Formal Verification of Distributed Algorithms in +CAL 2.0

M.S. Thesis

defended on 26 June, 2008

by

Sabina Akhtar

Master 2 Recherche
Informatique – Maîtrise du logiciel

Committee:

Dominique Méry Professor UHP-Nancy 1
Noëlle Carbonell Professor UHP-Nancy 1
Didier Galmiche Professor UHP-Nancy 1
Claude Godart Professor UHP-Nancy 1
Guy Perrier Professor Université Nancy 2

Advisors:

Stephan Merz Director of research INRIA Lorraine
Martin Quinson Maître de Conférences UHP-Nancy 1
Acknowledgements

First of all, I would like to thank Mr. Stephan Merz and Mr. Martin Quinson for their willingness and suggestions during my internship. Without their encouragement this work would not have been possible.

Then I would like to thank all my professors including Mr. Didier Galmiche and Mr. Dominique Méry and all others who supported me in my masters program and the Higher Education Commission (HEC), Pakistan for sponsoring my work.

Finally, I would like to thank my family who stood beside me in all the good and bad times.
To My Family,
# Table of Contents

Introduction ................................................................................................................................. 1

1. Background ................................................................................................................................. 2

1.1. Concurrent and Distributed Systems ......................................................................................... 2

1.2. Algorithmic Verification ........................................................................................................... 2

1.3. Temporal logic of actions TLA ................................................................................................. 6

1.4. TLA+ ......................................................................................................................................... 8

1.5. TLC model checker ................................................................................................................... 8

1.6. The preexisting +CAL .............................................................................................................. 8

1.7. Motivations for +CAL 2.0 ......................................................................................................... 11

2. +CAL 2.0 ....................................................................................................................................... 12

2.1. Structure of +CAL algorithm ................................................................................................... 12

2.2. Statements in +CAL 2.0 ........................................................................................................... 16

2.3. The +CAL 2.0 Compiler ........................................................................................................... 18

2.4. Examples of +CAL 2.0 specifications ....................................................................................... 24

3. Conclusions ................................................................................................................................ 27

3.1. +CAL 2.0 Achievements .......................................................................................................... 27

3.2. Conclusions and Future Work ................................................................................................. 28

References ...................................................................................................................................... 30
Introduction

Distributed systems are in the mainstream of information technology. They form the core of many important businesses as multiple distributed units may work simultaneously at multiple parts of a problem. Examples include banks and financial institutions where multiple systems, distributed over the network share a common resource, such as a database. Distributed systems are also being used to work on problems which require more computational power than an individual computer can offer, examples of such projects include the Stanford University Chemistry Department Folding@home project, which is focused on simulations of protein folding to find disease cures and SETI@home, which is focused on analyzing radio-telescope data to find evidence of intelligent signals from space, hosted by the Space Sciences Laboratory at the University of California, Berkeley.

There are some important challenges associated with distributed systems including race conditions, where two or more processes try to access the same resource and deadlock that refers to a situation where two or more processes are waiting for each other to finish and thus neither ever does.

These systems thus should be verified before deployment. There are many formal and informal techniques that have been proposed and used for their verification. Algorithmic verification is a formal technique; it means that the verification is itself performed algorithmically, in contrast to manual or interactive verification, for example using an interactive theorem prover. It is also known as Model Checking; which verifies a given model of the system to satisfy some properties. Algorithmic representation of a system can be achieved using +CAL developed by Leslie Lamport [12]. It contains a translator towards the TLA+ specifications language, which can then be verified using TLC model checker. The +CAL language has some limitations in terms of usage and functionality which make it difficult for an algorithm designer to use, as it requires the knowledge of the underlying architecture which is TLA+ language.

The work that we are going to present here aims at removing some of the limitations of the previous version of the +CAL language and we have tested our implementation by verifying some algorithms. The rest of the report is organized as follows; Section 1 presents the context of our work which includes a brief introduction to concurrent and distributed systems, algorithmic verification and temporal logic of actions as well as a discussion of the previous +CAL focusing on its limitations that motivated this work. Then, Section 2 presents +CAL 2.0 by describing its algorithmic structure, constructs, and the +CAL compiler. Finally, Section 3 discusses the achievements by the +CAL 2.0 and present our conclusions.
1. Background

This section introduces concurrent and distributed systems and the formal technique used for their verification. It also discusses the process of model checking and its corresponding phases. Then, it presents temporal logic of actions and TLA+. And finally, it discusses about the preexisting +CAL and its limitations that motivated this work.

1.1. Concurrent and Distributed Systems

A distributed system consists of multiple processing units geographically distributed working together to solve a particular task, often within heterogeneous environment. On the other hand, in a concurrent system, the units are not geographically distributed they work on the same system. In a distributed system, the units communicate with each other through message passing in order to fulfill the task while, in a concurrent system, they can either communicate using message passing or shared variables. These kinds of systems have to deal with the varying latencies and uncertain failures of processing units or network.

Concurrent and distributed systems pose fundamental algorithmic problems such as achieving mutual exclusion in the use of resources, agreeing on a common response or taking a snapshot of the distributed system’s state. Distributed algorithms [10] are designed to solve these problems in distributed systems and verification techniques are used to establish their correctness. Algorithmic verification is one of the techniques that are used to perform this kind of verification. It requires an algorithm with its specifications and the properties that are required to be verified.

1.2. Algorithmic Verification

Verification is a process that checks whether software meets its specifications or not [1]. It requires the user to express the algorithm or the system according to certain formal specifications. Formal verification is required to ensure that systems produce expected results in real world, despite the errors and failures of their components and environment. Algorithmic verification is a formal approach that aims at a proving or disproving the correctness of an algorithm or a system. There are several techniques for this type of verification, such as model checking.

Model checking is a collection of techniques for automatic analysis of systems [1]. In this technique, the systems or algorithms are represented using a formal model like a transition system. The model checking phases and its process is shown in Figure 1. This process requires 3 basic steps [2], detailed below: Modeling; Specification and Verification.
1.2.1. Modeling

Modeling refers to representing an algorithm or a system as a formal model. This model must represent all the possible behaviors of an algorithm. One of the formalisms used to represent the algorithms is transition systems. As the number of processes increase, the number of states of the formal model increases exponentially. This results in state space explosion.

**Transition system**

A transition system defines the executions of a system, a step-by-step evaluation of a system, starting from some initial state; the system evolves by performing an allowed action which consequently changes the state of system [1]. The transition systems are often extended by additional constraints on admissible runs such as fairness conditions.

**Definition 1:** A transition system \( T = (S, I, \delta) \) is given by a set \( S \) of states, a non-empty subset \( I \subseteq S \) of initial states, and a total transition relation \( \delta \subseteq S \times S \) (that is, we require that for every state \( s \in S \) there exist \( t \in S \) such that \( (s, t) \in \delta \)).

A run of \( T \) is an infinite sequence \( \rho = s_0 s_1 \ldots \) of states \( s_i \in S \) such that \( s_0 \in I \) and for all \( i \in N, (s_i, s_{i+1}) \in \delta \).
1.2.2. Specification

Specification expresses the correctness properties of a system or an algorithm in a suitable formal language. A popular way to represent specifications of concurrent and distributed algorithms is to use temporal logic. Temporal logic is normally used to describe the properties of the system. Typically, these specifications contain temporal logic formulas that represent these properties. The use of temporal logic for writing specifications was proposed by Amir Pnueli [13].

Temporal Logic

The term temporal logic refer to any system of rules and symbolism for representing, and reasoning about, propositions qualified in terms of time [3, 9]. Temporal logic extends standard mathematical logic by operators that refer to the “flow of time” and relate assertions that are true at different instants. The temporal logic differs from a non-temporal logic in the sense that the statements in temporal logic have truth values that vary with time where the truth values of statements in a non-temporal logic are constant in time.

Consider the statement -- “it is raining”: its truth value will change according to time, at some times it may be raining and at some times it may not. In contrast, consider a statement such as “2 + 2 = 4” whose truth values do not change over time. With temporal logic, one can express the statements like “It is always raining in Nancy”, “It will eventually rain this week”, or “it won't rain until early July”.

Thus, temporal logic [3] has an important application as it can be used to state requirements of hardware or software systems. For example, it is possible to express properties such as “whenever a request is made, access to a resource is eventually granted, but it is never granted to two requestors simultaneously.”

Several formalisms are used in practice like LTL, CTL and CTL*. Temporal logic that has the ability to reason about single paths of execution is called linear time logic (LTL), whereas logics that can reason about multiple paths of execution are called branching time logics. These include Computation tree logic CTL and CTL*[3, 9]. Each of these formalisms is detailed below.

- LTL

As defined in [1, 9], Formulas of linear temporal logic LTL are inductively defined as follows:

* Every atomic proposition \( v \in V \) is a formula.
* Boolean combinations of formulas are formulas.
* If \( \phi \) and \( \psi \) are formulas then so are \( X \phi \) ("next \( \phi \)") and \( \phi U \psi \) ("\( \phi \) until \( \psi \)").
* If \( \phi \) and \( \psi \) are formulas then so are \( G \phi \) ("globally \( \phi \)") and \( F \phi \) ("eventually \( \phi \)"). \( G \) is also written as \( \Box \) (read as box) and \( F \) is also written as \( \Diamond \) (read as diamond).

Below we discuss semantics of temporal operators [9]:
* X stands for next; Xφ means it requires φ to hold at the next state
* G stands for always (globally); Gφ means that φ has to hold on the entire subsequent path.
* F stands for eventually (in the future); Fφ means that φ eventually has to hold (somewhere on the subsequent path).
* U stands for until; ψ U φ states that φ holds at the current or a future position, and ψ has to hold until that position. At that position ψ does not have to hold any more.

An LTL formula is evaluated over behaviors, an infinite sequence of truth evaluations and checks satisfaction of a formula for a path of runs, denoted as σ╞φ ("φ holds of σ"). There are two main types of properties that can be expressed using linear temporal logic: safety properties usually state that something bad never happens (G¬φ), while liveness properties state that something good keeps happening (GFψ or G(φ→Fψ)).

– CTL and CTL*

As discussed above an LTL formulas [1] assert properties of single behaviors, but as we are interested in system validity: we say that formula φ holds of T (written as T╞φ) if φ holds of all runs of T. In this sense, LTL formulas express correctness properties of a system. The existence of a run satisfying a certain property cannot be expressed in LTL. Such properties can be expressed in domain of branching time logics. Computational tree logic (CTL) is a branching time logic, meaning that its model of time is a tree-like structure in which the future is not determined; there are different paths in the future, any one of which might be the “actual” path that is realized [9].

CTL logic requires that all the temporal operators should be used with quantifiers ∀ (for all, written as A) or ∃ (there exists, written as E) which express the tree-like aspect of the behavior. Semantics of A and E are to either check all paths from the current state or to check if there exists a path from the current state satisfying the formula.

The CTL* logic can be seen as the combination of the two above logics LTL and CTL.

1.2.3. Verification

Verification is the final process in which a system or an algorithm is verified using the model (transition system) and its specifications along with the set of properties. One should make sure that the model of the system faithfully represents its specifications. The method of verification depends on the type of temporal logic used for the writing the specifications.

The outcome of this process can either be a verification that the system properties hold in all reachable states, or a counter-example where the properties are not satisfied. In addition to these two results, it can also result in a problem of state space explosion meaning that the system size grows above the capacities of the computer running the verification.
For the verification of distributed systems, two classes of properties are of primary importance [5]: safety properties and liveness properties. A safety property expresses that a bad event will never occur. Safety properties are very important to be verified for concurrent and distributed systems such as mutual exclusion. It is a fundamental problem as we have multiple processing units in a concurrent system or geographically distributed units in a distributed system and they might share a resource in order to carry out their task. In this case, we can define a safety property that only one unit can access a shared resource. By verifying this property, we can confirm the correct execution of the system.

Safety properties are satisfied by systems that do nothing, liveness properties are important to assert that the system indeed makes progress. In distributed systems, machines can create many errors including failure. So, we need to verify liveness properties which may include [6]:

* Each request by a machine for service will eventually be answered.
* Messages sent by a machine eventually reach its destination.
* A process that has requested to access the critical section will eventually enter the critical section.

### 1.3. Temporal logic of actions TLA

The temporal logic of actions TLA [4] was developed by Leslie Lamport. TLA combines the logic of actions with linear-time temporal logic. It is used to specify and describe the behaviors of distributed systems. Some important TLA operators and concepts, will be used in the coming sections are as follows:

- **TLA operators**
  The basic temporal logic operators include: □ (read as box), ◊ (read as diamond) and ▷ (read as leads to). For any temporal formula, we define $F \Rightarrow G$ to equal $\mathcal{L}(F \Rightarrow G)$. This formula states that any time $F$ is true, $G$ is true then or at some later time. The two quantifiers that are used in TLA, are same as the ones in the standard first-order logic. They are $\exists$ (read as there exists) and $\forall$ (read as for all). Detailed information regarding these operators can be found in [3,4]

- **Action formulas**
  Actions or action formulas are predicates that are evaluated over two states. Syntactically, they are boolean expressions built from unprimed variables, primed variables and constant symbols. For example, $x' + z = y$ and $z' < 3$ are actions, where $x$, $y$ and $z$ are variables. The unprimed variables refer to the values of the variables in the old state and the primed variables refer to the values of the variables in the new state. The old state signifies the state in which an action is executed and the state that is produced by applying that action is called the new state. The meaning $[A]$ of an action $A$ is a relation between states – a function that assigns a boolean $s[A]/t$ to the pair of states $s$, $t$. We write $s[A]/t$ by considering $s$ to be the “old state” and $t$ to be the “new state”.


- State formulas and state functions

A (possibly non-Boolean) expression that does not contain primed variables is called a state function and an action formula that does not contain primed variables is called state formula. Action formulas are used to express the initial conditions of a system, as well as invariants of a system. A variable can also be a state formula.

Now, we can also define the action UNCHANGED $f$, for $f$ a state function, by

$$UNCHANGED f \equiv f' = f$$

where $f'$ refers to the new value of $f$. Thus, an UNCHANGED $f$ step is one in which the value of $f$ does not change.

- The Enabled predicate

We can define $Enabled A$, for any action $A$, to be the predicate that is true for a state if and only if it is possible to take an $A$ step starting in that state. The predicate $Enabled A$ can be defined syntactically as follows:

If $v_1, \ldots, v_n$ are all variables that appear in $A$, then

$$Enabled A \equiv \exists c_1, \ldots, c_n : A(c_1/v'_1, \ldots, c_n/v'_n)$$

- $[A]_f$ and $<A>_f$

For any action $A$ and state function $f$, we can define

$$[A]_f \equiv A \lor (f' = f)$$

where $f$ will often be a tuple that includes all variables that appear in $A$. This means that either an $A$ step is taken or the state function $f$ remains unchanged. It is also called a stuttering step.

We can define $<A>_f$ for any action $A$ and state function $f$, such that:

$$<A>_f \equiv A \land (f' \neq f)$$

where $f$ is any state function. This action formula states that $A$ occurs while $f$ changes simultaneously. This formula is used to represent liveness property.

- Temporal formulas:

In TLA, algorithm execution is viewed as sequence of steps, and each step is producing a new state by changing the values of variables. So, semantically an algorithm is a collection of all its possible executions. Temporal logic is used to reason about algorithms [4]. So, in TLA temporal formulas are used to represent all system behaviors.

The overall system behavior is represented using the following formula:
\( \text{Init} \land \Box [\text{Next}]_{\text{vars}} \land \text{Liveness} \)

\( \text{Init} \) is defined as a predicate specifying the initial values of variables. \( \text{Next} \) is the next-state relation that is the disjunction of possible actions which can be taken. \( [\text{Next}]_{\text{vars}} \) states that either a stuttering step is taken or any of the possible actions in the next-state relation. And the last part \( \text{Liveness} \), represents in general the fairness conditions that are assumed.

1.4. TLA+

TLA+ is a complete specification language that is based upon the temporal logic of actions. It was developed by Leslie Lamport [11]. TLA+ combines TLA and mathematical set theory [8]. It allows specifying and reasoning about concurrent and distributed systems. Temporal logic of actions is used to describe the system and their properties. The use of set theory allows TLA+ to be more expressive and easier for specifying algorithms at a high level of abstraction.

TLA+ has module structure. The body of a module contains a list of statements. These statements can be declarations, definitions, assumptions, or theorems. Further modules can also be added as sub-modules. TLA+ module structure is defined in detail in [14].

1.5. TLC model checker

TLC was primarily developed by Yuan Yu. It is a model checker for verifying a TLA+ specification by checking TLA properties of a finite-state model of the specification [7]. TLC takes a file containing the TLA+ model of the algorithm and a configuration file that contains the specification information, constant definitions, and the properties or invariants that need to be verified by the TLC model checker. It explores all the reachable states, looking for some execution of the system where:

- an invariant or other temporal property expressed by a TLA formula, including liveness properties are not satisfied, or
- deadlock has occurred. It means that there is no possible next state to be explored.

1.6. The preexisting +CAL

+CAL was proposed by Leslie Lamport [12] as a high level algorithmic language to generate TLA+ code for concurrent and distributed algorithms. Since TLA+ is not a programming language but rather a specification formalism, +CAL eases the use of TLA+ for algorithm designers. +CAL provides very simple statements to express non-deterministic algorithms. Moreover, +CAL is rather an algorithmic language than a programming language. Leslie Lamport spots the following differences between algorithms and programs:

- Algorithms perform operations on arbitrary mathematical objects while programs perform operations on simple objects like booleans and integers.
- An algorithm is a generalized form of solving a problem and it can result in multiple solutions but a program gives only a single method for solving a problem.
- There is no concept of a step in a program while in an algorithm, each step is clearly defined by labels and an execution of an algorithm consists of a sequence of steps.
The +CAL language is available in two different syntaxes, P-syntax and C-syntax. P-syntax is similar to the Pascal language, while the C-syntax is closer to the syntax of the C programming language. We now present the model of the alternating bit protocol [12] as an example.

The alternating bit protocol is a distributed message-passing algorithm. A sender and a receiver communicate over FIFO channels msgC and ackC.

The sender sends a message \( m \) by repeatedly sending the pair \((m, sbit)\) on the channel \( \text{msgC} \), where \( sbit \) is 0 or 1. To acknowledge the receipt of the message, the sender repeatedly sends \( sbit \) on channel \( \text{ackC} \). After receiving the acknowledgement, the sender complements \( sbit \) and begins sending the next message.

The +CAL code using P-syntax for this algorithm is shown below:

```
1 --algorithm ABProtocol
2 variables input = <<>>; output = <<>>; msgC = <<>>; ackC<<>>;
3
4 macro Send(m, chan)
5 begin
6   chan := Append(chan, m)
7 end macro
8
9 macro Rcv(v, chan)
10 begin
11   when chan # <<>>;
12     v := Head(chan);
13     chan := Tail (chan)
14 end macro
15
16 process Sender = "S"
17 variables next = 1; sbit = 0; ack;
18 begin
19 s:while (TRUE) do
20 either
21 with m \in Msg do
22     input := Append(input,m)
23   end with;
24 or
25 when next <= Len(input);
26 Send(<<input[next], sbit i>>, msgC)
27 or
28 Rcv(ack, ackC);
29 if ack = sbit then
30   next := next + 1;
31   sbit := (sbit + 1)%2
32 end if;
33 end either;
34 end while;
35 end process
```
process Receiver = "R"
variables rbit = 1; msg;
begin
  r:while (TRUE) do
    either
      Send(rbit, ackC )
    or
      Rcv(msg, msgC);
    if msg[2] # rbit then
      rbit := (rbit + 1)%2;
      output := Append(output, msg[1])
    end if;
  end either;
end while;
end process

process LoseMsg = "L"
begin
  l:while (TRUE) do
    either
      with i \in 1.. Len(msgC) do
        msgC := Remove(i, msgC)
      end with;
    or
      with i \in 1.. Len(ackC ) do
        ackC := Remove(i, ackC )
      end with;
    end either;
end while;
end process
end algorithm

This algorithm appears within a TLA+ module that imports the other modules and declares constants for the TLC model checker. The +CAL translator uses the same file for writing the TLA+ code, inserting it between the lines begin translation and end translation that should appear in the TLA+ module. These lines are written as follows:

\* BEGIN TRANSLATION
\* END TRANSLATION

The usage of same file for writing the TLA+ code makes it difficult for the user to work in +CAL, as it always requires a reload for updating the +CAL code.

The translated algorithm can be validated by running TLC in simulation mode, where non-deterministic choices are resolved randomly. It can be verified by using TLC in model checking mode, where it exhaustively generates all executions. +CAL has many statements that are used to express non-determinism. In the above example, with and either statements are used to show non-determinism. For example, inside label “s” at line 19, we have a “with” statement starting from line 21 to 23. This statement will be processed by executing the following statement which is inside “with” statement:

input := Append(input, m)

with a value for m equal to a non-deterministically chosen element from the set Msg.
1.7. Motivations for +CAL 2.0

+CAL is a very powerful algorithmic language. It contains all the statements that are required to express an algorithm as they are in other programming languages. But in +CAL language, some of the concepts are missing. For example, there is no concept of variable scope, while in other higher level languages the scopes of the variables are strictly defined. So, a programmer working in +CAL language can make errors, this will result in verification of an algorithm that is not correct. The example given in previous section has 3 processes, a Sender process on lines 16 – 35, a Receiver process on lines 37 – 51 and a LoseMsg process on lines 53 – 66.

Two of them having their own local variables, for example, the process named Receiver has a variable rbit as its local variable on line 38. But other processes can easily access it and modify its value. The translator never complains, so, a programmer can mistakenly modify it somewhere else as well and he will end up getting wrong results.

Lamport intended +CAL to be accessible for algorithm designers. The algorithm designers have to learn the new language +CAL but in addition to that they also have to learn the underlying architecture that is TLA+ language, and in particular know how +CAL constructs are represented in TLA+. Learning a complete new architecture can be a very difficult task. The algorithm designers have to face this problem while checking fairness conditions or when they are want to verify an invariant or a temporal property. Actually, the expressions of invariants or properties require the same identifiers that were generated by the compiler in TLA+ code.

For the example given in the previous section, we might want to check the liveness property that every message that is chosen is eventually delivered. This is expressed by the following temporal logic formula, which asserts that for every \( i \), if \( \text{input} \) contains \( i \) elements then \( \text{output} \) will eventually contain \( i \) elements [12]:

\[
\forall i \in \text{Nat} : (\text{Len}(\text{input}) = i) \Rightarrow (\text{Len}(\text{output}) = i)
\]

But a user has to define this formula inside the TLA+ translation and it also requires the configuration file to be modified for the TLC model checker to verify this property. So, a user must know about the TLA+ language.

Another issue that motivated this step was the difficulty to express concurrent algorithms where entities share some state variables and distributed algorithms where the state of all the entities are hidden from the other ones. This was not possible because of single level of processes. In practice, another level of processes is also used that are called threads.

Some other shortcomings of the original +CAL algorithm that we tried to address are as follows:

- Labels define the atomic steps in +CAL algorithm but a user is not allowed to place them freely in the algorithm.
- Because of technical issues in the translation to the TLA+ formalism, it was not possible to modify a variable twice within an atomic step while such construct is often required in algorithms.
2. +CAL 2.0

Our work consisted of designing a new version of +CAL that remedies the problems mentioned above and writing a compiler for it. The previous +CAL had some limitations as stated in the previous section and we tried to remove some of them in order to make +CAL a self-contained language for modeling and verifying distributed algorithms with its ability of expressing non-determinism. The user can write algorithms in the new +CAL without having to learn the underlying architecture that is TLA+ language.

In this section, first, we introduce the algorithm structure of +CAL 2.0 then we discuss the statements for writing algorithm in +CAL and finally we discuss the +CAL compiler.

2.1. Structure of +CAL algorithm

The structure of +CAL algorithm is designed to accommodate all the algorithmic verification information that is required by a user to be verified. We briefly explain the basic structure of a +CAL algorithm with the example code from the FastMutex algorithm detailed in Section 2.4. We can divide the structure in 6 basic sections, as follows:

- Header section
- Declarations section
- Process section
- Main block section
- Property section
- Instance section

2.1.1. Header section

This section includes the information regarding the name of algorithm, the TLA+ modules to be imported and the constants that need to be declared by the programmer that will be used by TLC during model checking phase. The initialization of the constant symbols is done in the instance section which will be explained later. A sample form of header section is shown below:

```plaintext
algorithm <<algorithm name>>
EXTENDS <<modules to be imported>>
constant[s] <<constant names>>
```

“algorithm”, “EXTENDS” and “constants” are reserved words. The EXTENDS and constants sections are optional and can be omitted if they are not necessary. An example of this section is shown below:

```plaintext
algorithm FastMutex
EXTENDS Naturals,TLC,Sequences,FiniteSets
constant N
```
2.1.2. Declarations section

This section contains the declarations of global variables and functions. The function declarations are optional. Declaration sections also appear as part of the definitions of processes and threads. The only difference between the declaration section of main algorithm and process or thread’s definition is the scope of that declaration. It will be accessible only within that definition. The variable declaration and its syntax is as follows:

```
variable[s] <<variable names>>
```

Variable names are separated by commas and they can be assigned an initial value. An example code is shown below:

```
variables x = 0, y = 0, b = [i \in 1..N |-> FALSE], count = 0
```

Declarations section also contains the declaration and definition of functions and procedures. Each function or procedure declaration is followed by its definition. Its structure is as follows:

```
function/procedure <<function/procedure name>>([parameters])
   <<variable declarations>>
   <<function/procedure body>>
```

A function or procedure’s name is followed by its parameter values. Its definition also contains variable declarations as explained earlier. The variables included in this part will be local to that function or procedure and they can never be accessed by any other function, procedure, process, thread or main algorithm. We can also have multiple function or procedure declarations. They can even have same names but with different number of parameters.

2.1.3. Process section

This section contains declarations of processes followed by their definitions. We can define multiple processes in this section. A sample structure of a process declaration and definition is as follows:

```
[[strongly] fair] process <<process name>>([parameters])
   <<variable declarations>>
   <<function declarations>>
   <<thread declarations>>
   <<process body>>
   <<property specifications>>
```
The reserved words “strongly” and “fair” are optional. They are used to indicate the fairness property of the current process. In the previous version, the user was required to add the fairness property in the generated TLA+ code and it required some further modifications as well. Now, the fairness conditions are assumed and the invariants or properties are checked for every instance of the process that will become active during program execution.

The process name can be followed by its parameter values. Default values can also be defined by the user. Two different processes can also have same names but with different number of parameters. The property specifications are defined in the property section mentioned in Section 2.1.5 below. A sample code from FastMutex example is shown below:

```plaintext
fair process Proc(id)  
    variable j = 0  
begin  
    ncs: loop  
        begin  
            ...  
        end  
    end  
end
```

In the thread declaration part, the user can declare threads along with their definitions. This part is also optional and can be omitted if not required. This means that a process can only access its own threads and no other process can use them. The structure of a thread declaration is as follows:

```plaintext
[[strongly] fair] thread <<thread name>>([parameters])  
    <<variable declarations>>  
    <<function declarations>>  
    <<thread body>>  
    <<property specifications>>
```

The user can specify fairness for threads as well. The name of the thread is also followed by its parameter values. Threads can access the local variables and functions of the process within which they are defined. The property specifications are defined in the property section.

The processes and threads are not executed at the time of initialization of TLC model checker. The only entity that starts is the main block. Processes and threads are created dynamically using the "run" statement described in Section 2.2.4.
2.1.4. Main block section

This section contains the main execution block of any algorithm. This is the only thread of execution that is started at the initialization of TLC model checker. The syntax of the main block is:

```
begin
  <<statements of the block>>
end
```

A function, process or thread body will have the same format for their corresponding block sections. An example is shown below:

```
begin
  run Proc(1);
  run Proc(2);
end
```

2.1.5. Property section

This section contains the invariants and the temporal properties to be verified for the corresponding block. For example, if this section is included after the process body, then it is verified for each instance of that process. A user can specify multiple invariants and temporal properties. They are optional as well. The syntax for writing these invariants and properties is as follows:

```
invariant <<invariant to be verified>>
temporal <<temporal property to be verified>>
```

The reserved word “invariant” is followed by the expression that represents the invariant to be verified. And the reserved word “temporal” is followed by the expression that represents the temporal property to be verified. At the time of TLA+ code generation, these invariants and the properties are added to the TLA file. An example is shown below:

```
invariant count < 2
```

2.1.6. Instance section

This section contains the initializations for the constants declared in header section and the constraints.

```
constant[s] <<constant declarations>>
```
An example code from FastMutex algorithm is shown below:

```
constant N = 2
```

### 2.2. Statements in +CAL 2.0

The statements in +CAL 2.0 are very simple and for the most part similar to the statements of previous +CAL. A user can add a label at the start of any statement; fairness of individual statements can be defined in +CAL 2.0 by annotating the label with a fairness declaration as in:

```
[[strongly] fair] <<label name>>: <<statements>>
```

The statements of +CAL 2.0 are explained below with their corresponding structure and sample code from the FastMutex algorithm in Section 2.4.

#### 2.2.1. Assignment

An assignment statement is either an assignment to a variable or to an element of the variable if it is a record or an array. Two sample examples from FastMutex algorithm is shown below:

```
count := count - 1;
b[id] := FALSE;
```

The first one is a simple assignment statement, while the other one is an assignment to the element id of array b.

#### 2.2.2. Branch

A `branch` statement is combination of “if”, “either” and “when” statements that existed in the previous version. Its structure is shown below:

```
brach
    <<condition>> then
    begin
      <<statements>>
    end
or
    <<condition>> then
    begin
      <<statements>>
    end
```
The structure of branch statement specified above will behave like an “if” statement. However, if the “else” block is not specified then it will behave like the “either” statement in the previous version of the +CAL. Further, if a user only specifies single block within a branch, without any “or” and “else” blocks, then it would work like “when” statement in the previous version of the +CAL.

The branch statement can have multiple “or” blocks, and any of those blocks whose condition evaluates to true is chosen (non-deterministically) for execution. It is similar to the “either” statement. So, this single statement represents all the 3 statements of the previous +CAL.

2.2.3. Loop

A loop statement repeats the execution of its block of statements. A goto statement can be used to leave the loop body of the loop. The structure of the loop statement is shown below:

```plaintext
loop
  begin
    <<statements>>
  end
```

2.2.4. Run

A run statement is used to execute a process or thread. It prepares the execution stack for the process or thread. In previous +CAL, all processes were started at the initialization of the algorithm; now a user can execute a process at any time within the +CAL algorithm. The structure of run statement is as follows:

```plaintext
run [process/thread] <<process/thread name>>;
```

An example of run statement from FastMutex algorithm is shown below:

```plaintext
run Proc(1);
```

A run statement can also be used with an assignment statement to store the ID of the process or thread.
2.2.5. Call

A call statement is used to execute a function or procedure. The structure of a call statement is shown below:

```
call [function/procedure] <<function/procedure name>>;
```

2.2.6. Return

A return statement is used to return the execution control to the label, next to the label from which the function/procedure was called. Its structure is as follows:

```
return;
```

2.2.7. Goto

A goto statement is used to transfer the execution control to the given label. Its structure is as follows:

```
goto <<label name>>;
```

2.2.8. Skip

A skip statement does not have any affect. It simply passes the control to the next statement. The structure is as follows:

```
skip;
```

2.2.9. Print

A print statement is same as skip statement. The only difference is that it asks TLC to print the value of a given variable or expression. It is written as follows:

```
print <<expression>>;
```

An expression can be a single or composition of variables and constants. It can also be a text within quotes.

2.3. The +CAL 2.0 Compiler

The structure of the +CAL 2.0 compiler is represented in the class diagram shown in Figure 2. Compilation of a +CAL algorithm consists of multiple phases in order to parse and generate its corresponding TLA+ translation. Compilation phases are shown in Figure 3. The first phase includes parsing of the +CAL algorithm which generate an abstract syntax tree
(AST), representing the structure of the source code. In the second phase, the algorithm is translated into intermediate language that generates an exploded tree and finally it is translated to TLA+ language that results in TLA file and configuration file. The intermediate language can be considered as a simplification of +CAL statements that are closer to TLA+ language. The intermediate language of an algorithm is represented by the class “ExplodedTree”. It will be explained later in the Section 2.3.2. The compilation process results in the creation of two files. One of them contains the translation of the algorithm into TLA+ code, while the other one is the configuration file that contains the constant initializations, invariants and properties to be verified by the TLC model checker.

Figure 2: Class diagram of +CAL compiler

The core classes of the +CAL compiler are PcalParser, which generates the AST, PcalTranslator, which translates the AST into the intermediate language, and
PcalTLAGenerator, which writes the TLA and configuration files. Each of these classes will be explained in more detail in the following sections.

2.3.1. PcalParser

The +CAL parser is generated by a JavaCC grammar written for +CAL. JavaCC is a parser generator that is similar to flex/bison. During this phase, the algorithm is analyzed for syntactic errors. The algorithm must follow the grammatical rules in order to proceed to the next stage of translation. The parser also maintains a symbol table which is represented by the class “SymbolTable”. This data structure is used to check the scope of each symbol in the algorithm and it also contains other information regarding the type of the symbols. This phase generates an abstract syntax tree (AST) that represents the +CAL algorithm. This tree is then passed on to the next phase that performs the translation to the intermediate representation.

2.3.2. PcalTranslator

This phase takes an AST tree representing a +CAL algorithm and explodes it into simplified format called intermediate language (described below) that is closer to the TLA+ language. This two-stage translation simplifies the structure of the compiler. During this phase, an algorithm is transformed to atomic execution blocks identified by their corresponding labels in the original algorithm. If labels are not present in the algorithm then they are added by the translator during this phase. The labels define atomic steps and are used for compilation purposes.

Figure 3: Compilation process of +CAL compiler.
The labels must follow the same rules as in the previous version of +CAL. The rules for the insertion of labels are as follows:

1. The first statement of the main algorithm, function, procedure, process or thread must be labeled.
2. The loop statement must be labeled.
3. If a statement is preceded by any of the following statements then it must be labeled:
   - return statement.
   - function or procedure call statement.
   - goto statement.
   - process or thread execution statement represented as “run” statement in new +CAL.
   - a branch statement that contains any of the above 3 statements.

**Intermediate language**

The intermediate language is defined as a simplification of +CAL statements in a form that is closer to TLA+ language. This language only contains two syntactic constructions (namely branches and assignments) whereas the +CAL syntax contains over 20 constructions, as detailed in Section 2.2. An explicit "pc" variable is used to represent control flow and allow jumps of execution from any point in the intermediate code. This language is organized into multiple atomic blocks represented by the label names.

A loop statement is represented as follow in the intermediate language:

<table>
<thead>
<tr>
<th>+CAL statement</th>
<th>Intermediate Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbl: loop</td>
<td></td>
</tr>
<tr>
<td>begin</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;body of loop&gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pc := lbl1;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In this case, the body of the loop must contain simple statements like assignment statements. If it contains other statements that require labels or it simply contains labels then it is split into different steps according to their corresponding labels. Similarly, a branch statement is translated as follows:

<table>
<thead>
<tr>
<th>+CAL statement</th>
<th>Intermediate Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbl: branch</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;condition1&gt;&gt; then</td>
<td></td>
</tr>
<tr>
<td>begin</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;body1&gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>or</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;condition1&gt;&gt; :</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;body1&gt;&gt;</td>
</tr>
<tr>
<td></td>
<td>pc := lbl2;</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;condition2&gt;&gt; :</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;body2&gt;&gt;</td>
</tr>
</tbody>
</table>

21
The intermediate translation of an algorithm is represented using the class “ExplodedTree”. An exploded tree contains the simplification of +CAL statements of the algorithm. It also contains other information regarding invariants and properties that need to be added to TLA file for verification.

2.3.3. PcaTLAGenerator

This is the phase that performs actual TLA+ code generation. This phase takes an exploded tree that represents the input algorithm in simplified form and creates two files for the TLC model checker, TLA file and configuration file. This phase also takes care of the variables that are assigned new values, multiple times within an atomic step using LET/IN constructs in the generated TLA+ code. The Figure 4 shows an example.

The TLA generation phase is performed in multiple stages. First, the header section of TLA file is generated. The header section contains the module name which is created using the name of the file, the modules in the EXTENDS part to be imported by TLC model
checker and the constants that were declared by the user in the original algorithm. The constants are not initialized in the header section of TLA file. Instead, they are initialized in the configuration file. The constants initializations that contain functions or sequences cannot be added to configuration file. So, they are added as definitions in the TLA file. For example, if we have a constant with function initialization:

\[
MyFunc = \{id \in MySet \rightarrow FALSE\}
\]

Then, it will be added to TLA file as a definition as follows:

\[
MyFunc == \{id \in MySet \rightarrow FALSE\}
\]

This section also contains the variable declarations and their initializations. Only the global variables of the original algorithm are added in this section.

In the next stage, the TLA generator reads the exploded tree and generates the corresponding TLA+ code. Each label gives rise to a TLA+ action, represented as an operator in TLA file. So, an action is defined by its corresponding statements. Some extra guards are also added with the definition of each operator. Each action contains an UNCHANGED conjunct to specify which variables are not updated at this step and the variables that have changed are also updated using separate conjunctions.

After the translation of labels and their corresponding statements, the next-state relation is generated. It is defined as the disjunction of the actions representing the main block and the actions of all the active process and thread instances. An extra disjunct prevents TLC from declaring deadlock at normal termination of the algorithm. For example, the FastMutex algorithm given in Section 2.4 will have its next-state relation defined as follows:

\[
Next == _\text{Main}(1)
\]

\[
\text{// (\forall self \in _\text{Proc}\_\text{IDSet} : \_\text{Proc}(self))}
\]

\[
\text{// (_pc[1] = "Done")}
\]

\[
\text{// (\forall self \in _\text{Proc}\_\text{IDSet} : _pc[self] = "Done")}
\]

\[
\text{// UNCHANGED vars)
\]

\_\text{Main} is defined as an operator in the TLA+ code. Its definition contains the names of the operators that were labels in the original algorithm and represented the main block.

TLA+ specifications are defined by the Spec operator. It has the following form:

\[
\text{Spec} == \text{Init} \lor [\text{Next}]_\text{vars} \lor L
\]

where \text{Init} is the predicate that specifies the initial conditions. \text{Next} is the next state relation as defined earlier. It states the next step to be taken. The last part of the specifications represents the fairness conditions to be assumed for the algorithm.

The fairness conditions for the atomic steps or the labels can also be added if required. For example, if a user wants to check weak fairness for a label “Lab_1”, as follows:
If the above statement is with the main block of the algorithm, then its corresponding
fairness condition will be:
\[ \text{WF\_vars}(\text{Lab}_1(1)) \]

If that statement is within the body of a process with name “Proc”, then its
corresponding fairness condition will be:
\[ \forall p \in \_\text{Proc\_IDSet} : \text{WF\_vars}(\text{Lab}_1(p)) \]

This states that the weak fairness for the label “Lab_1” must hold for all the instances
of the process “Proc”. The above fairness condition implies that the step “Lab_1” must
eventually occur.

Now, if the algorithm contains any invariants or temporal properties to be verified,
then it adds them as definitions at the end of the TLA+ code. Finally, the configuration file is
created. The specification information is added to this file. If there exist invariants or temporal
properties to be verified then their operator names in the TLA file are added to their
_corresponding section. For example, if there is one invariant, Inv0 and one temporal property,
Temp0, they will be added to configuration file as follows:

\[
\text{INVARIANT} \quad \text{Inv0} \\
\text{PROPERTY} \quad \text{Temp0}
\]

The names of the invariants and temporal property formulas are created by default. If
there are any fairness conditions to be assumed, then they are added to the specification
section. By default, the specification section is as follows:

\[
\text{SPECIFICATION} \quad \text{Spec}
\]

2.4. **Examples of +CAL 2.0 specifications**

Here we present some of the algorithms written in the +CAL 2.0.

2.4.1. **FastMutex Algorithm**

The first example is the FastMutex algorithm [15]. It is designed to avoid
simultaneous use of a common resource. In the example below, are competing for the critical
section represented by label “cs” at line 77. This algorithm can be written in +CAL 2.0 as
follows:

```
1 algorithm FastMutex
2 EXTENDS Naturals,TLC,Sequences,FiniteSets
3 constant N
4 variables x = 0 , y = 0 , b = [i \in 1..N |-> FALSE], count = 0
5 fair process Proc(id)
6 variable j = 0
7 begin
```
```plaintext
ncs: loop
begin
  skip ;
start:  b[id] := TRUE ;
x := id ;
branch
  y # 0 then
  begin
    b[id] := FALSE ;
    branch
      y = 0 then
      begin
        goto start;
      end
    end
  else
    begin
      skip;
    end
  end
y := id ;
branch
  x # id then
  begin
    b[id] := FALSE ;
j := 1;
  end
  loop
  begin
    branch
      j <= N then
      begin
        branch
          ~b[j] then
          begin
            j := j+1;
goto 19;
          end
        end
      else
        begin
          skip;
        end
      end
j := 1;
branch
  y # id then
  begin
    branch
      y = 0 then
      begin
        goto start;
      end
    end
  else
    begin
      skip;
    end
end
```

---

25
To verify the mutual exclusion of the 2 processes, we have used a global variable “count”. Each process increments it before entering the critical section and decrements it on leaving the critical section. This mean that count should never be greater than 1. The invariant that needs to be verified, is at line 88 that is as follows:

\[ \text{invariant count} < 2 \]

This invariant is successfully verified by the TLC model checker. The results produced by TLC stated that it was able to produce 44 states for this algorithm and out of them, 29 states were distinct states.

2.4.2. Peterson’s Algorithm

The next example is Peterson’s algorithm in the +CAL 2.0 that is shown below along with its intermediate translation for reference.
3. Conclusions

This section discusses the Achievements made by +CAL 2.0 and finally we conclude our work.

3.1. +CAL 2.0 Achievements

The previous +CAL version had some limitations which motivated us to develop a new +CAL. We discuss briefly how we managed to overcome those limitations.
(a) Learning TLA+ compilation process

We addressed this problem by allowing the algorithm designer to add fairness hypotheses as well as invariants or temporal properties inside the +CAL algorithm. The new +CAL provides the space for the user to add properties in the property section. Property section is available after process, thread and the main algorithm block, as discussed above. A user can also check fairness conditions for a process, thread or any label. They simply require the user to add the reserved word “fair” or “strongly fair”. Now, a user can add invariant or property expressions without having to modify the generated TLA file. Furthermore, a user had to modify the configuration file for instantiating the constants possibly requiring auxiliary definitions in the TLA+ module. Now, they can be easily instantiated in the “instance” section of the algorithm.

(b) Variable scoping problem

This issue is addressed using stack for all the processes and threads. Their corresponding local variables are put on the stack and they can use them by accessing it. For example, if there is a process with name “proc” then its corresponding variables and parameters will be on a stack named “_proc_data”. And whenever a variable of a process is updated or accessed, all the operations are performed on the corresponding stack of that process. So, only the global variables in +CAL algorithm are made global for TLA+ code.

(c) Single level of processes

In +CAL 2.0, a process can have multiple threads performing their tasks. These threads are executed independently like processes and they all communicate with each other via message passing or shared variables, local to their parent process. This helps in expressing concurrent systems. On the other hand, multiple processes do not have shared space as they are physically separated. This allows us to express distributed systems.

(d) Other minor problems

The problem that was about labeling the +CAL algorithm is also addressed. In +CAL, atomicity is defined by labels and a user is required to add the labels according to some rules. Now, a user can add labels where he wants and the compiler will add additional ones if required. The other minor problem was about modifying a variable twice within an atomic step or within a label. Now, the user can easily place multiple variable modification statements in the same atomic step. It renames the variable each time it is assigned a new value, and uses the last renamed variable if it used within in the expressions. An example is shown in the Figure 4.

3.2. Conclusions and Future Work

Distributed systems are in the mainstream of information technology. They are necessary when data needs to be processed that is generated at geographically distributed locations. Distributed systems are also attractive for increasing the throughput and the reliability of systems compared to centralized solutions. They form the core of many important businesses as multiple distributed units may work simultaneously at multiple parts of a problem, examples include banks and financial institutions where multiple systems, distributed over the network share a common resource, such as a database. This poses some important challenges for distributed systems including race conditions, where two or more processes try to access the same resource and deadlock that refers to a situation where two or
more processes are waiting for each other to finish and thus neither ever does. These systems thus should be verified formally before deployment. Model checking is a formal algorithmic verification technique. In this technique, programs are expressed in form of algorithms to perform this kind of verification. Verification is fruitfully performed at the level of (abstract) algorithms because they generate fewer states than actual implementations; also, errors found at this level are easier to correct than errors found only at the end of the development phase. +CAL is a tool that can be used to perform this type of verification because it is a translator towards the TLA+ specifications language which can then be verified using the TLC model checker.

Our work is a step towards making +CAL easier to use for the algorithm designers. We have tried to make it more practical as it is closer to pseudo-code notations and is formally verifiable. We started with the previous version of +CAL, developed by Leslie Lamport. Our implementation of a sample algorithm using the previous version of the +CAL language highlighted its limitations in terms of usability and functionality. There is no concept of variable scoping and all variables either locally defined to a function or to a process are accessible in other processes and functions, which affect both its usability and functionality. Then, the user has to learn about the process of compilation from +CAL to TLA+ in order to check the fairness conditions or for the verification of invariants and temporal properties, which further hinders its usability. Further, there are some other minor shortcomings including the constraint that a variable should not be modified twice within an atomic step. These limitations paved the way for +CAL 2.0.

The major achievement of +CAL 2.0 is verification of not just concurrent systems but also distributed systems. The previous +CAL was meant for only concurrent systems. We achieved this by allowing two processes to have their own local space for doing their job and running their own threads. No two processes can access each other’s space. For concurrent systems, it is achieved by a single process that has its own threads which can access the variables and function local to their parent process.

+CAL can be studied and experimented further to enhance its features for the ease of algorithm designers. Due to short period of time, we were not able to build a complete +CAL language compiler with all of the statements. These statements include: “foreach” and “with” statement which allow enumeration over sets, “break” statement which allows loops to be terminated, “assert” statement, “atomic” statement which allows the user to define its own atomic steps, parallel assignment statement. Moreover, some other functionalities are missing like functions don’t return a value. But it can be further improved and experimented. It can then be used to verify algorithms with very complex structures.

Another issue that can be studied is regarding the output of the TLC model checking tool. Its results are very complicated to understand especially when it results in counter-example. It outputs the last state of the system with an error reference to the TLA+ code and it is difficult for the users to find the exact problem in the +CAL code. Although the user of +CAL 2.0 does not need to know about the details of compilation to TLA+ for writing algorithms, counter-examples produced by TLC are still presented at the TLA+ level, exposing the translation. It would be useful to translate them back to the level of +CAL 2.0. We intend to take these shortcomings into account before a public release of +CAL 2.0.
References


