu Cirstea and Stephan Merz

University of Lorraine, CNRS, Inria, LORIA, Nancy, France

Abstract. PlusCal is a language for describing algorithms at a high level of abstraction. The PlusCal translator generates a TLA$^+$ specification that can be verified using the TLA$^+$ model checkers or proof assistant. We describe Distributed PlusCal, an extension of PlusCal that is intended to facilitate the description of distributed algorithms. Distributed PlusCal adds two orthogonal concepts to PlusCal: (i) processes can consist of several threads that share process-local variables, and (ii) Distributed PlusCal provides communication channels with associated primitives for sending and receiving messages. The existing PlusCal translator has been extended to support these concepts, and we report on initial experience with the use of Distributed PlusCal.

1 Introduction

Distributed systems and the algorithms that these systems implement are notoriously difficult to design and to verify. This is due to the high number of potential executions that interleave steps of system components (distributed nodes, threads, messaging subsystem) executing independently, leading to bugs that are difficult to reproduce. Formal verification techniques such as model checking or theorem proving help ensure correctness properties of algorithms and of programs. They can be applied at different levels of abstraction. In particular, verifying formal specifications of distributed algorithms at high levels of abstraction allows designers to identify errors that would be very costly to correct during later development stages.

However, languages used for formal modeling and verification are often substantially different from languages used in software development. For example, TLA$^+$ specifications [10] are formulas in a logical language that mixes mathematical set theory and temporal logic, which can be intimidating for system developers. It is therefore desirable to introduce front-end languages that are more familiar to system designers, while still enabling the use of formal verification techniques. In the context of TLA$^+$, PlusCal [11] is a language for describing sequential or concurrent algorithms at a high level of abstraction. PlusCal combines

* This version of the publication has been accepted for publication, after peer review but is not the Version of Record. The Version of Record is available online at `http://dx.doi.org/000000`. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use `https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms`.

the "look and feel" of imperative pseudo-code with the power of mathematical set theory, used for modeling the data structures manipulated by the algorithm. The PlusCal translator generates a TLA$^+$ specification from a PlusCal algorithm, and this specification can be verified using the existing TLA$^+$ model checkers TLC [17] and Apalache [6] or the TLAPS proof assistant [2], thus allowing a system designer to obtain high confidence in the correctness of the algorithm.

However, PlusCal lacks certain constructs that would be useful for modeling distributed algorithms. In particular, it only offers top-level parallel processes, making it difficult to model distributed systems where nodes consist of several threads executing in parallel. In PlusCal, different threads of the same node must be modeled as individual processes, and variables shared by these threads must then be declared as global variables, obscuring the structure of the code. PlusCal also lacks primitives that support inter-process communication through message passing. Instead, such operations have to be described by defining global variables representing the channels and implementing the send/receive operations using low-level TLA$^+$ operators. Constructs similar to those missing in PlusCal can be found in several languages used for programming distributed systems. For example, lightweight threads exist in Ada [1] in the form of *tasks* and in Go [3] as *goroutines*. The latter also offers built-in channel primitives, which are also available in other programming languages including DistAlgo [12], MPI [14], and Erlang [18].

We propose an extension of PlusCal, that we call Distributed PlusCal, which allows users to specify multi-threaded processes and that provides built-in communication channels accessible with classical send/receive primitives. The syntax for threads is simple and intuitive and the translation to TLA$^+$ takes into account the possible interactions with other PlusCal features such as macros, procedures or fairness. Distributed PlusCal supports two types of channels that differ in whether they guarantee that messages are received in the order that they were sent or not. The operations on channels include standard send and receive, as well as a multicast operation that can be used to send a given message to several nodes simultaneously.

In the next section we briefly present TLA$^+$ and PlusCal. Section 3 describes the Distributed PlusCal language and the translation to TLA$^+$. In Section 4 we discuss the specification of two classical algorithms in Distributed PlusCal. We eventually conclude and discuss future work.

2 TLA$^+$ and PlusCal

2.1 The specification language TLA$^+$

TLA$^+$ [10] is a formalism for describing algorithms and systems at a high level of abstraction. It is based on mathematical set theory for representing data structures in terms of sets and functions, and on the Temporal Logic of Actions TLA for representing executions of systems. TLA$^+$ specifications usually have the form

$$Init \wedge \Box[Next]_v \wedge L$$

where *Init* is a predicate describing the possible initial states, *Next* is a predicate that represents the possible state transitions, v is a tuple containing all state variables that appear in the specification, and L is a liveness or fairness property expressed as a formula of temporal logic. Specifications are structured in modules; in particular, standard modules provide frequently used data structures such as integers, sequences or bags (multisets). A module may contain constant and variable declarations, statements of assumptions and theorems, and definitions of operators that are used in assembling the overall specification as well as correctness properties. Specifically, *state formulas* such as the initial condition or state invariants contain constant and variable symbols, where *actions* such as *Next* may also contain primed variable symbols, and *temporal formulas* additionally contain the operators $\Box$ ("always") and $\Diamond$ ("eventually") of linear-time temporal logic. State formulas are evaluated over individual states,[1] action formulas over pairs of states with unprimed variables denoting the value of the variable in the state before the transition and primed variables the value in the state after the transition. Temporal formulas are evaluated over infinite sequences of states.

As a concrete example, consider the specification

$$x \in \mathit{Nat} \land \Box[x' = x + 1]_{\langle x \rangle} \land \mathrm{WF}_{\langle x \rangle}(x' = x + 1)$$

of a counter represented by the variable x. The state predicate $x \in \mathit{Nat}$ requires the initial value of x to be some natural number. The action $[x' = x + 1]_{\langle x \rangle}$ asserts that at every step, x either increments by 1 or remains unchanged,[2] and the temporal formula $\mathrm{WF}_{\langle x \rangle}(x' = x + 1)$ states that the counter is incremented infinitely often. In general, the formula $\mathrm{WF}_v(A)$ representing weak fairnes of action A abbreviates the temporal formula $\Box\big(\Box(\textsc{enabled}\ \langle A \rangle_v) \Rightarrow \Diamond\langle A \rangle_v\big)$ that asserts that whenever an A transition that changes the value of v is forever enabled, then such a transition must occur eventually. Replacing the subformula $\Box(\textsc{enabled}\ \langle A \rangle_v)$ with $\Box\Diamond(\textsc{enabled}\ \langle A \rangle_v)$, one obtains the definition of $\mathrm{SF}_v(A)$ representing strong fairness of action A.

The formal verification of TLA$^+$ specifications is supported by the explicit-state model checker TLC [17], the SMT-based symbolic model checker Apalache [6], and the proof assistant TLAPS [2]. Konnov et al. [7] present an overview of the different verification tools applied to a common case study.

TLA$^+$ is an untyped language that provides a rich language of expressions. The expression $[x \in S \mapsto e]$ denotes the function with domain S such that every element x of S is mapped to the expression e (in which x may occur). This is reminiscent of a λ-expression while also introducing the domain of the function. Function application is written $f[x]$, and the expression $[f\ \texttt{EXCEPT}\ ![v] = e]$ denotes the function that is similar to f, except that argument v is mapped to e. Within e, the symbol @ can be used to denote $f[v]$. A tuple $s = \langle e_1, \ldots, e_n \rangle$ is a function with domain $1\,..\,n$. In particular, $s[i]$ (for $i \in 1\,..\,n$) denotes e_i.

[1] A state assigns a value to each variable.

[2] TLA$^+$ formulas are invariant under finite stuttering, and in particular specifications always allow for stuttering transitions.

```
algorithm <algorithm name>

(* Declaration section *)
variables <variable declarations>

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

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

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

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

Fig. 1. General structure of a PlusCal algorithm.

Similarly, a record such as $[foo \mapsto 42, bar \mapsto \texttt{FALSE}]$ is a function whose domain is the set of strings $\{\texttt{"foo"}, \texttt{"bar"}\}$; $r.foo$ is shorthand for $r[\texttt{"foo"}]$ and record update can be written $[r \texttt{ EXCEPT } !.foo = @ + 1]$.

2.2 The Algorithmic Langage PlusCal

PlusCal [11] was designed as a language for describing algorithms, providing a syntax that resembles imperative pseudo-code. PlusCal's expression language is TLA$^+$, and this makes the language highly expressive, but also means that PlusCal algorithms are in general not executable. PlusCal algorithms are written inside a comment in a TLA$^+$ module, using either a C-like syntax or the P-syntax closer to the Pascal programming language. The PlusCal translator generates a TLA$^+$ specification from the algorithm and inserts it within the module containing the algorithm. Properties of algorithms are expressed as TLA$^+$ formulas and are verified using the standard TLA$^+$ tools. A central objective in designing PlusCal was to ensure a simple translation from PlusCal to TLA$^+$, resulting in human-readable specifications.

The overall structure of a PlusCal algorithm is shown in Figure 1. It is subdivided into several (possibly empty) sections that must appear in the given order. Global variables are declared first, followed by definitions of operators that may be used in the remainder of the algorithm and that may refer to the previ-

ously declared variables. Macros may contain PlusCal statements; their bodies are expanded at translation time, similarly to the expansion of macros in the C language. In contrast, procedures are invoked using the `call` statement of PlusCal. They may declare local variables and may themselves contain procedure calls, including recursive calls. Procedures do not return values but may modify global variables; the `return` statement transfers control back to the calling site.

The final section contains either process declarations as shown in Fig. 1 or a single body of statements for representing a sequential algorithm. Process declarations may introduce a single or a fixed number of instances of the process; the expression on the right-hand side must evaluate to a constant or to a finite set of constants that represent the process identities. Within the body of the process, the identity is denoted by `self`. Process declarations may be annotated by fairness conditions `fair` for weak fairness or `fair+` for strong fairness.

PlusCal statements include `skip` (which does nothing), assignments, conditional statements, while loops, and procedure calls. Processes may synchronize using `await` (or, synonymously, `when`) instructions that block until a predicate becomes true. PlusCal also includes two forms of non-deterministic control structures: `either ... or ...` can be used to introduce a choice between a fixed number of alternatives, whereas the statement `with` $x \in S$ expresses a choice among the values in a set S. In particular, combining `either ... or` and `when` provides guarded commands à la Dijkstra.

An important concern when describing concurrent algorithms is to model the "grain of atomicity", i.e., which statements are assumed to execute without interleaving with statements of other processes. PlusCal uses statement labels to this effect: all statements appearing between two labels are executed atomically. In addition, PlusCal imposes certain rules on the placement of labels. For example, every `while` statement must be labeled, and therefore processes may interleave between every iteration of the loop body. Labels may also take modifiers `+` or `-` that increase or decrease the fairness constraints with respect to that of the enclosing process.

As an example, Fig. 2 shows the PlusCal representation of a semaphore-based mutual exclusion algorithm between N processes where the constant N is declared in the enclosing TLA$^+$ module. In particular, the test and decrement of the semaphore are executed atomically since the instructions are not separated by a label, and strong fairness guarantees starvation freedom.

2.3 Translating PlusCal to TLA$^+$.

Using the example of Fig. 2, we outline how the PlusCal translator generates the TLA$^+$ specification corresponding to an algorithm.

1. Generate TLA$^+$ variable declarations for all variables declared in the algorithm, regardless of their scope. The translator also declares the `pc` variable for tracking control flow, as well as a `stack` variable if the algorithm contains procedures. Define the tuple `vars` of all variables and the set `ProcSet` of all process identifiers.

```
--algorithm SemaphoreMutex {
  variables sem = 1;
  fair+ process (p \in 1..N) {
start:- while (TRUE) {
enter:     await sem > 0;
           sem := sem - 1;
cs:        skip;
exit:      sem := sem + 1;
} }     }
```

Fig. 2. Semaphore mutex example in PlusCal.

```
VARIABLES sem, pc
vars == << sem, pc >>
ProcSet == (1..N)
```

2. Generate `Init`, the predicate that specifies the initial values of all the declared variables. Process-local variables (including `pc`) are represented as functions with domain `ProcSet` in order to distinguish the values corresponding to different instances of processes.

```
Init == /\ sem = 1
        /\ pc = [self \in ProcSet |-> "start"]
```

3. For each label appearing in the algorithm, generate a TLA$^+$ action that represents the effect of the statements following that label. In a multi-process algorithm, the definition takes the parameter `self` that stands for the identifier of the process instance executing the statements.

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

4. For every process, generate a TLA$^+$ action that corresponds to the possible transitions of an instance of that process, as the disjunction of the actions generated for each label appearing in the process. Generate the action `Next` as the disjunction of the actions corresponding to each process, existentially quantified over its instances. In case control flow may reach the end of a process, `Next` contains an extra disjunct `Terminating` that requires all variables to remain unchanged. Finally, generate the overall specification `Spec`, including fairness conditions corresponding to the annotations in PlusCal.

```
p(self) == start(self) \/ enter(self) \/ cs(self) \/ exit(self)
Next == \E self \in 1..N: p(self)
Spec == /\ Init /\ [][Next]_vars
        /\ \A self \in 1..N: SF_vars(pc[self] # "start" /\ p(self))
```

3 Distributed PlusCal

Distributed PlusCal extends PlusCal with two independent features: it adds lightweight (sub-)processes, as well as communication channels with standard operations. Both of these features are available for the C-syntax and the P-syntax; we mainly use the former in this paper. These features are activated using the option `-distpcal` either on the command line or using the PlusCal options line in the enclosing TLA$^+$ module.

3.1 Sub-processes

A Distributed PlusCal process may have several sub-processes that we also call threads. When using the C-syntax, each thread appears within a pair of curly braces, a process with only one pair of braces corresponding to an ordinary PlusCal process. For P-syntax, threads are enclosed by `begin` and `end thread`. An example illustrating the declaration of threads using both syntaxes is given in Fig. 3. Threads cannot declare local variables but can access the local variables of the enclosing process as well as global variables. This corresponds to the intuition that in a distributed system, threads share local memory whereas nodes (represented by Distributed PlusCal processes) communicate via messages.

Translation to TLA$^+$. At every step in the algorithm, one of the threads performs a transition. Consequently, although the syntax additions for threads are quite minor compared to the single-threaded processes of PlusCal there are multiple aspects that have to be taken into account in the translation to TLA$^+$.

The `VARIABLES` declaration as well as `vars` and `ProcSet` are generated as for plain PlusCal. For Distributed PlusCal algorithms, the program counter progresses in each thread and thus, in order to distinguish the values corresponding to each thread, the `pc` variable is represented as a two-dimensional array indexed by the process identity and a thread index. The set `SubProcSet` of thread indexes depends on each process:

$$\texttt{SubProcSet}[p] = \{1, 2, \ldots, nt_p\}, p \in \texttt{ProcSet}$$

with nt_p the number of threads for process p. For the algorithm of Fig. 3, in TLA$^+$ this corresponds to:[3]

```
ProcSet == {3} \union (1..2)
SubProcSet == [self \in ProcSet |->  CASE self = 3 -> 1..2
                                     []   self \in 1..2 -> 1..1 ]
```

The `pc` of each thread is initialized to the label corresponding to its first action:

$$\texttt{pc}[p] = \langle l_1, l_2, \ldots, l_{nt_p} \rangle, \ p \in \texttt{ProcSet}$$

[3] The complete translation is presented in Figure 4 at the end of this section.

```
--algorithm MyAlgo {
  variables
    tab = [ x \in 1..2 |-> 0 ];
  process (pid = 3)
  variables lv = 0;
  {
s1: lv := lv + 1;
    tab[1] := tab[1] + lv;
  }
  {
s2: lv := lv + 1;
    tab[2] := tab[2] + lv;
  }

  process (qid \in 1..2)
  variables t = 0;
  {
rc: await tab[self] > 0;
    t := tab[self];
ut: t := t + 1;
  }
}
```

```
--algorithm MyAlgo
  variables
    tab = [ x \in 1..2 |-> 0 ];
  process pid = 3
  variables lv = 0;
  begin
s1: lv := lv + 1;
    tab[1] := tab[1] + lv;
  end thread
  begin
s2: lv := lv + 1;
    tab[2] := tab[2] + lv;
  end thread

  process qid \in 1..2
  variables t = 0;
  begin
rc: await tab[self] > 0;
    t := tab[self];
ut: t := t + 1;
  end thread
end algorithm
```

Fig. 3. A Distributed PlusCal algorithm in which one process has two threads.

with l_i the label of the first action in the thread i of process p. If procedures are defined, they can be called from any thread and thus, the `stack` variable is also a two-dimensional array initialized to the empty record for each thread:

$$\texttt{stack}[p] = \langle\langle\rangle, \ldots, \langle\rangle\rangle, \;\; p \in \texttt{ProcSet}, \;\; \mathit{size}(\texttt{stack}[p]) = nt_p$$

For our example we have:

```
Init == ...
     /\ pc = [self \in ProcSet |-> CASE self = 3 -> <<"s1","s2">>
                                     [] self \in 1..2 -> <<"rc">>]
```

Then, the translation of each thread corresponds to the disjunction of all its atomic actions, and a process corresponds to the disjunction of all its threads. Thus, for each $p \in \texttt{ProcSet}$:

$$p = \bigvee_{i=1..nt_p} p_i \quad \text{with} \quad p_i = \bigvee_{l \in \mathcal{A}_{p,i}} l$$

where $\mathcal{A}_{p,i}$ denotes the set of (labeled) actions of the thread i in process p.

If p is a process template (specified with `\in`), the translation of p and of its threads takes the parameter `self` that stands, as in PlusCal, for the identifier of the process instance executing the statements.

The first process in our algorithm consists of two threads, translated respectively with the operators `pid1` and `pid2` which correspond to the PlusCal action translations but taking into account the fact that the `pc` variable depends not only on the process id (3 in this case) but on the thread as well.

```
s1 == /\ pc[3][1]  = "s1"
      /\ lv' = lv + 1
      /\ tab' = [tab EXCEPT ![1] = tab[1] + lv']
      /\ pc' = [pc EXCEPT ![3][1] = "Done"]
      /\ UNCHANGED << stack, y, lvp, t >>
pid1 == s1

s2 == /\ pc[3][2]  = "s2"
      /\ ...
pid2 == s2

pid == pid1 \/ pid2
```

For the second process there is only one thread, and since this is a process template (`qid \in 1..2`), the translation uses a parameter `self`:

```
rc(self) == /\ pc[self][1]  = "rc"
            /\ ...
ut(self) == /\ pc[self][1]  = "ut"
            /\ ...
qid1(self) == rc(self) \/ ut(self)

qid(self) == qid1(self)
```

The `Spec` operator and in particular the `Next` operator are generated as for a single-threaded PlusCal specification, with the stuttering action `Terminating` adapted to activate only when all threads have terminated:

```
Terminating == /\ \A self \in ProcSet : \A sub \in SubProcSet[self]:
                     pc[self][sub] = "Done"
               /\ UNCHANGED vars
Next == pid \/ (\E self \in 1..2: qid(self)) \/ Terminating
```

Macros and procedures. Macros can be used as in plain PlusCal and behave as textual substitutions. Procedures are also used similarly to PlusCal but some care has to be taken in the translation since a procedure can be called in any thread. First, as explained above, the `stack` variable is a two-dimensional array in Distributed PlusCal. For the same reasons, the TLA$^+$ variable declarations corresponding to the procedure parameters and local variables are also two-dimensional arrays. Moreover, the operators corresponding to the procedure and to its actions are parameterized not only by a process, as in PlusCal, but also by a thread.

For example, consider the following procedure:

```
procedure foo(ind = 0, y = 0)
variables lvp = 0;
{
s:  lvp := lvp + y;
    tab[ind] := tab[ind] + lvp;
e:  return;
}
```

Three variable declarations corresponding to the parameters and to the local variable of the procedure are added to the `Init` operator:

```
/\ ind = [ self \in ProcSet |-> [ thd \in SubProcSet[self] |-> 0]]
/\ y = [ self \in ProcSet |-> [ thd \in SubProcSet[self] |-> 0]]
/\ lvp = [ self \in ProcSet |-> [ thd \in SubProcSet[self] |-> 0]]
```

and the following operator is generated for the action labeled `s`:

```
s(self, thd) ==
/\ pc[self][thd] = "s"
/\ lvp' = [lvp EXCEPT ![self][thd] = lvp[self][thd] + y[self][thd]]
/\ tab' = [tab EXCEPT ![ind[self][thd]] = tab[ind[self][thd]]
                                          + lvp'[self][thd]]
/\ pc' = [pc EXCEPT ![self][thd] = "e"]
/\ UNCHANGED << stack, ind, y, lv, t >>
```

The operator `e` is handled in a similar way and finally, an operator is generated for the whole procedure:

```
foo(self, thd) == s(self, thd) \/ e(self, thd)
```

One could use this procedure in the specification in Fig. 3 and replace the line 8 by `call foo(1,lv)` in which case the line 3 in the original translation of `s1` becomes

```
/\ /\ stack' = [stack EXCEPT ![3][1] = << [ procedure |->  "foo",
                                            pc        |->  "Done",
                                            lvp       |->  lvp[3][1],
                                            ind       |->  ind[3][1],
                                            y         |->  y[3][1] ] >>
                                         \o stack[3][1]]
   /\ ind' = [ind EXCEPT ![3][1] = 1]
   /\ y' = [y EXCEPT ![3][1] = lv']
```

Fairness. As in PlusCal, fairness conditions can be attached to the algorithm, to process templates or to labels. When fairness is introduced at the top level with `fair algorithm`, we indicate that some statement must eventually be executed if the algorithm can take a step, and this corresponds to the condition `WF_vars(Next)` in the definition of `Spec`.

One can strengthen the condition to ensure that a process will eventually execute if it remains enabled by using the `fair` keyword at the process level. For Distributed PlusCal this corresponds to fairness for each thread of the respective process template. In our example, when writing `fair process (pid = 3)`, the following conditions are added to `Spec`:

```
/\ WF_vars(pid1)
/\ WF_vars(pid2)
```

As in PlusCal, we can strengthen even more the condition with `fair+` and in this case the condition `SF_vars` is used instead of `WF_vars` for all the threads.

The overall fairness requirement attached to a process can be modulated for individual actions and we can exclude a label from the fairness assumption using `-` or assume strong fairness using `+`. If, for the sake of the example, we change the specification of the first process to

```
fair process (pid = 3)
variables lv = 0;
{
s1a:+ lv := lv + 1;
s1b:- tab[1] := tab[1] + lv;
}
{
   // unchanged
}
```

then, the conditions added to `Spec` become

```
/\ WF_vars((pc[3][1] # "s1b") /\ pid1) /\ SF_vars(s1a)
/\ WF_vars(pid2)
```

PlusCal's `-termination` option can be used to generate a TLA$^+$ formula that asserts that all processes and threads will eventually terminate.

```
Spec == /\ Init /\ [][Next]_vars
        /\ WF_vars(pid1)
        /\ WF_vars(pid2)
        /\ \A self \in 1..2 : WF_vars(qid1(self))

Termination == <>(\A self \in ProcSet: \A sub \in SubProcSet[self] :
                     pc[self][sub] = "Done")
```

3.2 Communication Channels

In contrast to threads that communicate using shared variables, processes (or nodes) in distributed systems usually communicate by message passing. In a PlusCal algorithm, channels must be modeled explicitly using global variables, and operations on channels be defined using TLA$^+$ operators or macros.

```
VARIABLES tab, pc, lv, t
vars == << tab, pc, lv, t >>

ProcSet == {3} \cup (1..2)
SubProcSet == [self \in ProcSet |->  CASE self = 3 -> 1..2
                                     []   self \in 1..2 -> 1..1 ]

Init == /\ tab = [ x \in 1..2 |-> 0 ]
        /\ lv = 0
        /\ t = [self \in 1..2 |-> 0]
        /\ pc = [self \in ProcSet |-> CASE self = 3 -> <<"s1","s2">>
                                        [] self \in 1..2 -> <<"rc">>]

s1 == /\ pc[3][1]  = "s1"
      /\ lv' = lv + 1
      /\ tab' = [tab EXCEPT ![1] = tab[1] + lv']
      /\ pc' = [pc EXCEPT ![3][1] = "Done"]
      /\ t' = t
pid1 == s1
s2 == /\ pc[3][2]  = "s2"
      /\ lv' = lv + 1
      /\ tab' = [tab EXCEPT ![2] = tab[2] + lv']
      /\ pc' = [pc EXCEPT ![3][2] = "Done"]
      /\ t' = t
pid2 == s2
pid == pid1 \/ pid2

rc(self) == /\ pc[self][1]  = "rc"
            /\ tab[self] > 0
            /\ t' = [t EXCEPT ![self] = tab[self]]
            /\ pc' = [pc EXCEPT ![self][1] = "ut"]
            /\ UNCHANGED << tab, lv >>
ut(self) == /\ pc[self][1]  = "ut"
            /\ t' = [t EXCEPT ![self] = t[self] + 1]
            /\ pc' = [pc EXCEPT ![self][1] = "Done"]
            /\ UNCHANGED << tab, lv >>
qid1(self) == rc(self) \/ ut(self)
qid(self) == qid1(self)

Terminating == /\ \A self \in ProcSet : \A thread \in SubProcSet[self]:
                     pc[self][thread] = "Done"
               /\ UNCHANGED vars

Next == pid \/ (\E self \in 1..2: qid(self)) \/ Terminating
Spec == Init /\ [][Next]_vars
```

Fig. 4. Translation in TLA$^+$ of the algorithm in Fig. 3.

```
--algorithm ChannelAlgo {
  variables s = 0;
  channels ch[Nodes];
  define {
      Nodes == 1..2
      Id == 3
  }

  process (pid = Id)                    process (qid \in Nodes)
  {                                     variables t = 0;
s1: send(ch[1],Id+1);                   {
  }                                   rcv: receive(ch[self],t);
  {                                   add: s := s + t;
s2: send(ch[2],Id+2);                    }
  }                                  }
```

Fig. 5. A PlusCal algorithm using channels.

Channels in Distributed PlusCal are built-in and can be declared just after the global variables of the algorithm. Channels that guarantee that messages are received in the order in which they are sent are declared using the keyword `fifo` whereas channels that do not offer such guarantees are declared as `channel`.[4] Channels are unbounded, meaning that there is no maximum capacity for the number of messages they can hold.

One can declare an N-dimensional matrix of unordered channels by writing

$$\texttt{channel}\ id[Expr_1][Expr_2]\ldots[Expr_N]$$

with id the name of the channel and $\langle Expr_i\rangle, i = 1..N$, the expressions defining the indexing sets; no set is specified for a simple 0-dimensional channel. Such a declaration gives rise in TLA$^+$ to the declaration of a variable named id and to an initialisation

$$id = [x_1 \in Expr_1, \ldots, x_N \in Expr_N \mapsto EmptyBag];$$

or just $id = EmptyBag$ for a simple channel, where $EmptyBag$ is the operator from module *Bags* representing an empty bag (i.e., multi-set). More precisely, unordered channels are represented in TLA$^+$ using bags and ordered channels using sequences. Thus, if the channel were declared with `fifo` the initialisation would use the empty sequence $\langle\rangle$ instead of the empty bag.

The following initialization predicate is generated for the algorithm in Fig. 5:

```
Init == ...
        /\ ch = [ _n20 \in  Nodes |-> EmptyBag ]
```

[4] Following the PlusCal convention that the keyword `variable` can also be written `variables`, we also allow the plural forms for `fifo` and `channel`.

with `_n20` a freshly generated variable.

The following operations are supported on (unordered or ordered) channels:

- `send`(*chan*, *expr*): sends a message corresponding to the expression *expr* on channel *chan*;
- `receive`(*chan*, *var*): receives a message from channel *chan* and stores it in the variable *var*;
- `multicast`(*chanId*, $[i_1\ op_1\ expr_1, \ldots, i_N\ op_N\ expr_N \mapsto expr$): for an N-dimensional channel ($N > 0$) named *chanId*, sends on all the channels whose indexes match $\langle expr_i \rangle$ with respect to op_i (with op_i either $=$ or $\in$), the message corresponding to the expression *expr*.

Sending a message corresponds to adding the message to the bag (respectively, at the end of the sequence). For our example, this corresponds to the following statement added to the action labeled `s1`:

```
/\ ch' = [ch EXCEPT ![1] = @ (+) SetToBag({Id+1})]
```

where `SetToBag` is the operator for constructing a bag from a set and `(+)` denotes bag union. If `ch` had been declared as `fifo`, its new value would have been `Append(@, Id+1)`.

The receive operation consists in checking the existence of a message in the channel, removing it and assigning it to the target variable. For our example, this corresponds to:

```
/\ \E __c1__ \in DOMAIN ch[self]:
     /\ ch' = [ch EXCEPT ![self] = @ (-) SetToBag({__c1__})]
     /\ t' = [t EXCEPT ![self] = __c1__]
```

For a `fifo` channel, the message at the head of the channel is received using a similar definition in terms of the standard operators `Len`, `Tail` and `Head` on sequences. Note that in both cases `receive` is a blocking operation that is enabled only if a message is present on the channel.

The process `pid` in the algorithm `ChannelAlgo` could be replaced by a single-threaded one consisting of a single multicast operation:

```
multicast(ch, [n \in Nodes |-> Id+n ] );
```

sending a message on `ch[1]` and `ch[2]`. In TLA$^+$ this corresponds to:

```
/\ ch' = [n \in DOMAIN ch |->  IF n \in Nodes
                               THEN ch[n] (+) SetToBag({Id+n})
                               ELSE ch[n]]
```

In this example, the domain of the multicast coincides with the domain of the channel but an expression restricting the domain such as `Nodes \ {1}`, can be used.

Macros handle channels as any other PlusCal object and, in particular, channels can be passed as arguments to macros. The broadcast of a message `m` on all the channels of an N-dimensional channel can be expressed using a macro:

```
macro broadcast(chan,m) {
    multicast(chan,[ag \in DOMAIN chan |-> m]);
}
```

Procedures are also compatible with channels. We should note however that, since the modifications performed in the procedure on its arguments are not persistent, passing as parameter a channel or the target variable for a receive operation is not really useful.

4 Evaluation

PlusCal and Distributed PlusCal are front-ends for writing TLA$^+$ specifications: they provide an input syntax that is more familiar to programmers than writing logical formulas, without giving up on a precise formal semantics. The main objective of Distributed PlusCal is to help the user express distributed algorithms in a more natural way than would be possible using regular PlusCal, while remaining backward compatible for algorithms that do not make use of the extensions.

The tool has been developed as a fork of the TLA$^+$ repository[5] and, in particular, extends the version 1.11 of PlusCal. The source code as well as some examples and a quite extensive test suite are publicly available.[6] A README file provides instructions on compiling the source code and using it (or the precompiled distribution available in the tlatools/dist directory) to translate Distributed PlusCal algorithms. The examples presented in this paper are available in the tlatools/examples-distpcal directory.

4.1 Two distributed algorithms expressed in Distributed PlusCal

We briefly discuss our experience with modeling distributed algorithms using Distributed PlusCal. Figure 6 shows the distributed mutual-exclusion algorithm from [8] written in Distributed PlusCal; the GitHub repository also contains a Distributed PlusCal expression of the Paxos consensus algorithm [9]. Both algorithms consist of N nodes, each of which has a main thread and a concurrently executing helper thread that handles messages received from other nodes. The distributed mutual-exclusion algorithm relies on FIFO communication between nodes, Paxos does not impose any ordering on messages. In both cases, the possibility of declaring multiple threads per node rather than having only top-level processes, makes expressing the algorithms more natural. In particular, the local scope of variables would be lost if the language did not provide threads. We believe that this observation explains why these algorithms are available from the collection of TLA$^+$ examples[7] only in the form of TLA$^+$ specifications rather than PlusCal algorithms.

[5] https://github.com/tlaplus/tlaplus
[6] https://github.com/DistributedPlusCal/DistributedPlusCal
[7] https://github.com/tlaplus/Examples/tree/master/specifications

```
------------------------ 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
     Request(c) == [type |-> "request", clock |-> c]
     Release(c) == [type |-> "release", clock |-> c]
     Acknowledge(c) == [type |-> "ack", clock |-> c]
   }
   process(node \in Nodes)
     variables clock = 0, req = [n \in Nodes |-> 0],
               ack = {}, sndr = self, msg = Request(0);
   { \* thread executing the main algorithm
ncs: while (TRUE) {
       skip;  \* non-critical section
try:   clock := clock + 1; req[self] := clock; ack := {self};
       multicast(network, [m = self, n \in Nodes |-> Request(clock)]);
enter: await (ack = Nodes /\ \A n \in Nodes \ {self} :
                                \/ req[n] = 0
                                \/ req[self] < req[n]
                                \/ req[self] = req[n] /\ self < n);
cs:    skip;  \* critical section
exit:  clock := clock + 1;
       multicast(network, [m = 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; };
         msg := Request(0); sndr := self;
     }  \* end while
   } \* end message handling thread
}  **)
```

Fig. 6. Lamport's distributed mutual-exclusion algorithm.

The main safety properties of these algorithms are mutual exclusion (for LamportMutex) and agreement (for Paxos), expressed respectively as the TLA$^+$ formulas

$$\begin{aligned} \mathit{Mutex} &\triangleq \forall m, n \in \mathit{Nodes} : m \neq n \Rightarrow \neg(\mathit{pc}[m] = \texttt{"cs"} \land \mathit{pc}[n] = \texttt{"cs"}) \\ \mathit{Agreement} &\triangleq \forall m, n \in \mathit{Nodes} : \ \mathit{chosen}[m] \neq \mathit{None} \land \mathit{chosen}[n] \neq \mathit{None} \\ &\qquad\qquad \Rightarrow \ \mathit{chosen}[m] = \mathit{chosen}[n] \end{aligned}$$

The TLC model checker is able to verify these properties for the specifications generated from our Distributed PlusCal algorithms.[8] For example, when `fifos` is replaced by `channels` in algorithm LamportMutex, TLC generates a counter-example that illustrates why FIFO channels are necessary for this algorithm. The size of the state space for LamportMutex matches that of the existing TLA$^+$ specification of this algorithm. For Paxos, the state space generated by the Distributed PlusCal version is larger than that of the existing TLA$^+$ specification, which makes a few shortcuts whereas we emphasized readability.

As for existing PlusCal algorithms, the interactive proof assistant TLAPS can also be used for reasoning about Distributed PlusCal algorithms. The use of the symbolic model checker Apalache requires type annotations for variables and operator definitions; just as for regular PlusCal these have to be added manually by the user. A future version of the translator could propagate type annotations at the Distributed PlusCal level to the generated TLA$^+$ specification.

4.2 Related Work

Modular PlusCal is a variant of PlusCal based on archetypes (similar to PlusCal's processes), mapping macros (a more disciplined form of macros avoiding side-effects), and distinguishes parameter passing by value or by reference. The PGo compiler [4] can generate either regular PlusCal algorithms or Go programs from Modular PlusCal algorithms. Just like PlusCal processes, archetypes in Modular PlusCal cannot contain multiple threads executing in parallel. Unlike Distributed PlusCal, Modular PlusCal is not backward compatible with ordinary PlusCal.

The DistAlgo language [13] is a domain-specific language, implemented in Python, for writing distributed programs. It provides primitives for interprocess communication through messages, including asynchronous message reception, and contains declarative constructs such as queries over the histories of sent and received messages. However, its focus is on execution rather than verification.

IronFleet [5] introduced a methodology for describing distributed algorithms as state machines, similar to TLA$^+$, in a form that was amenable to automated program verification with Dafny. Languages such as EventML [15] or Verdi [16] embed the semantics of distributed algorithms and systems in interactive proof assistants and therefore require familiarity with these frameworks for modeling and verification.

[8] As is standard in finite-state model checking, finite bounds have to be introduced for variables that could grow indefinitely such as clocks or ballots.

5 Conclusion

We presented an extension of the PlusCal algorithm language for describing distributed algorithms. Rather than introducing many new features that could break the design objectives of PlusCal being a lightweight front-end to writing TLA$^+$ specifications, our objective was to add few, orthogonal concepts while both remaining compatible with the existing language and keeping simple the generation of human-readable TLA$^+$ specifications. The added concepts are inspired from those found in distributed programming languages. Compared to the original PlusCal language, Distributed PlusCal allows processes to consist of multiple threads that communicate via process-local variables, and it introduces communication channels that can be declared as preserving FIFO order or not. Whereas PlusCal supports writing concurrent programs by providing a process abstraction, elevating threads to top-level processes requires all variables shared between threads to be declared as global variables, breaking locality. Moreover, PlusCal does not provide communication channels with corresponding operations. Although these can be represented using global variables and macros or operator definitions, this requires that users write low-level TLA$^+$ instead of expressing their algorithm in PlusCal.

We have illustrated Distributed PlusCal using two well-known algorithms and our preliminary findings indicate that the extensions provided by Distributed PlusCal help us express distributed algorithms in a natural way. Moreover, any overhead incurred in verification with respect to a specification written in TLA$^+$ is not different from that of ordinary PlusCal. However, more experience, including by users of Distributed PlusCal different from its authors, will be necessary for a more thorough evaluation of the language.

As for PlusCal, the semantics of the language is given by the translation towards TLA$^+$ and, in the future, we intend to give a formal description in order to compare it with semantics of distributed programming languages and eventually aim for a translation of some Distributed PlusCal specifications into distributed programs.

Currently, Distributed PlusCal supports a fixed number of distinct threads per process. It may be interesting to add replicated instances of threads, just as processes can be replicated in PlusCal. For example, such a feature could be used for modeling processors having multiple CPU and GPU cores. Distributed PlusCal does not currently allow threads to be nested inside another thread. Doing so would require maintaining a tree of control locations and would thus result in a more complicated translation to TLA$^+$.

Beyond providing just two types of FIFO and unordered channels, one could imagine providing a collection of user-extensible libraries for representing channels supporting different abstractions of causality in distributed systems.

Eventually, we aim at integrating Distributed PlusCal into the existing PlusCal translator.

Acknowledgments. We would like to thank several Master students, and in particular Heba Alkayed, who contributed to earlier versions of Distributed PlusCal.

References

1. John Barnes. *Programming in Ada 2012*. Cambridge University Press, USA, 2014.
2. Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. TLA$^+$ proofs. In Dimitra Giannakopoulou and Dominique Méry, editors, *18th Intl. Symp. Formal Methods (FM 2012)*, volume 7436 of *LNCS*, pages 147–154, Paris, France, 2012. Springer.
3. Alan A.A. Donovan and Brian W. Kernighan. *The Go Programming Language*. Addison-Wesley Professional, 1st edition, 2015.
4. Finn Hackett, Shayan Hosseini, Renato Costa, Matthew Do, and Ivan Beschastnikh. Compiling distributed system models with PGo. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors, *Proc. 28th ACM Intl. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 159–175, Vancouver, Canada, 2023. ACM.
5. Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In Ethan L. Miller and Steven Hand, editors, *Proc. 25th Symp. Operating Systems Principles (SOSP)*, pages 1–17, Monterey, CA, U.S.A., 2015. ACM.
6. Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. TLA$^+$ model checking made symbolic. *Proc. ACM Program. Lang.*, 3(OOPSLA):123:1–123:30, 2019.
7. Igor Konnov, Markus Kuppe, and Stephan Merz. Specification and verification with the TLA$^+$ Trifecta: TLC, Apalache, and TLAPS. In Tiziana Margaria and Bernhard Steffen, editors, *11th Intl. Symp. Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2022)*, volume 13701 of *Lecture Notes in Computer Science*, pages 88–105, Rhodes, Greece, 2022. Springer.
8. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
9. Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
10. Leslie Lamport. *Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, USA, 2002.
11. 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.
12. Yanhong A. Liu, Scott D. Stoller, and Bo Lin. From clarity to efficiency for distributed algorithms. *ACM Transactions on Programming Languages and Systems*, 39(3):1–41, May 2017.
13. Yanhong A. Liu, Scott D. Stoller, and Bo Lin. From clarity to efficiency for distributed algorithms. *ACM Trans. Program. Lang. Syst.*, 39(3):12:1–12:41, 2017.
14. Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.
15. Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. Eventml: Specification, verification, and implementation of crash-tolerant state machine replication systems. *Sci. Comput. Program.*, 148:26–48, 2017.
16. James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In David Grove and Stephen M. Blackburn, editors, *Proc. 36th ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, pages 357–368, Portand, OR, U.S.A., 2015. ACM.

17. Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA$^+$ specifications. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, pages 54–66, Bad Herrenalb, Germany, 1999. Springer.
18. Peter Zeller, Annette Bieniusa, and Carla Ferreira. Teaching practical realistic verification of distributed algorithms in Erlang with TLA+. In Annette Bieniusa and Viktória Fördós, editors, *Erlang Workshop*, pages 14–23. ACM, 2020.