UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Formal Verification of Distributed Algorithms using Distributed-PlusCal

Report

soutenu le 23 septembre 2016

A thesis submitted to obtain a

Master de l'Université de Lorraine
(mention informatique)

by

Heba Al-kayed

Composition du jury

Président :	Le président
Rapporteurs :	Le rapporteur 1 Le rapporteur 2 Le rapporteur 3
Examinateurs :	L'examinateur 1 L'examinateur 2

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thesul.

Remerciements

Les remerciements.

Je dédie cette thèse
à ma machine.
Oui, à Pandore,
qui fut la première de toutes.

Summary

1

Introduction

This is a very nice thesis about TLA [?].

Motivations why this extension

Outline

2

Background info

a brief overview of ModelChecking, TLA and Pluscal possibly with an example to show why it's used or its advantages. maybe mention real life applications like amazon's AWS.

2.1 TLA+

[?].

2.2 PlusCal algorithm language

3

Related work

position of our work compared with other work

3.1 PGO

3.1.1 Modular PlusCal

4

Distributed PlusCal

Distributed PlusCal is an extension of PlusCal, they are algorithm languages meant for writing algorithms not programs. they are both translated to TLA+.

Since distributed algorithms are based on continuous interactions among components, they benefit greatly from testing failure conditions like deadlocks or race conditions at early stages of development at design level, and TLA+ provides a flexibility and an expressiveness that makes it able to specify and verify those algorithms. One of the popular examples of incorporating TLA+ to verify distributed algorithms is it's usage at Amazon Web Services [?].

Initially PlusCal was introduced to accompany TLA+ since TLA+ syntax may seem unfamiliar to most users, this enabled users to express their algorithms using a well known syntax and then use the translator provided with the TLA+ Toolbox to generate the corresponding TLA+ code that is needed to check for the algorithm's correctness through the TLC model checker.
PlusCal offers two syntax styles one called c-syntax resembling c programming style and p-syntax(p for prolix), A language manual is available on the PlusCal Web site for both syntaxes.

As an example on the translation process lets look at labels in PlusCal, labels are used to indicate atomicity between statements where an atomic step starts at a label and ends at the next label, these atomic steps are translated into actions in TLA+ as shown in the example below where A and B are labels within the same process.

Listing 4.1 – PlusCal

```
(* --algorithm foo{
variables x = 0;
process (c \in 1..3)
    {
  A:x := x + 1;
  B:x := 0;
  }
} *)
```

Listing 4.2 – TLA+ Translation

```

A(self) == /\ pc[self] = "A"
           /\ x' = x + 1
           /\ pc' = [pc EXCEPT
               ![self] = "B"]

 B(self) == /\ pc[self] = "B"
           /\ x' = 0
           /\ pc' = [pc EXCEPT
               ![self] = "Done"]
```

The translation also introduces a variable *pc* to represent the control state, it's main job is to handle the navigation from one action to the other, as seen in the above example the first line of any action is setting the *pc* variable to that action, then before the action finishes running it's statements it must indicate what action (for the same process) needs to be executed next, we notice that action A specifies that the next action to be executed is action B, and B specifies the next action is "Done" indicating that the entire process has actually finished its execution.

Distributed algorithms are concerned with fault tolerance and consistency between multiple elements, PlusCal helps to verify liveness properties such as weak fairness by using the keyword *fair* before a process as shown in the example below.

Listing 4.3 – PlusCal

```
(* --algorithm foo{
variables x = 0;
    fair process (c \in 1..3)
    {
     A:
       x := x + 1;
     B:
      x := 0;
    }
} *)
```

Listing 4.4 – TLA+ Translation

```
c(self) == A(self) \/ B(self)

Spec == /\ Init /\ [][Next]_vars
        /\ \A self \in 1..3 :
            WF_vars(c(self))
```

the translation consists of adding a disconjunction of the actions corresponding to the labels of the process and then adding the condition *WF_vars* to the *Spec* to verify if the newly added action remains continuously enabled.

The overall translation strategy is explained here [?], and since we extended the same translator responsible for parsing PlusCal into TLA+ we inherited the same semantics and grammar and added our own.

Our motivations for creating Disrtibuted PlusCal are quite similar to the motivations that created PlusCal, we wanted a syntax that would spare the user from having to model primitives that usually accompany distributed algorithms such as sub-processes and communication channels.

In the sections to follow we will be explaining what we implemented,why it is needed and how it is translated into TLA+, for this we will be doing a walkthrough on the two phase commit example.

4.1 Channels

4.1.1 Unordered channels

example with it's translation

4.1.2 FIFO channels

example with it's translation

4.1.3 Supported channel functions

expected syntax and limitations

4.2 Sub-Processes

A PlusCal process has a block that holds the body of the process, Distributed PlusCal gives each process the opportunity to define more than one block where each block has a body, This enables the process to be executing multiple tasks in parallel, for example a sub-process can be used to send/receive data to/from other processes asynchronously while the another sub-process is executing it's statement not concerned by such commands.

// more on the benefits of this for distributed algorithms especially

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 making communication between them possible if needed.

The example below shows the sub-processes defined for the two phase commit algorithm, the example is written in c-syntax and the sub-processes are surrounded by curly braces.

Listing 4.5 – Distributed PlusCal Sub-Processes

```
EXTENDS Sequences, Naturals

CONSTANTS Coord, Agent

(* PlusCal options (-distpcal, -label) *)

(***
--algorithm TPC {
  \* message channels
  channels coord,agt[Agent];

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
    await (aState # "unknown");

    \* asynchronous message reception
    receive(agt[self], aState);
    clear(agt);
  }

  \* second process
  fair node (c = Coord)
  variables cState = "unknown",
            commits = {}, msg = {};
             \* agents that agree to commit
  { \*sub process
    c1: await(cState \in {"commit", "abort"});

    broadcast(agt, [ag \in Agent|-> cState]);
  } { \* sub process

        while (cState \notin {"abort", "commit"}) {
            receive(coord, msg);
            if (msg.type = "refuse") {
                cState := "abort";
            }
            else if (msg.type = "accept") {
                commits := commits \cup {msg.agent};
                 if (commits = Agent) {
                     cState := "commit";
                 }
              }
        }
  }
 }
}
```

if we zoom in on the first process for example we'll see that it consists of two sub-processes, the first one contains the labels a1 and a2 and the second one's body consists of the statements between the second pair of curly braces.

//The send, receive and clear and broadcast would've been explained already //in the previous section.

we can notice that the variable *aState* is shared between the sub-processes in fact it is used for communication between them, label a2 holds an *await* statement that is waiting for aState to have the value of either *"commit"* or *"abort"*, and the variable is being set to one of these values by the second sub-process by the receive function.

it's important to note that the sub-processes do not allow variable declarations they only use variables declared for the entire process.

Defining sub-processes had some effects on the translation to TLA+, because when you declare sub-processes you are actually partitioning the process, so now whenever we are referring to a process we need to know not only which process it is but also which sub-process are we referring to.

Listing 4.6 – TLA+ translation for Sub-Processes

```

\* BEGIN TRANSLATION
VARIABLES coord, agt, pc, aState, cState, commits, msg

vars == << coord, agt, pc, aState, cState, commits, msg >>

ProcSet == (Agent) \cup {Coord}

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

Init == (* Global variables *)
        /\ coord = {}
        /\ agt = [a0 \in Agent |-> {}]
        (* Node a *)
        /\ aState = [self \in Agent |-> "unknown"]
        (* Node c *)
        /\ cState = "unknown"
        /\ commits = {}
        /\ msg = {}
        /\ pc = [self \in ProcSet |-> CASE self \in Agent -> <<"a1","Lbl_1">>
                                          [] self = Coord -> <<"c1","Lbl_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 >>

\* Process 1 Sub-Process 1 : an action with the statements in a2 label
```

```
a2(self) == /\ pc[self] [1] = "a2"
            /\ (aState[self] \in {"commit", "abort"})
            /\ pc' = [pc EXCEPT ![self] = [@  EXCEPT ![1] = "Done"]]
            /\ UNCHANGED << coord, agt, aState, cState, commits, msg >>

\* Process 1 Sub-Process 2 :an action for the statements of await and receive
Lbl_1(self) == /\ pc[self] [2] = "Lbl_1"
               /\ (aState[self] # "unknown")
               /\ \E a1318 \in agt[self]:
                    /\ aState' = [aState EXCEPT ![self] = a1318]
                    /\ agt' = [agt EXCEPT ![self] = agt[self] \ {a1318}]
               /\ pc' = [pc EXCEPT ![self] = [@  EXCEPT ![2] = "Lbl_2"]]
               /\ UNCHANGED << coord, cState, commits, msg >>

\* Process 1 Sub-Process 2 :an action created for the clear function
Lbl_2(self) == /\ pc[self] [2] = "Lbl_2"
               /\ agt' = [a0 \in Agent |-> {}]
               /\ pc' = [pc EXCEPT ![self] = [@  EXCEPT ![2] = "Done"]]
               /\ UNCHANGED << coord, aState, cState, commits, msg >>

\* created for the fairness condition
a(self) == a1(self) \/ a2(self) \/ Lbl_1(self) \/ Lbl_2(self)


\* Process 2 Sub-Process 1 :an action for the statements in label c1
c1 == /\ pc[Coord] [1] = "c1"
      /\ (cState \in {"commit", "abort"})
      /\ agt' = [ag \in Agent |-> agt[ag] \cup  {cState} ]
      /\ pc' = [pc EXCEPT ![Coord] = [@  EXCEPT ![1] = "Done"]]
      /\ UNCHANGED << coord, aState, cState, commits, msg >>

\* Process 2 Sub-Process 2 :an action for the statements in the sub-process
Lbl_3 == /\ pc[Coord] [2] = "Lbl_3"
         /\ IF cState \notin {"abort", "commit"}
               THEN /\ \E c1312 \in coord:
                         /\ coord' = coord \ {c1312}
                         /\ msg' = c1312
                    /\ IF msg'.type = "refuse"
                          THEN /\ cState' = "abort"
                               /\ UNCHANGED commits
                          ELSE /\ IF msg'.type = "accept"
                                     THEN /\ commits' = (commits \cup
                                         {msg'.agent})
                                          /\ IF commits' = Agent
                                                THEN /\ cState' = "commit"
                                                ELSE /\ TRUE
                                                     /\ UNCHANGED cState
                                     ELSE /\ TRUE
                                          /\ UNCHANGED << cState, commits >>
                    /\ pc' = [pc EXCEPT ![Coord] = [@  EXCEPT ![2] = "Lbl_3"]]
               ELSE /\ pc' = [pc EXCEPT ![Coord] = [@  EXCEPT ![2] = "Done"]]
                    /\ UNCHANGED << coord, cState, commits, msg >>
         /\ UNCHANGED << agt, aState >>

\* created for the fairness condition
c == c1 \/ Lbl_3

(* Allow infinite stuttering to prevent deadlock on termination. *)
```

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

Next == c
           \/ (\E self \in Agent: a(self))
           \/ Terminating

Spec == /\ Init /\ [][Next]_vars
        /\ \A self \in Agent : WF_vars(a(self))
        /\ WF_vars(c)

Termination == <>(\A self \in ProcSet: \A thread \in ThreadSet[self] :
    pc[self][thread] = "Done")

\* END TRANSLATION
```

In the translation above, every label is transformed into an action, some actions are also created when needed, for example the receive function and the clean function in process 1 sub-process 2 both modify the same channel and it would be wrong to place the two assignments in the same atomic step thus an auxiliary action is created to hold the body of the clear function.

The affected areas of the translation include the following :

— **ThreadSet**
A set called ThreadSet can be found at line 9, it is the set of all the thread identifiers per process.

— **pc variable**
The *pc* variable in TLA+ that is used to indicate the current point of execution and the next statement to be executed with respect to a process now has to indicate also which sub-process of that process is involved.

At line 21 the pc variable is initialized to a sequence of actions depending on the type of the process, for example if *self* is in Agent, this means it's a process of type Agent that has the following actions associated with it (a1, a2, Lbl_1, Lbl_2), However if we think about the sub-processes for processes of type Agent we'll see that for the first sub-process we have one action that is considered to be the entry point and that is action a1, and for the second sub-process there is also one entry point action and that is Lbl_1, so initially the pc for a process of type Agent will be at either one of these entry point actions.

After the initialization, the pc variable is used as follows **pc[ProcessIdentifier][Sub-Process]**, the modification here is the addition of the second pair of brackets, lines like 25 for example specify that we are currently at the first sub-process of the process, same thing once we want to move to the next action at line 32 we state that we want the execution of the first sub-process of this process to move on to action a2.

5

Code Documentation

5.1 general structure of the toolbox and it's components

try to describe the general flow

5.2 parsing and expansion process

5.3 some software-based diagram

or maybe an AST description graph

6

Conclusion and future work