Formal Verification of Distributed Algorithms using Distributed PlusCal

Stephan Merz, Horatui Cirstea, Heba Al-kayed
Loria, Inria

***Abstract*—**

***Index Terms*—**

I. 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.

Distributed systems benefit greatly from testing failure conditions like deadlocks or race conditions at early stages of development at design level.

Formal verification methods have been employed successfully to model the system and its properties and then verify its correctness. One of the verification methods is called model checking [**?**], it 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.

II. Model checking using TLA$^+$

A. 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.

```
------------ 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]
=============================================
```

Fig. 1. A memory specification in TLA+.

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 [**?**] [**?**].

B. PlusCal Algorithmic Langage

TLA+ is a specification formalism not a programming language. It relies on mathematical logic and formulas for structuring specifications, which may discourage programmers from using it.

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.

```
(** 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>

**)
```

Fig. 2. General structure of a PlusCal algorithm

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.

```
(*
--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 ;
          }
  }
}
*)
```

Fig. 3. Semaphore mutex example in PlusCal

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.

C. 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 [**?**].

III. Distributed PlusCal

DISTRIBUTED PLUSCAL is a language used to describe distributed algorithms, it extends PlusCal. Our motivations for creating Disrtibuted PlusCal are quite similar to the motivations that created PlusCal, we want a syntax that would spare the user from having to model primitives that usually accompany distributed algorithms.

Since we extended the existing PlusCal translator responsible for compiling PlusCal into TLA$^+$ we inherited the same syntax and semantics and added our own constructs.

A. Structure of an algorithm

The general structure and organization of a Distributed PlusCal algorithm is shown in Figure 4.

The `PlusCal options section` holds options to be passed to the translator, for example adding $-label$ to the PlusCal options turns on the automatic labeling of the algorithm by the translator. Since we extended the PlusCal translator we notify the translator to parse a Distributed PlusCal algorithm by passing the $-distpcal$ option.

Figure 4 resembles Figure 2, however the variable declarations in the `declarations section` allows the user to

```
(* PlusCal options section *)
(* PlusCal options (-distpcal) *)

(* algorithm <algorithm name>

(* Declaration section *)
variables <variable declarations>
channels <channel declarations>
fifos <fifo 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>
  <sub-processes>

*)
```

Fig. 4. General structure of a Distributed PlusCal algorithm

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 III-B.

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 IV.

In the sections that follow we will be doing a walk-through on the two phase commit protocol that is used in distributed transactions that consist of multiple operations, performed at multiple sites. The goal of the protocol is to reach consensus between the different elements that carry out the transaction together such that if an element decides to abort all the other elements abort as well, and the transaction is rolled back. The full implementation is included in appendix **??**.

B. 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.

For some algorithms, message ordering is critical, for example if a message is sent to subscribe in a service and another is send to unsubscribe, it is crucial that the first message is received and handled before the second. However, some algorithms are not dependent on ordering. Distributed PlusCal offers two types of channels, unordered channels as well as FIFO based channels.

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.

1) Unordered Channels: Unordered channels are declared and used as communication gateways in algorithms that are not dependent on the order in which the messages are handled.

The general structure for channel declaration is shown below. An unordered channel is declared with the keyword **channel** or **channels**.

$$\textbf{channel } \langle id \rangle [\langle dimension1, ..., dimensionN \rangle];$$

In PlusCal a channel would be modeled by a plain variable with a declaration of the form

$$\textbf{variable } \langle id \rangle [\langle d1 \in dimension1, ...dN \in dimensionN | - > \{\} \rangle];$$

The $dimensions$ are TLA$^+$ sets that are defined in the TLA$^+$ file, they represent the keys of the channel. Defining a multidimensional channel corresponds to defining a set of channels.

The corresponding TLA$^+$ translation of the unordered channel declared with N dimensions is shown below.

$$\langle id \rangle = [\langle d1 \in dimension1, ...dN \in dimensionN | - > \{\} \rangle];$$

In our two phase commit example, we have two unordered channels $coord$ and agt, The channel agt has a dimension composed of the values of the set $Agent$ which is a constant in our algorithm, by defining a channel with an $Agent$ dimension we are defining a channel for each agent. The Distributed PlusCal declarations and the TLA$^+$ translation are in Figure 5.

```
channels coord, agt[Agent]| ⇒
 Init == (* Global variables *)
          /\ coord = {}
           /\ agt = [a0 \in Agent |-> {}]
```

Fig. 5. Unordered channel declaration - TLA$^+$ translation example

Unordered channel are represented by TLA$^+$ sets. TLA$^+$ sets are the basic construct for representing data. Sets are not concerned with the order of elements within their collection.

Set operators include $\in$ which is written as `\in` in TLA$^+$ checks whether an element is a part of a set, $\cup$ is written as `\cup` adds an element to the set.

Distributed PlusCal extends PlusCals allowed statements within a body to include channel operations.

The list below shows the syntax of the channel operators and their TLA$^+$ translations, to simplify the examples assume we are dealing with a one-dimensional channel (indexed by set $IdSet$) with the identifier $chan$.

◇ **send** The `send` operator carries two parameters, the channel `chan` and the message `msg`. The purpose of this operator is to add the message `msg` to the given channel. Since the unordered channel is represented using a TLA+ set, addition to a set is performed using the set operator ∪ which is written in TLA+ as `\cup`.

```
send(chan[e], msg) ≜
     chan' = [chan EXCEPT ![e] = chan[e]
         \cup {msg}]
```

◇ **broadcast**
The `broadcast` operator takes two parameters, the channel `chan` and the message `msg`. The purpose of this operator is to send the `msg` to all the elements designated by `chan`.

```
broadcast(chan, msg) ≜
         chan' = [el \in IdSet |->
                     chan[el] \cup {msg}]
```

◇ **multicast**
The `multicast` operator takes two parameters, the channel `chan` and a TLA+ expression. The second parameter specifies the intended recipients of the message and the message to be sent. The purpose of this operator is to send a `msg` to some channels that are designated by the `chan`. The receiving channels are the ones that satisfy the set `bub` specified by the programmer in the second argument.

```
multicast(chan, [a \in sub |-> msg]) ≜

chan' = [a \in DOMAIN chan |->
     IF a \in sub
     THEN chan[a] \cup {msg}
     ELSE chan[a]]
```

◇ **receive** The `receive` operator has two parameters, the channel `chan` and a TLA+ variable `var`. The purpose of this operator is to remove an element from the channel and assign it to the variable sent in the second argument.

```
receive(chan[element], var)| ≜
    \E e \in chan[element]:
      /\ var' = e
      /\ chan' = [chan EXCEPT ![element] =
        chan[element] \ {e}]
```

◇ **clear** The *clear* operator empties the channel *chan*.
`clear(chan)` ≜ `chan'= [el \in IdSet -> {}]`

The Two phase commit example in appendix **??** includes the functions `send, broadcast, receive` and `clear` with their TLA+ translations. The listings below focus on the `send` and `receive` operations.

In the twp phase commit example, an agent process decides whether it wants to "accept" or "refuse" a transaction and the decision is held by the variable `st`, then the process informs the coordinator of its decision through the channel `coord`.

```
send(coord, [type |-> st, agent |-> self]) ⇒
  coord' = (coord \cup {[type |-> st,
     age\textbf{}nt |-> self]})
```

The coordinator receives the decisions of the agents though the `coord` channel, the translation consists of first ensuring that the channel is not empty, then removing an element from the channel and placing it in the variable of the second argument.

```
receive(coord, msg)! ⇒
    \E c \in coord:
    /\ coord' = coord \ {c}
    /\ msg' = c
```

2) FIFO channels: FIFO channels are declared and used for algorithms whose correctness depends on handling the messages in the order of their creation.

The general structure for channel declaration is shown below. A FIFO channel is declared with the keyword **fifo** or **fifos**.

$$\textbf{fifo}\ \langle id\rangle[\langle dimension1, ..., dimensionN\rangle];$$

In PlusCal a channel would be modeled by a plain variable with a declaration of the form

$$\textbf{variable}\ \langle id\rangle[\langle d1 \in dimension1, ...dN \in dimensionN|- > \langle\rangle\rangle];$$

The TLA+ translation a FIFO channel is as follows

$$\langle id\rangle = [\langle d1 \in dimension1, ...dN \in dimensionN|- > \langle\rangle\rangle];$$

FIFO channels are represented as TLA+ sequences. TLA+ sequences are ordered collections, analogous to lists. Adding to a sequence corresponds to appending at the end of the sequence with the `Append` operator, and removing the first element of a sequence can be accomplished with the `Tail` operation. We take advantage of these operators to maintain the order in the FIFO channels.

All sequences are TLA+ functions, where $\langle\rangle$ represents an empty sequence.

Appendix **??** includes an example for modeling Lamports Mutual Exclusion algorithm, an algorithm that uses time stamps to order critical section requests and to resolve any conflicts between them. The example introduces a two dimensional FIFO channel `network`, the dimensions are sets introduced on the algorithm.

The Distributed PlusCal declaration and the TLA+ translation of the `network` FIFO channel are in Figure 6.

```
fifo network[Nodes, Nodes] ⇒

network = [n0 \in Nodes, n1 \in Nodes |-> ⟨⟩]
```

Fig. 6. FIFO channel declaration - TLA+ translation

Distributed PlusCal added to the statements allowed within a body functions that are applied on channels. The list below shows the syntax of the channel operators and their TLA+ translations, to simplify the examples assume we are dealing

with a one dimensional channel (indexed by `IdSet`) with the identifier `chan`.

◇ **send** The `send` operator carries two parameters, the channel `chan` and the message `msg`. Since the FIFO channel is represented using a TLA+ sequence, addition to a sequence is performed using the operator `Append` which adds to the tail of the sequence.

```
send(chan[e], msg) ≜
    chan' = [chan EXCEPT ![e] =
              Append(@, msg)]
```

◇ **broadcast** The `broadcast` operator takes two parameters, the channel `chan` and the message `msg`.

```
broadcast(chan, msg) ≜
    chan' = [el \in IdSet ->
        Append(chan[el], msg)]
```

◇ **multicast**
The `multicast` operator takes two parameters, the channel `chan` and a TLA+ expression. The second parameter specifies the intended recipients of the message and the message to be sent.

```
multicast(chan, [a \in sub |-> msg]) ≜
    chan'=[a \in DOMAIN chan |->
       IF a \in sub
       THEN Append(chan[a], msg)
       ELSE chan[a]]
```

◇ **receive**
The `receive` operator has two parameters, the channel `chan` and a TLA+ variable `var`. The purpose of this operator is to remove the first element from the channel using the `Head` and `Tail` operators and assign it to the variable sent in the second argument.

```
receive(chan[element],var) ≜
    /\ Len(chan[element]) > 0
    /\ var' = [Head(chan[element])]
    /\ chan' = [chan EXCEPT ![element] =
        Tail(@) ]
```

◇ **clear** The `clear` operator empties the channel `chan`.

```
clear(chan) ≜ chan'= [el \in IdSet -> ⟨⟩]
```

The Lamport Mutex example in Appendix **??** includes the operators `send, multicast, receive` and `clear` with their TLA+ translations. The listings below focus on the `receive` operation as an example.

In the example, `Nodes` represent a set with all the nodes in our algorithm. The `network[Nodes,Nodes]` represents a set of channels, where a channel is referenced by a `[sender, receiver]` where `sender, receiver` ∪ `Nodes`. When a process receives from the channel `network[n,self]` where `n \in Nodes`, the channel is receiving messages that are sent for it.

The TLA+ translation uses the operators `Head` and `Tail` in order to read the first element in the channel and remove it as well.

```
receive(network[n,self], msg) ⇒
    /\ Len(network[n,self]) > 0
    /\ msg' = [msg EXCEPT ![self] =
        Head(network[n,self])]
    /\ network' = [network EXCEPT ![n,self] =
         Tail(@) ]
```

```
  fair process (a \in Agent)
  variable aState = "unknown"; { \*
      sub-process

a1: if (aState = "unknown") {
        with(st \in {"accept", "refuse"}) {
          aState := st;
          send(coord, [type |-> st, agent |->
              self]);
        };
    };
    a2: await(aState \in {"commit", "abort"})

  } { \* sub-process

    a3:await (aState # "unknown");
       receive(agt[self], aState);

    a4:clear(agt);
  }
```

Fig. 7. Distributed PlusCal Sub-Processes

IV. Sub-Processes

Distributed PlusCal gives the user the opportunity to define multiple sub-processes per process. Each sub-process consists of labeled statements. Essentially this enables the process to execute multiple tasks in parallel.

The body of a sub-process maintains the same syntax as the body of a PlusCal process. All the sub-processes share the same variables declared for the process, this makes communication between them possible if needed.

The Figure 7 shows a sub-process defined for the two phase commit algorithm, the example is written in c-syntax and in this case the sub-processes are surrounded by curly braces. The entire algorithm is in Appendix **??**.

The process shown in 7 represents an Agent process that communicates with a coordinator in order to apply a decision. The process consists of two sub-processes, where the first sub-process contains the atomic labels «a1, a2» and the second sub-process contains «a3, a4».

its important to note that the sub-processes do not allow variable declarations but they use variables declared for the entire process.

A. TLA+ Translation

The translation of a Distributed PlusCal process into TLA+ meant having to introduce structures and sets that refer to the sub-processes as well as the processes.

A set SubProcSet is added to hold all the sub-process identifiers, with respect to which process they belonged to.

$$SubProcSet \triangleq$$

$$[P \in ProcSet \mapsto \text{IF } P \in IdSet \text{ THEN } 1..n \text{ ELSE } 1..m]$$

The value of the **pc** variable in PlusCal is a single string equal to the label of the next statement to be executed with respect to a process. In Distributed PlusCal we extended the definition to indicate which sub-process is involved as well.

Since in TLA+ a function with domain $1..n$ for some $n \in Nat$ is a sequence, the pc variable in Distributed PlusCal is initialized as

$$pc = [self \in ProcSet \mapsto [self \in IdSet \mapsto \langle \text{"}lbl\text{"}, \ldots \rangle]]$$

where the labels that appear in the sequence are the entry point actions for each sub-process.

The **pc** is referenced within the produced TLA+ translation as

$$pc[ProcessId][SubProcessIndex]$$

The entire TLA+ translation of the Two Phase Commit example can be found in appendix **??**, now we will be focusing on the elements that were added or modified in the translation process from PlusCal to TLA+. The following steps demonstrate these differences in the translation of the process in figure 7.

1) Generate the set SubProcSet that identifies the sub-processes based on the process type. In our example we have two process types, one for Agents and the other for Coordinators, both process types have two sub-processes.

```
SubProcSet == [n \in ProcSet |-> IF n \in
    Agent THEN 1..2
                              ELSE (**Coord**)
                                  1..2]
```

2) Generate the pc variable to be a point of control for the processes and sub-processes. In our example, for processes of type Agent we have two sub-processes. The first sub-process begins execution at action "a1", and the second one begins at action "a3".

```
Init == (* Global variables *)
        /\ coord = {}
        /\ agt = [a0 \in Agent |-> {}]
        (* Process a *)
        /\ aState = [self \in Agent |->
            "unknown"]
        (* Process c *)
        /\ cState = "unknown"
        /\ commits = {}
        /\ msg = {}
        /\ pc = [self \in ProcSet |-> CASE
            self \in Agent -> <<"a1","a3">>
             [] self = Coord -> <<"c1","c2">>]
```

3) Translate the process labels into TLA+ actions. The pc variable is referenced with respect to the appropriate sub-process identifier. In our example, when the pc variable is accessed in the actions $\langle$"$a1$","$a2$"$\rangle$ which belong to the first sub-process the pc variable is referenced by the sub-process identifier [1] as shown in Figure 3.

```
\* Process 1 Sub-Process 1 : an action with
    the statements in a1 label
a1(self) ==
    /\ pc[self] [1] = "a1"
    /\ IF aState[self] = "unknown" THEN
          /\ \E st \in {"accept", "refuse"}:
               /\ aState' = [aState EXCEPT
                   ![self] = st]
               /\ coord' = (coord \cup {[type
                   |-> st, agent |-> self]})
      ELSE
          /\ TRUE
          /\ UNCHANGED << coord, aState >>
    /\ pc' = [pc EXCEPT ![self] = [@  EXCEPT
        ![1] = "a2"]]
    /\ UNCHANGED << agt, cState, commits, msg
        >>
```

V. Conclusion

In summary, this paper addresses Distributed PlusCal, a language that provides algorithm designers with an interface in which distributed algorithms and their properties can be expressed naturally. While the PlusCal language may enforce some limitations that make it difficult to express distributed algorithms in a natural way, Distributed PlusCal introduced constructs that can overcome these limitation.

In distributed algorithms several processes or threads may coexist and asynchronously communicate while being a part of a distributed node. Distributed PlusCal firstly introduced Sub-processes IV to allow the declaration of these threads, and secondly we provided the user with communication channel constructs III-B to assist in the communication between those processes.

Extending the PlusCal translator allowed our translator

to be backward compatible. The Distributed PlusCal translator can translate PlusCal models as well as Distributed PlusCal models.