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

- Model checking is a verification method that is automatic and model-based(system is represented by a model, a specification is represented as a formula and we check whether the model satisfies the formula ==> if not we have counter models), it is intended to be used for concurrent systems.

- The purpose of a modeling language is to describe what a system must perform, not how a system must perform. - the users of these modeling languages are algorithm designers who are responsible for describing the functionality of the system in terms of algorithms before actual implementation. Thus, these languages should be simple so that the users can learn and use the language constructs easily.

- PlusCal language by Leslie Lamport provides simple pseudo-code like interface for the user to express concurrent systems.

- distributed system bugs are difficult to find by testing, they tend to be non-reproducible or not covered by test-cases.

- TLA+ -

2

Background info

This chapter is an overview of TLA+ and PlusCal.

2.1 TLA$^+$

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 2.1 shows a TLA$^+$ specification of a simple memory. It declares four constant parameters *Address*, *Value*, *InitValue*, and *NoValue*, and states a hypothesis on the values that these parameters can be instantiated with. The state space of the specification is represented by the two variables *chan* and *mem*. Intuitively, *mem* holds the current memory, whereas *chan* is an output channel that reflects the result of the preceding operation.

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

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

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

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

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

VARIABLES chan, mem

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

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

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

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

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

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

FIGURE 2.1 – A memory specification in TLA$^+$.

For More details on the syntax and grammer of TLA+, see **??**.

2.2 PlusCal algorithm language

TLA$^+$ is a specification formalism not a programming language, in order for a user to incorporate it properly the user must possess knowledge about mathematical set theory, this is uncustomary among users who write programs and algorithms, Thus PlusCal was proposed to accompany TLA+, for the reason that PlusCal used conventional program constructs.

An algorithm language is used to focus on aspects of the algorithm such as data manipulation rather than irrelevant and distracting details that involve programming-language objects and data structures.

PlusCal is an algorithm language that describes both concurrent and sequential algorithms, it maintains the powerful 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 correct-

ness through the TLC model checker.

A PlusCal algorithm is located in a comment statement within the tla file, the general structure of a PlusCal algorithm is shown in Figure 2.2.

```
(* 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>


**********************)
```

FIGURE 2.2 – 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 depend on 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, macros have the same expansion behavior as C pre-processing macros. **Procedure** in PlusCal take a number of arguments, can define their own local variables and can modify the global variables, however they have no return value.A **Process** begins in one of two ways :

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

The first form begins a process set, the second an individual process. these statements are optionally followed by declaration 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.

All PlusCal statements must be within a label, where statements within the same label are executed atomically. PlusCal enforces a strict ordering of its blocks. The define block has to come before any macros, which has to come before any procedures, which has to come before any processes. The full grammar of the PlusCal algorithm language can be found at appendix A of the PlusCal manual[[?]].

The Figure 2.3 shows the modeling of a semaphore mutex example in PlusCal, Semaphores are integer variables that are used to solve the critical section problem.

The translation process is carried out by the compiler expecting the structure we described as well as some semantic rules, for instance the first statement of any process or procedure as this statement

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

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

FIGURE 2.3 – Semaphore mutex example in PlusCal

marks the starting point of that entity, and macros are not allowed to contain any labels since all the statements in the body are executed as a part of the same atomic step.

The compiler generates the translation by carrying on the following tasks :

1. Generate all the definitions and variables regardless of their scope within the PlusCal algorithm, as well as *vars* the tuple of all variables.

```
\* BEGIN TRANSLATION
VARIABLES sem, pc

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

FIGURE 2.4 – Translation of definitions and variables

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

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

FIGURE 2.5 – Generate *ProcSet*

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 translator produces a variable *pc* that 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 predicate that represents the entry point of execution for the process.

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

FIGURE 2.6 – Generate *Init*

4. Define a TLA$^+$ action for each atomic operation of the algorithm. 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 current process's identifier. The *pc* variable is used to indicate the navigation between the actions for a process.

```
st(self) == /\ pc[self] = "st"
            /\ pc' = [pc EXCEPT ![self] = "enter"]
            /\ UNCHANGED sem

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

cs(self) == /\ pc[self] = "cs"
            /\ TRUE
            /\ pc' = [pc EXCEPT ![self] = "exit"]
            /\ UNCHANGED sem

exit(self) == /\ pc[self] = "exit"
              /\ sem' = sem + 1
              /\ pc' = [pc EXCEPT ![self] = "st"]

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

FIGURE 2.7 – Translation of PlusCal labels into TLA+

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

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

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

FIGURE 2.8 – Generate *Next* and *Spec*

The overall translation strategy can be found [**?**].

3

Related work

There have been other PlusCal extensions, we will be mentioning **??**, a tool represented as a part of a master thesis, it aims to produce an implementation in Go(C based language developed by Google) based on a PlusCal/TLA+ specification.

3.1 PGO

PGo is a source to source compiler written in Java. It compiles specifications written in an extension of PlusCal, called Modular PlusCal to Go programs, PGo can compile Modular PlusCal to PlusCal, PlusCal to Go, and Modular PlusCal to Go.

3.1.1 Modular PlusCal

Modular PlusCal is an extension of PlusCal, using Modular PlusCal the user can separate the specification into two components, a system functionality specification concerned with what the algorithm is supposed to achieve and an environment specification concerned about how we may want it achieved.

For example, if we consider a server/client based communication system, the system functionality can be a client requesting services, this is called system functionality, which is not related to how the client is requesting that service via a TCP connection for example, this is called an environment specification. Another added value for this separation is that the programmer can reuse concept defined in one specification in another specification since the environment specification isn't dependent on the functionality of the system.

Modular PlusCal introduced Architypes, Mapping Macros, and Instances to achieve the modularization of the spec.

Architypes
They are considered to be the blue print of a PlusCal process, they are used to specify the system behaviours. They have the same semantics except for the inability to access global variables unless passed to the architypes as arguments.

Using the *ref* keyword before a global variable that is sent as an argument lets us know that this architype can modify it.

these restrictions provide the needed isolation between system and environment behaviours.

Mapping Macros They specify the environment, they define interfaces for reading and writing on a global variable that represents a network. like pluscal macros they cannot contain any labels inside them, all statements are apart of the same step.

Modeling mapping macros independently from system functionality allows the user to reuse the mapping macros.

Instances Instances are the glue that holds the archtypes and mapping macros together. instantiates a process or a group of processes using the specified architype, each argument passed is bound with a mapping macro to control read and write functions on it.

- show a smaller example or grammer

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 its usage at Amazon Web Services [**?**].

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.

Since we extended the existing PluCal translator responsible for parsing PlusCal into TLA+ we inherited the same semantics and grammar and added our own which can be found in the figure below.

⟨ Process ⟩ ⊨ *process* (⟨ variable decleration ⟩)⟨ local-variables ⟩ ⟨ body ⟩{⟨ body ⟩}
⟨ local-variables ⟩ ⊨ Ø|*variables* ⟨declaration-list⟩ |*channels* ⟨declaration-list⟩ |*FIFOs* ⟨declaration-list⟩
⟨ declaration-list ⟩ ⊨ ⟨ variable-declaration ⟩ | ⟨ variable-declaration ⟩, ⟨ decleration-list⟩
⟨ body ⟩ ⊨ ⟨ labeled-statement-list ⟩

FIGURE 4.1 – Grammar for Distributed PlusCal

In the sections that 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 walk-through on the two phase commit example, the two phase commit is a protocol used for distributed transactions that consist of multiple operations, performed at multiple sites, the goal of it is to reach consensus between the different elements that carry out the transaction together such that if an element decides to abort all other elements abort as well and the transaction is rolled back, and in order to actually commit all elements must agree and be able to commit.

4.1 Communication Channels

PlusCal enables the user to define variables with TLA+ syntax, the variable types can be sets, sequence, sequence of sets etc,

- differentiate sets and sequences and mentions the operators of each with small examples

4.1.1 channels

-set based example with it's translation

4.1.2 FIFO channels

-sequence based example with it's translation

4.1.3 Supported channel functions

expected syntax and limitations - send, receive, broadcast, multicast, clear

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 of labeled statements, This enables the process to be executing multiple tasks in parallel, for example a sub-process can be used to send date to other processes while another sub-process can be responsible for receiving data asynchronously, the programmer can divide tasks between sub-processes and give a sub-process a theme if needed, that is by dividing the algorithm into multiple sub-processes where each sub-process works independently, this is to some extent a form of modularization.

//i've tried saying this modularization thing in different ways by now not sure if the idea is clear or if this is even a 'big deal' for modeling a spec!

// 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, this makes communication between them possible if needed.

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

Listing 4.1 – Distributed PlusCal Sub-Processes

```
1
2  (* PlusCal options (-distpcal) *)
3
4  (***
5  --algorithm TPC {
6
7    \* message channels
8    channels coord, agt[Agent];
9
10   fair process (a \in Agent)
11   variable aState = "unknown"; {
12
13 a1: if (aState = "unknown") {
14           with(st \in {"accept", "refuse"}) {
15             aState := st;
16             send(coord, [type |-> st, agent |-> self]);
17           };
18      };
19      a2: await(aState \in {"commit", "abort"})
20
21   } {
22
23      a3:await (aState # "unknown");
24         receive(agt[self], aState);
```

```
25
26      a4:clear(agt);
27    }
28
29    fair process (c = Coord)
30    variables cState = "unknown",
31              commits = {}, msg = {};
32               \* agents that agree to commit
33    {
34      c1: await(cState \in {"commit", "abort"});
35          broadcast(agt, [ag \in Agent|-> cState]);
36    } {
37
38       c2:while (cState \notin {"abort", "commit"}) {
39          receive(coord, msg);
40               if (msg.type = "refuse") {
41                    cState := "abort";
42               }
43               else if (msg.type = "accept") {
44                    commits := commits \cup {msg.agent};
45                     if (commits = Agent) {
46                        cState := "commit";
47                     }
48                 }
49          }
50    }
51   }
52 ***)
```

The first process consists of two sub-processes, the first one contains the labels (a1, a2) and the second one contains (a3, a4).

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

The variable *aState* is shared between the sub-processes, in fact it is used for communication between them, label a2 holds an *await* statement at line 19 that is waiting for aState to have the value of either *"commit"* or *"abort"*, the variable is being set to one of these values by a3 in the second sub-process by the receive function at line 24.

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

4.2.1 TLA+ Translation

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.

The entire TLA+ translation of the two phase commit example can be found in appendix A.1, now we will be focusing on elements that we introduced to the general structure of the TLA+ file.

— **SubProcSet**

SubProcSet is the set of all the sub-process identifiers per process.

Listing 4.2 – TLA+ translation for Sub-Processes

```
1
```

```
ProcSet == (Agent) \cup {Coord}

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

— **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 it has to indicate also which sub-process is involved.
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, a3, a4), However, a1 is considered the entry point for the first sub-process and a3 for the second one. so we need to initialize pc variable to look something like this

Process Type	Sub-Process	Entry Point
Agent	1	a1
	2	a3
Coord	1	c1
	2	c2

In TLA+ a function with domain 1..n for some n in Nat is a sequence, so the pc values consist of sequences to represent the above table, so we initialize pc in the listing 4.3 to have pc[Agent] = <"a1", "a3"> and pc[Coord] = <"c1", "c2">.
//sequences and sets would've been explained in the previous section for the channels

Listing 4.3 – TLA+ translation for Sub-Processes

```

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","a3">>
                                        [] self = Coord -> <<"c1","c2">>]
```

The example in listing 4.4 shows how the *pc* variable is used, the second pair of brackets that is added when accessing the *pc* variable is used to indicate the sub-process, at line 3 we check that the sub-process is currently at this action, then at line 10 we specify that the next action to be executed for the sub-process is action a2.

Listing 4.4 – TLA+ translation for Sub-Processes

```

\* 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]})
8                     ELSE /\ TRUE
9                          /\ UNCHANGED << coord, aState >>
10               /\ pc' = [pc EXCEPT ![self] = [@  EXCEPT ![1] = "a2"]]
11               /\ UNCHANGED << agt, cState, commits, msg >>
```

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

Appendices

A

Distributed PlusCal to TLA+ Examples

A.1 Two Phase Commit

Listing A.1 – TLA+ translation for Sub-Processes

```
1
2  ------------------------------ MODULE 2pc ------------------------------
3  EXTENDS Sequences, Naturals
4
5  CONSTANTS Coord, Agent
6
7  State == {"unknown", "accept", "refuse", "commit", "abort"}
8
9
10 (* PlusCal options (-distpcal) *)
11
12 (***
13 --algorithm TPC {
14
15   \* message channels
16   channels coord, agt[Agent];
17
18   fair process (a \in Agent)
19   variable aState = "unknown"; {
20
21 a1: if (aState = "unknown") {
22         with(st \in {"accept", "refuse"}) {
23           aState := st;
24           send(coord, [type |-> st, agent |-> self]);
25         };
26     };
27     a2: await(aState \in {"commit", "abort"})
28
29   } {
30
31     a3:await (aState # "unknown");
32        receive(agt[self], aState);
33
34     a4:clear(agt);
35   }
```

```
36
37    fair process (c = Coord)
38    variables cState = "unknown",
39              commits = {}, msg = {};
40               \* agents that agree to commit
41    {
42      c1: await(cState \in {"commit", "abort"});
43          broadcast(agt, [ag \in Agent|-> cState]);
44    } {
45
46       c2:while (cState \notin {"abort", "commit"}) {
47          receive(coord, msg);
48              if (msg.type = "refuse") {
49                  cState := "abort";
50              }
51              else if (msg.type = "accept") {
52                  commits := commits \cup {msg.agent};
53                   if (commits = Agent) {
54                      cState := "commit";
55                   }
56                }
57          }
58    }
59   }
60  ***)
61  \* BEGIN TRANSLATION
62  VARIABLES coord, agt, pc, aState, cState, commits, msg
63
64  vars == << coord, agt, pc, aState, cState, commits, msg >>
65
66  ProcSet == (Agent) \cup {Coord}
67
68  SubProcSet == [n \in ProcSet |-> IF n \in Agent THEN 1..2
69                                   ELSE (**Coord**) 1..2]
70
71  Init == (* Global variables *)
72          /\ coord = {}
73          /\ agt = [a0 \in Agent |-> {}]
74          (* Node a *)
75          /\ aState = [self \in Agent |-> "unknown"]
76          (* Node c *)
77          /\ cState = "unknown"
78          /\ commits = {}
79          /\ msg = {}
80          /\ pc = [self \in ProcSet |-> CASE self \in Agent -> <<"a1","a3">>
81                                            [] self = Coord -> <<"c1","c2">>]
82
83  a1(self) == /\ pc[self] [1] = "a1"
84              /\ IF aState[self] = "unknown"
85                    THEN /\ \E st \in {"accept", "refuse"}:
86                              /\ aState' = [aState EXCEPT ![self] = st]
87                              /\ coord' = (coord \cup {[type |-> st, agent |->
                                    self]})
88                    ELSE /\ TRUE
89                         /\ UNCHANGED << coord, aState >>
90              /\ pc' = [pc EXCEPT ![self] = [@  EXCEPT ![1] = "a2"]]
91              /\ UNCHANGED << agt, cState, commits, msg >>
92
```

```
93  a2(self) == /\ pc[self] [1] = "a2"
94              /\ (aState[self] \in {"commit", "abort"})
95              /\ pc' = [pc EXCEPT ![self] = [@  EXCEPT ![1] = "Done"]]
96              /\ UNCHANGED << coord, agt, aState, cState, commits, msg >>
97
98  a3(self) == /\ pc[self] [2] = "a3"
99              /\ (aState[self] # "unknown")
100             /\ \E a1519 \in agt[self]:
101                  /\ aState' = [aState EXCEPT ![self] = a1519]
102                  /\ agt' = [agt EXCEPT ![self] = agt[self] \ {a1519}]
103             /\ pc' = [pc EXCEPT ![self] = [@  EXCEPT ![2] = "a4"]]
104             /\ UNCHANGED << coord, cState, commits, msg >>
105
106 a4(self) == /\ pc[self] [2] = "a4"
107             /\ agt' = [a0 \in Agent |-> {}]
108             /\ pc' = [pc EXCEPT ![self] = [@  EXCEPT ![2] = "Done"]]
109             /\ UNCHANGED << coord, aState, cState, commits, msg >>
110
111 a(self) == a1(self) \/ a2(self) \/ a3(self) \/ a4(self)
112
113 c1 == /\ pc[Coord] [1] = "c1"
114       /\ (cState \in {"commit", "abort"})
115       /\ agt' = [ag \in Agent |-> agt[ag] \cup  {cState} ]
116       /\ pc' = [pc EXCEPT ![Coord] = [@  EXCEPT ![1] = "Done"]]
117       /\ UNCHANGED << coord, aState, cState, commits, msg >>
118
119 c2 == /\ pc[Coord] [2] = "c2"
120       /\ IF cState \notin {"abort", "commit"}
121             THEN /\ \E c1512 \in coord:
122                       /\ coord' = coord \ {c1512}
123                       /\ msg' = c1512
124                  /\ IF msg'.type = "refuse"
125                        THEN /\ cState' = "abort"
126                             /\ UNCHANGED commits
127                        ELSE /\ IF msg'.type = "accept"
128                                   THEN /\ commits' = (commits \cup {msg'.agent})
129                                        /\ IF commits' = Agent
130                                              THEN /\ cState' = "commit"
131                                              ELSE /\ TRUE
132                                                   /\ UNCHANGED cState
133                                   ELSE /\ TRUE
134                                        /\ UNCHANGED << cState, commits >>
135                  /\ pc' = [pc EXCEPT ![Coord] = [@  EXCEPT ![2] = "c2"]]
136             ELSE /\ pc' = [pc EXCEPT ![Coord] = [@  EXCEPT ![2] = "Done"]]
137                  /\ UNCHANGED << coord, cState, commits, msg >>
138       /\ UNCHANGED << agt, aState >>
139
140 c == c1 \/ c2
141
142 (* Allow infinite stuttering to prevent deadlock on termination. *)
143 Terminating == /\ \A self \in ProcSet : \A sub \in SubProcSet[self]:
        pc[self][sub] = "Done"
144                /\ UNCHANGED vars
145
146 Next == c
147           \/ (\E self \in Agent: a(self))
148           \/ Terminating
149
```

```
150 Spec == /\ Init /\ [][Next]_vars
151         /\ \A self \in Agent : WF_vars(a(self))
152         /\ WF_vars(c)
153
154 Termination == <>(\A self \in ProcSet: \A sub \in SubProcSet[self] :
        pc[self][sub] = "Done")
155
156 \* END TRANSLATION
157 ============================================================================
```

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 clear 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.

- insert labels in example, before await and also for receive and clear then modify the example

A.2 Lamport Mutex