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

$$Init \land \Box[Next]_{vars} \land L$$

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.

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

```
---------------------------- 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$^+$.

2.2 PlusCal algorithm language

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 [Lam09] 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.

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 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 2.2.

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

```
(** 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

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 :

$$\begin{gathered} \mathit{process}(\mathit{ProcName} \in \mathit{IdSet}) \\ \mathit{process}(\mathit{ProcName} = \mathit{Id}) \end{gathered}$$

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. 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 [Lam09].

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.

Translation to TLA$^+$. The PlusCal translator expects as input a PlusCal algorithm following the structure described previously. It 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.

```
(*
--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

```
\* 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 |-> "st"]
```

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 current process's identifier. For example, the following action is generated for label `start` 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* that constrains the possible state transitions, and the complete specification *Spec*.

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

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

A more detailed description of the translation strategy can be found in [Lam09].

3

Related work

- pluscal extensions for generation code to make sure thatverified spec isn't lost while being implemented - **??** 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 behaviors. 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 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. TLA+ provides a flexibility and an expressiveness that enables it specify and verify those algorithms. One of the popular modern examples of incorporating TLA+ to verify distributed algorithms is its usage at Amazon Web Services [NRZ+15].

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 PluCal translator responsible for parsing PlusCal into TLA+ we inherited the same semantics and grammar and added our own.

4.1 Structure of an algorithm

The general structure and organization of a Distributed PlusCal algorithm is shown in Figure 4.1.

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

(* algorithm <algorithm name>

(* Declaration section *)
variables | channels | FIFOs <variable declarations>

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

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

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

(* Processes section *)
process (<name> [=|\in] <Expr>))
  variables | channels | FIFOs <variable declarations>
  <sub-processes>

*)
```

FIGURE 4.1 – General structure of a Distributed PlusCal algorithm

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 4.1 resembles figure 2.2, however the variable declarations in **declarations, procedures, and processes** sections enables the user to 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 4.2.

In the **Process Section** the process can hold multiple sub-processes each with it's own body of statements.More details on the sub-processes are available in section 4.3.

In the sections that follow we will be doing a walk-through on the two phase commit protocol A.1, it 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 other elements abort as well, and the transaction is rolled back.

4.2 Communication Channels

- why important for distributed algorithms - defining channels with 'dimensions'

4.2.1 channels

-set based example with it's translation

4.2.2 FIFO channels

-sequence based example with it's translation

4.2.3 Supported channel functions

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

4.3 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 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 4.3 shows a sub-process defined for the two phase commit algorithm, the example is written in c-syntax and the sub-processes are surrounded by curly braces. The entire algorithm is in Appendix A.1.

Listing 4.1 – Distributed PlusCal Sub-Processes

```

  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);
  }
```

The process 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».

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

4.3.1 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.

Initially a set **SubProcSet** was added to hold all the sub-process identifiers, with respect to which process they belonged to.

$$SubProcSet == [P \in ProcSet| - > IFP \in IdSetTHEN1..nELSE1..m]$$

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 :

$$pc = [self \in ProcSet| - > self \in IdSet - ><< "action", ... >>]$$

where actions in the sequence are the entry point actions for sub-process respectively.

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 A.1, now we will be focusing on the translation of the process in figure 4.3.

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.

Listing 4.2 – Distributed PlusCal Translation- SubProcSet

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

2. Generate the set *pc* variable to as 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".

Listing 4.3 – TLA+ translation for Sub-Processes

```

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, verifying the pc variable at each action based on both the process identifier and the sub-process identifier. In our example, when the *pc* variable is accessed in «"a1", "a2"» the sub-process ID is 1, and from «"a3", "a4"» the the sub-process ID is 2.

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

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 >>

a3(self) == /\ pc[self] [2] = "a3"
            /\ (aState[self] # "unknown")
            /\ \E a1519 \in agt[self]:
                 /\ aState' = [aState EXCEPT ![self] = a1519]
                 /\ agt' = [agt EXCEPT ![self] = agt[self] \ {a1519}]
            /\ pc' = [pc EXCEPT ![self] = [@  EXCEPT ![2] = "a4"]]
            /\ UNCHANGED << coord, cState, commits, msg >>

a4(self) == /\ pc[self] [2] = "a4"
            /\ agt' = [a0 \in Agent |-> {}]
            /\ pc' = [pc EXCEPT ![self] = [@  EXCEPT ![2] = "Done"]]
            /\ UNCHANGED << coord, aState, 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

```

----------------------------- MODULE 2pc -----------------------------
EXTENDS Sequences, Naturals

CONSTANTS Coord, Agent

State == {"unknown", "accept", "refuse", "commit", "abort"}


(* PlusCal options (-distpcal) *)

(***
--algorithm TPC {

  \* message channels
  channels coord, agt[Agent];

  fair process (a \in Agent)
  variable aState = "unknown"; {

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

  } {

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

    a4:clear(agt);
  }

  fair process (c = Coord)
  variables cState = "unknown",
            commits = {}, msg = {};
             \* agents that agree to commit
  {
    c1: await(cState \in {"commit", "abort"});
        broadcast(agt, [ag \in Agent|-> cState]);
  } {

     c2: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";
                 }
              }
        }
  }
 }
***)
\* BEGIN TRANSLATION
VARIABLES coord, agt, pc, aState, cState, commits, msg

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

ProcSet == (Agent) \cup {Coord}

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

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 >>

```

```
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 >>

a3(self) == /\ pc[self] [2] = "a3"
            /\ (aState[self] # "unknown")
            /\ \E a1519 \in agt[self]:
                 /\ aState' = [aState EXCEPT ![self] = a1519]
                 /\ agt' = [agt EXCEPT ![self] = agt[self] \ {a1519}]
            /\ pc' = [pc EXCEPT ![self] = [@  EXCEPT ![2] = "a4"]]
            /\ UNCHANGED << coord, cState, commits, msg >>

a4(self) == /\ pc[self] [2] = "a4"
            /\ agt' = [a0 \in Agent |-> {}]
            /\ pc' = [pc EXCEPT ![self] = [@  EXCEPT ![2] = "Done"]]
            /\ UNCHANGED << coord, aState, cState, commits, msg >>

a(self) == a1(self) \/ a2(self) \/ a3(self) \/ a4(self)

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 >>

c2 == /\ pc[Coord] [2] = "c2"
      /\ IF cState \notin {"abort", "commit"}
            THEN /\ \E c1512 \in coord:
                      /\ coord' = coord \ {c1512}
                      /\ msg' = c1512
                 /\ 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] = "c2"]]
            ELSE /\ pc' = [pc EXCEPT ![Coord] = [@  EXCEPT ![2] = "Done"]]
                 /\ UNCHANGED << coord, cState, commits, msg >>
      /\ UNCHANGED << agt, aState >>

c == c1 \/ c2

(* Allow infinite stuttering to prevent deadlock on termination. *)
Terminating == /\ \A self \in ProcSet : \A sub \in SubProcSet[self]:
    pc[self][sub] = "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 sub \in SubProcSet[self] :
    pc[self][sub] = "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 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

Bibliographie

[Lam09] Leslie Lamport. The pluscal algorithm language. volume 5684, pages 36–60, 08 2009.

[NRZ^{+}15] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Communications of the ACM*, 58 :66–73, 03 2015.