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

Writing a distributed algorithm is not an easy task which is why most programming languages provide supporting primitives that make this process a bit easier for the programmer, in Distributed PlusCal we as well introduced some of these supporting primitives such as threads and communication channels.

Both PlusCal and Distributed PlusCal translators expect a file with a '.tla' extension, an algorithm that starts with '–algorithm', and the decleration of variables and processes, so there are not many differences in the structure of the file itself, we have added an option to the options in PlusCal with the name 'distpcal', the presence of this option is considered to be the enabler of the Distributed PlusCal translator and once it's found in the options statement then our translator will be the one doing the parsing regardless of the actual content of the PlusCal statements present in the file.

4.1 Threads

A PlusCal process consists of two parts, a variable deceleration part and part that holds the statements to be executed when the process is running. Distributed PlusCal gives each process the opportunity to define more than one part to hold statements to be executed, that is if we consider the first block to be the main thread of a process, then we can think about the added blocks as threads that run in parallel with the main block.

This enables the process to be executing multiple tasks in parallel, for example a thread can be used to send/receive data to/from other processes asynchronously while the main thread is executing it's statement uninterrupted by such commands.

The body of a thread maintains all the same properties and syntax as the body of a normal PlusCal block, all the threads share the same variables declared for the process making synchronization between these threads simple and straightforward.

//to be rephrased

Figure 4.1 shows the general structure of a Distributed PlusCal process written in C-Syntax, the main differences between this structure and the structure of a process in PlusCal are the ability to define multiple threads and the fact that a 'process' can also hold the name 'node'.

```
---- MODULE module_name ----
\* TLA+ code

(* PlusCal options (-distpcal) *)
(*--algorithm algorithm_name{
    variables global_variables

    process (p_name = foo)
        variables local_variables
    { \* begin main thread
        \* pluscal code
    } \* end main thread

    process(p_group \in bar) \* set
    variables local_variables
    {\* begin main thread

      \* pluscal code

    }\* begin main thread
    {\* begin thread

      \* pluscal code

    }\* end thread
    {\* begin thread

      \* pluscal code

    }\* end thread
}
**)
====
```

FIGURE 4.1 – Process Structure

it's important to note that the threads are optional, meaning we can still parse a process with only it's main thread using our translator, also threads do not allow variable declarations they only use variables declared for the entire process.

4.1.1 TLA+ Translation

Defining threads for processes had some effects on the translation to TLA+, because when you declare threads you are actually partitioning the process into multiple sub-parts, so later on whenever we are refering to a process we need to know not only which process it is but also which sub-thread.

Figure 4.2 shows a part of the translation of the Two Phase Commit example.//to be changed to a simpler example then also show Next and Termination

```
1  ---------------------------- MODULE 2pc ----------------------------
2  EXTENDS Sequences, Naturals
3
4  CONSTANTS Coord, Agent
5
6  State == {"unknown", "accept", "refuse", "commit", "abort"}
7
8  \* BEGIN TRANSLATION
9  VARIABLES coord, agt, pc, aState, cState, commits, msg
10
11 vars == << coord, agt, pc, aState, cState, commits, msg >>
12
13 NodeSet == (Agent) \cup {Coord}
14
15 ThreadSet == [n \in NodeSet |-> IF n \in Agent THEN 1..2
16                                  ELSE (**Coord**) 1..2]
17
18 Init == (* Global variables *)
19         /\ coord = {}
20         /\ agt = [a0 \in Agent |-> {}]
21         (* Node a *)
22         /\ aState = [self \in Agent |-> "unknown"]
23         (* Node c *)
24         /\ cState = "unknown"
25         /\ commits = {}
26         /\ msg = {}
27         /\ pc = [self \in NodeSet |-> CASE self \in Agent -> <<"a1","Lbl_1">>
28                                       [] self = Coord -> <<"c1","Lbl_3">>]
29
30 a1(self) == /\ pc[self] [1] = "a1"
31             /\ IF aState[self] = "unknown"
32                   THEN /\ \E st \in {"accept", "refuse"}:
33                             /\ aState' = [aState EXCEPT ![self] = st]
34                             /\ coord' = (coord \cup {[type |-> st, agent |-> self]})
35                   ELSE /\ TRUE
36                        /\ UNCHANGED << coord, aState >>
37             /\ pc' = [pc EXCEPT ![self] = [@  EXCEPT ![1] = "a2"]]
38             /\ UNCHANGED << agt, cState, commits, msg >>
```

FIGURE 4.2 – Process Translation

One of the first things noticeable is the set *NodeSet* at line 18 which is a set of all process identifiers, similarly *ThreadSet* at line 15 is the set of all the thread identifiers per process.

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 had to now indicate also which sub-part of the process is involved, so we extended it to include information per process per thread.
At line 27 we initialize the *pc* variable to be a function whose domain is the set of process identifiers and the range consists of a sequence of labels that are considered the entry points per thread.
Lines 30 and 37 give a clear idea of how the *pc* is expected to be used within the translation.

4.2 Channels

4.2.1 Unordered channels

example with it's translation

4.2.2 FIFO channels

example with it's translation

4.2.3 Supported channel functions

expected syntax and limitations

4.2.4 Walk through of the two phase commit example and its translation

our examples with their translations

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