N/A

Auteur : Leruth, Guillaume

Promoteur(s) : Fontaine, Pascal

Faculté : Faculté des Sciences appliquées

Diplôme : Master : ingénieur civil en informatique, à finalité spécialisée en "computer systems security"

Année académique : 2023-2024

URI/URL : http://hdl.handle.net/2268.2/20979

Specifying and Verifying Safety Properties of Parallel Programming Algorithms Using the TLA$^+$ Toolbox

Supervisor
Pascal Fontaine, Professor - ULiège

Co-Supervisor
Stephan Merz, Director of Research - INRIA Nancy

Master's thesis completed in order to obtain the degree of
Master of Science in Computer Science and Engineering

by Leruth Guillaume

LIÈGE
université

University of Liège
-
School of Engineering and Computer Science
Academic year 2023-2024

Acknowledgments

I would first like to express my utmost gratitude to my promoter and professor Pascal Fontaine for encouraging me to pursue this Master's thesis although my situation did initially not meet the ideal prerequisites for this topic. Without both of the continued academic guidance and moral support he kindly offered me, this endeavor would not have been possible.

I would then like to extend my sincere gratefulness to my co-promoter Stephan Merz, Senior Researcher at the INRIA Nancy research center, whose insights on the TLA$^+$ environment and precious advice were invaluable to the production of this Master's thesis. His contributions to the specification of Barz's algorithm offered a well-needed kickstart to my understanding of the TLA$^+$ Toolbox.

I also wish to mention the cheerful and warm welcome I received from the members of the VeriDis team during my stay in Nancy. Thanks to them, I had the opportunity to catch a glimpse of the world of research in formal methods within a productive environment that significantly pushed this work forward.

Finally, on a more personal level, I want to express my warmest thanks to my family and friends for supporting me throughout my entire academic journey. I owe every single one of my achievements to their unconditional benevolence and encouragement.

Abstract

As implied by its title, the goal of this Master's thesis is to specify and verify safety properties of a few classical algorithms taken from the Parallel Programming lectures taught by Prof. Pascal Fontaine by using the TLA$^+$ tool suite. The original motivation was to constitute a formal proof companion for the course which would allow to spot remaining mistakes in the lecture material while providing an alternative unambiguous description of the algorithms and their properties for the students.

The first chapter of this work lays out the minimal concepts of concurrent programming required to motivate the distinctive significance of software verification in the world of parallel and distributed algorithms. The second one presents all the tools of the TLA$^+$ environment used in the following chapters, from the underlying TLA mathematical model up to the proof manager that automatically verifies properties of an algorithm. A rough methodology is established to turn the pseudo-code descriptions of algorithms presented in the lecture material into clear TLA$^+$ specifications and to consequently identify and express safety properties of interest before checking and proving them.

The next chapters consist in applying the previously described tools and methods to algorithms taken from the Parallel Programming course, namely Barz's algorithm and the Readers-writers. The resulting TLA$^+$ specifications, adjoining definitions and properties as well as TLAPS proofs constitute the original contribution of this Master's thesis.

Finally, the conclusion chapter briefly sums up the results obtained in those chapters.

Contents

1 Parallel Programming 1

1.1 Concurrent Programming . . . 1

1.2 Interleaving Semantics . . . 1

2 The TLA$^+$ Language and Tools 3

2.1 Overview . . . 3

2.2 Temporal Logic of Actions . . . 4

2.3 Expressing Programs with the TLA Logic . . . 8

2.4 PlusCal . . . 12

2.5 From PlusCal to TLA$^+$ Specifications . . . 14

2.6 The TLC Model Checker . . . 17

2.7 TLAPS . . . 19

2.8 TLATeX . . . 21

2.9 General Methodology . . . 21

3 Barz's Algorithm 26

3.1 Semaphores . . . 26

3.2 Historical Context . . . 27

3.3 The Algorithm . . . 27

3.4 TLA$^+$ Specification . . . 29

3.5 Inductive Invariants . . . 32

3.6 Introduction to Refinement . . . 37

3.7 Abstract Specification for the General Semaphore . . . 38

3.8 Refinement Mapping . . . 40

3.9 Refinement Proof . . . 44

4 Readers-Writers 46

4.1 Problem Setting . . . 46

4.2 Proposed Solutions . . . 47

4.3 PlusCal Specification . . . 49

4.4 Simple Inductive Invariants . . . 51

4.5 Main Inductive Invariant . . . 51

5 Conclusion 58

Chapter 1

Parallel Programming

1.1 Concurrent Programming

Running an algorithm in a parallel setting has a potential to increase the performance that can be expected compared to its sequential version. On the other hand, parallelism introduces new inherent issues that have the potential to jeopardize the correctness of the algorithm. As portrayed by the first chapter of [2], concurrent programming is an abstract setting that allows to write and reason on programs that may at least partially be run in parallel. It is thus to be differentiated from any concrete parallel or distributed programming environments like software threads, multi-core processors or computer clusters.

It is to be considered that the concurrent sections of a program allow a finite number of abstract processes to execute the given sequence of instructions "at the same time". As a result of the previous comment, this notion of time should not be related to any practical implementation of parallelism. Therefore, it is not even necessary to assume the processes really work in parallel. Rather, simply supposing they work at arbitrary individual rates, within a common timeframe is sufficient. This vague model encompasses both fully parallel schemes and executions in which the processes simply take turns doing their job, as well as any behavior in between.

1.2 Interleaving Semantics

The only decisive feature of the concurrent programming model described in the previous section is the associated interleaving semantics. As presented in [2], from an external spectator's point of view, an execution of the abstract concurrent programming model consists in a sequence of steps, each taken by an arbitrary process, with nothing happening in between. These actions are individually considered as an instantaneous modification of the state of the algorithm and can thus never be interrupted by another step taken by a different process. Such steps are referred to as atomic since they define the minimal level of granularity for expressing any change in the current model.

In order to correctly reason about concurrent programs, one must consider that the next atomic step of a concurrent execution might be taken by any of the processes unless there is a reason explicitly preventing them to do so. Hence, compared to its sequential equivalent, the abstract concurrent programming model of a deterministic algorithm allows for multiple sequences of steps to be generated. In other words, although the processes may follow the same relative execution paths, the order in which the atomic steps of the different processes are interleaved on the abstract timeline can vary from run to run. Therefore, when reasoning about the execution of a concurrent program, all possible interleavings of atomic steps must be considered.

Beyond the practical aspect that the number of possible executions to be considered may easily become prohibitive, there is a way more vicious concern regarding the interleaving semantics defined hereabove. The current model indeed does not incorporate shared memory yet although its use is pervasive in order to allow processes to collaborate. However, as soon as one considers that part of the state of the system is accessible to multiple processes, the independence between them is waived which means the choice of interleaving might have an impact on the behavior of the algorithm.

For example, the outcome of a toy concurrent program simply returning the value of a variable shared by two processes depends on which process performs the last assignment operation in the interleaving. Given that the two processes P_1 and P_2 set the variable to values 1 and 2 respectively, the program outputs 1 if P_2 takes its atomic step first. The initial value 2 set by P_2 is indeed subsequently overwritten by P_1. Conversely, it is equally possible under the interleaving model for the program to output 2 due to P_1 being picked to take a step first.

The above example clearly illustrates the idea that the interleaving semantics may introduce an additional source of nondeterminism that does not arise in the sequential programming setting. Such a dependence of the behavior of an algorithm on the particular execution run is referred to as a race condition when it hinders the correctness of the algorithm with respect to an established specification. In real parallel algorithm implementations, race conditions are known to be the cause of so-called Heisenbugs. Such bugs compromise the overall correctness of a program while being extremely hard to detect using traditional debugging approaches. They are indeed tied to specific interleavings that may hardly ever occur when running the program. Avoiding this category of bugs is the principal motivation behind trying to apply formal verification methods to concurrent algorithms.

Chapter 2

The TLA$^+$ Language and Tools

2.1 Overview

The name TLA stands for “Temporal Logic of Actions” (or alternatively “Three-Letter Acronym”) and initially refers to the formal system of logic introduced by Leslie Lamport in [9], in 1994. Prior to this, as stated by Lamport himself in [7], he was trying to specify and prove properties of concurrent programs using the previously developed linear temporal logic. However, after becoming “disillusioned with temporal logic when he saw how Schwartz, Melliar-Smith, and Fritz Vogt were spending days trying to specify a simple FIFO queue”, he left them and went on to develop TLA.

For its part, TLA$^+$ (note the + symbol) designates the high-level formal modeling language consequently created by Lamport in order to provide a practical and elegant way to specify and check concurrent systems based on TLA. Moreover, TLA$^+$ may also refer to the companion IDE and its toolbox which offer the full set of capabilities gravitating around the original language. The goal of this section is to briefly present the different modules and how they interact, before diving in a detailed description of each of them in the rest of this chapter.

The easiest way to start with TLA$^+$ is to download and install the IDE[1]. Although this IDE is officially referred to as “The TLA$^+$ Toolbox”, the provided tools could, strictly speaking, be used outside this particular editor environment. They are notably available with various degrees of integration as extensions geared towards more mainstream IDEs.

Beyond the obvious text editing, file management or syntax highlighting

[1] Access the TLA$^+$ Home Page at https://lamport.azurewebsites.net/tla/toolbox.html

capabilities offered by the TLA$^+$ Toolbox for both TLA$^+$ and its companion language PlusCal, its defining features encompass the following:

- The "SANY' Syntactic Analyzer parses both languages for errors in specifications.
- The PlusCal Translator transpiles more intuitive PlusCal specifications to their TLA$^+$ equivalent which can be processed by the following tools.
- The "TLC" Model Checker allows to derive concrete models from abstract specifications and to check safety properties by exhaustive enumeration of the reachable states.
- The "TLAPS" Proof System is an add-on developed by the Microsoft Research-INRIA Joint Centre. It is thus not distributed along the current version of the TLA$^+$ Toolbox IDE and must be downloaded and plugged in manually[2]. The provided TLA Proof Manager is able to automatically check formal proofs of TLA$^+$ assertions written in a dedicated proof sub-language by converting them to obligations that are in turn fed to multiple backend provers.
- The "TLATeX" Pretty-Printer allows to typeset the ASCII-formatted specifications for display in PDF documents by using the LaTeX engine.

2.2 Temporal Logic of Actions

The goal of this section is not to exhaustively and formally describe the logic of TLA as in the original paper [9] by Lamport. Many details are indeed not required to understand the content of this Master's thesis and are introduced to solve issues that do not even arise in the following. As a result, although heavily based on this paper, the following section will be more of an introduction to the main concepts behind TLA and should be sufficient to grasp all the consequent developments.

As an introduction, it could be said that Lamport's driving motivation lies at the beginning of [9], under the sentence "Logic is the formalization of everyday mathematics, and everyday mathematics is simpler than programs". Hence, a defining characteristic of TLA is that everything will be represented using a unified mathematical framework, from a program itself, to the properties one wants said program to satisfy and even the formal proofs that these properties are verified. Thus, although the syntax of TLA$^+$ and especially PlusCal may end up looking like the one of a programming language, the semantics behind it are always those of TLA which are essen-

[2] See instructions: https://proofs.tlapl.us/doc/web/content/Download/Binaries.html

tially mathematics. As such confusion could obviously happen throughout this Master's thesis, the end goal being to specify computer programs, the previous statement will be repeated and detailed any time the reader may be misled.

On the other hand, a shortcoming of TLA$^+$ and comparable formal specification tools used for software verification can be inferred from the previous citation. Indeed, although such systems are able to provide strict theoretical guarantees regarding the abstract mathematical model, it is the user's job to construct the specification according to the original program's behavior. Otherwise, any derived guarantee is de facto invalidated. Therefore, and because real-world physical computer systems suffer from way more causes of error than logical faults in software, one would be dishonest as to claim any form of absolute guarantee of proper functioning based on the output of a tool like TLA$^+$. Despite the previous argument, verifying an algorithm with TLA$^+$ does by no way constitute a useless process completely disconnected from the real implementation. In many cases, this method indeed allows to uncover critical mistakes that would be almost impossible to identify using other methods. This is particularly true for the Heisenbugs introduced in Section 1.2.

First of all, the starting idea of TLA (and standard temporal logic before it) is to mathematically represent the behavior of computer systems by discrete-time dynamic systems. This implies reducing any form of memory of the physical devices to the abstract notion of state. The execution of a program is thus seen as a sequence of discrete steps which describe how to transition from the current state to a new one. This view will of course eventually overlap with the notion of atomic steps associated to the interleaving semantics.

Using the formalism of Lamport's paper [9], **Var** designates the infinite set of possible variable names and Val a collection of any value that might be useful. In this setting, a state s is no more than a mapping from **Var** to Val assigning the value $s[\![x]\!]$ to a variable name x. The set of all possible such states is then referred to as **St**. One can notice the previous definitions are rather loose but this results from the choice of Lamport to take advantage of the full expressiveness of mathematics in order not to introduce any constraint at this stage.

A notable consequence of this approach is the fact that TLA$^+$ variables are not typed at all, being no more than textual labels that can be attached to anything, just like mathematical definitions. As a result and similarly to dynamically typed programming languages, a same TLA variable can be associated to a set by one state, a natural number by another or even a whole specification by a third. This has the benefit of being extremely flexible but comes to the price of TLA$^+$ being unable to ascertain a variable respects a given type over the course of the execution of the system without a formal proof.

Building on top of the previous notion of state, Lamport introduces state functions and predicates as expressions of variables and constant symbols like x^2+y-3 and $x^2 = y-3$ respectively. They both express a mapping from states in **St** to "values" but predicates only map them to the booleans true and false (which are thereby not strictly considered values) while functions work on all the values of Val. The derivation of a value from the expression of the function or predicate is formally stated by the following definition:

$$s[\![f]\!] \triangleq f\ (\forall `v' : s[\![v]\!]/v)$$

Practically, the value or boolean associated by the function or predicate f to the state s is obtained by replacing all textual occurrences v of variables by the value $s[\![v]\!]$ associated to them by said state s. In the other way around, the underlying philosophy might be understood more easily by considering variables will take values that change from state to state and that the value of a function is computed for a given state based on the value of some of the variables in this state.

Up to this point, individual states have been defined along formulas that can be evaluated over them. The original motivation was nevertheless to represent the execution of a program by a sequence of states linked by transitions. This is where the need for the concept of temporal logic arises. The object of study of temporal logic is indeed to reason about a whole sequence of states, generally referred to as a behavior. On the other hand, atemporal formulas like the ones presented up to now involve a single state and can be dealt with using conventional first-order logic. The basic element of reasoning is thus now a sequence σ belonging to the set $\mathbf{St}^\infty$ of all infinite sequences built by picking states from $\mathbf{St}$.

Many temporal operators can be defined to express various properties of a behavior σ but the only one that is required to go further in understanding TLA is the "always" operator $\Box$. When applied to a so-called elementary formula, this unary operator makes it a predicate on the whole sequence σ rather than on a single state s. Simply put, the resulting temporal formula evaluates to the boolean true if and only if the inner formula is verified for every state s appearing in the sequence σ. Lamport gives the following formal definition:

$$\langle s_0, s_1, s_2, \dots \rangle\, [\![\Box F]\!] \triangleq \forall n \in \mathbb{N} : \langle s_n, s_{n+1}, s_{n+2}, \dots \rangle\, [\![F]\!]$$

In other words, the temporal formula $\Box F$ associates the same boolean to the infinite sequence σ composed of states $s_0, s_1, s_2, \dots$ as F when universally quantified on the states of σ. The missing implicit assumption for this interpretation to be correct is that, as an elementary predicate F can only be evaluated on one state at a time, doing so on a sequence of states is an alias for evaluating it on the first state of this sequence only. The right-hand side of the above definition is thus indeed equivalent to saying F must be true for any state s_n appearing in σ in order for $\sigma[\![\Box F]\!]$ to be true.

The tools developed up to now allow to mathematically characterize the behavior of a computer system as expected. However, referring to the beginning of Section 2.1, classical linear temporal logic was already available to Lamport and his colleagues while they were trying to specify a FIFO queue. It was nevertheless not sufficient to prevent him from leaving the project out of frustration. The shortcoming of this logic, which Lamport subsequently addressed by introducing TLA is that, it does not say anything about the links between consecutive states of a behavior. On the other hand, the traditional imperative programming approach typically consists in a sequence of instructions, each describing how to mutate the current state into the next one. This notion of transition may implicitly result in restrictions applied by the program to the states that can be reached during its own execution. It can however not be expressed using linear temporal logic only. Properties associated to those restrictions are thus impossible to state or prove. As a result, a considerable part of the information embedded in the program is de facto left aside by the current model.

The above argument calls for the introduction of a new object in the logic, coined "action" by Lamport and giving its name to the resulting Temporal Logic of *Actions*. Intuitively, actions are similar to the state predicates defined earlier but are boolean expressions on a pair of states rather than on a single one. The expression of an action may thus contain a new primitive under the form of primed variables. Primed variables allow to differentiate the values taken by variables in either of the two states forming the pair and to thereby express relations between them as in $x' = y + 1$. The formal definition of actions is given by Lamport as follows:

$$s[\![\mathcal{A}]\!]t \triangleq \mathcal{A}\left(\forall\text{`}v\text{'} : s[\![v]\!]/v, t[\![v]\!]/v'\right)$$

Similarly to the previous definition of state predicates, the boolean attributed by the action $\mathcal{A}$ to the ordered pair of states (s, t) is derived by replacing all occurrences of unprimed variables v by the value $s[\![v]\!]$ given to them by state s and primed variables v' by the value $t[\![v]\!]$ from state t. In that sense, boolean predicates are the subset of actions that do not make use of primed variables in their expression. Thereby, their "value" does not depend at all on the second state t of the pair (s, t) they are evaluated over.

Now interpreting actions in the context of temporal logic, the semantics of "current" and "next" state can be associated to the states s and t respectively. As a result, two states s and t following each other in a temporal sequence can now be related using an action $\mathcal{A}$ whose expression is designed to map true to the pair (s, t) and false to any other value of $\mathbf{St}^2$. This is also described as (s, t) being an $\mathcal{A}$ step. Now, evaluating an action $\mathcal{A}$ over a behavior $\sigma = \langle s_0, s_1, \dots \rangle$ can be defined as simply verifying if its first pair of states (s_0, s_1) is an $\mathcal{A}$ step or not. Thanks to this alias, the previous definition of the $\Box$ operator at the base of temporal logic is extended to elementary formulas containing actions without modification. The following equation can be written to describe this formally:

$$\begin{aligned} \langle s_0, s_1, s_2, \dots \rangle \, [\![\Box\mathcal{A}]\!] &\equiv \forall n \in \mathbb{N} : \langle s_n, s_{n+1}, s_{n+2}, \dots \rangle \, [\![\mathcal{A}]\!] \\ &\equiv \forall n \in \mathbb{N} : s_n [\![\mathcal{A}]\!] s_{n+1} \end{aligned}$$

In other words, an action $\mathcal{A}$ is always true, in the sense of the $\Box$ operator, over a behavior $\sigma = \langle s_0, s_1, s_2, \dots \rangle$ if and only if it is true for every pair of consecutive states (s_n, s_{n+1}) appearing in the sequence.

The previous developments roughly correspond to the so-called "Raw Temporal Logic of Actions" (RTLA) introduced by Lamport. Many subsequent steps would be required to add liveness, fairness along other properties and take the current logic from RTLA to Simple TLA and then finally the full-fledged TLA. Among those, only stuttering steps will be introduced in the following section as they are necessary for the refinement of Barz's algorithm presented in Section 3.8. The rest of the conception of TLA as well as more formal descriptions of the preceding can be found in the original paper [9] but are beyond the scope of this Master's thesis.

2.3 Expressing Programs with the TLA Logic

Going back to the original objective of the previous section to abstract computer programs as sequences of states, RTLA allows to express rules on the transitions followed by the system each time a state change happens in its behavior. All the prerequisites to translating an algorithm to its TLA specification have thus been developed.

It is however crucial to first state that a TLA specification is a temporal logic formula and nothing else. This may seem trivial at this point since nothing else has been developed, but it is easy to lose track of it. Indeed, as stated in the introduction to this chapter, while the line between a computer program and its formal specification is clear at this stage, it may fade away in the following, as the practical tools reconcile both worlds in a sometimes misleading way.

Therefore and to state it clearly, a real-world program is a sequence of instructions telling the computer what initial state to start from and what operations to execute one after another to modify said state. A program thus produces a single execution trace each time it is run, which corresponds to a single behavior σ in the abstract formalism. Due to different causes of nondeterminism like dependence on values provided by a human user, deliberate (pseudo-)randomness or the interleaving semantics presented in Section 1.2, the behavior σ resulting from the execution of the program may nonetheless arbitrarily change from run to run.

On the other hand, a TLA specification is no more than a logic formula expressing a predicate on said behaviors. The specification must thus be seen as a boolean function taking as input any behavior σ out of the theoretical set $\mathbf{St}^{\infty}$ of all possible behaviors over the state of the program. The goal is therefore to write the expression of the specification formula so as to map exactly all the achievable behaviors of the original program to the boolean true. Hence, any other undesired behavior of $\mathbf{St}^{\infty}$ is associated to false.

In order to build such an expression for the specification formula, a natural structure fortunately stems from the model that has been chosen to represent programs. First, programs typically express constraints on their initial state. For example, variables may be assigned an explicit initial value along their declaration. Similarly, a program might expect arguments of a predefined type. This typically translates to a stricter set of possible values than the loose Val default collection of TLA. Moreover, most programs only have a single identified entry point for the control flow to begin at.

The previous remark highlights the fact that high-level constructs of usual programming languages may hide part of the complete state of the computer system by abstracting it away. In this case, the control flow from one instruction to the next seems so natural that it is easy to overlook the low-level program counter required to implement it. As a result, the control flow of a program must be explicitly accounted for in specifications by introducing a variable simulating the program counter. However, automatic management of this auxiliary program counter variable comes out of the box when using PlusCal as described in Section 2.4. It will thus not be detailed further here.

To sum up, all the above constraints can be expressed by a first-order state predicate generally called *Init* and constructed as a conjunction of assertions about the value of the different variables present in the algorithm. As a result, evaluating *Init* alone over the behaviors of $\mathbf{St}^{\infty}$ already allows to rule out any of them whose first state thus does not match the initial constraints of the target program.

Then, the main endeavor of the specification process is to translate each instruction, or more precisely atomic operation, of the program to a TLA action. This step is rather trivial as it suffices to look at all the variables of the program individually and determine which are affected by the operation. An action expression must consequently be written for each of them, laying out how to derive their new value using the primed variable notation. Putting syntax differences aside, this relation is generally what the original instruction was intended to express in the first place.

There is however a subtle nuance in that an action should also be provided for all the unchanged variables. In order to understand such a requirement, one should refer to imperative programming languages, which implicitly consider the value of a variable to be unaltered unless affected by an assignment. On the other hand, if the expression of a TLA action were to say nothing about the new value of a variable, nothing could be concluded at all. As a result, the action would undiscriminatingly accept both the steps that modify the variable and those that leave it untouched. Because such ambiguity is typically not desired, TLA provides syntactic sugar of the form *Unchanged* f for the action defined by expression $f' = f$.

At this stage, one TLA action has been written for each atomic operation of the program. Note that those actions should incorporate the whole control flow of the program. All possible modifications of the program counter are indeed explicitly described like for any other variable. It is thus possible to create a new formula by considering the disjunction of all the previously defined actions. The resulting composite action is referred to as *Next* (or $\mathcal{M}$ in [9]) and is only true for the pairs of states that respect one of the legal transition in the original system. In this sense, when provided with the current state, this action can be seen as a predicate that discriminates the possible next states of the system from the rest of **St**.

Combining the previously developed *Init* and *Next*, one can simply build the temporal formula $Init \land \Box Next$. Given any behavior $\sigma = \langle s_0, s_1, s_2, \dots \rangle$ belonging to $\mathbf{St}^\infty$, the resulting predicate first checks that the initial state s_0 verifies *Init*. Due to the always $\Box$ operator, all transitions between consecutive states s_n and s_{n+1} should then respect one of the actions that make up *Next*. Thanks to this formulation, the goal expressed at the beginning of Section 2.2 to represent a computer system as a discrete-time dynamic system is finally fulfilled.

The formula structure developed hereabove could be sufficient as a canonical form for TLA specification. But, as explained at the very end of the previous section, stuttering steps have been left out up to now, although they constitute a crucial aspect of TLA. As their name indicates, stuttering steps are actions that cause the system to "repeat itself" instead of changing state. An attentive reader may realize that no additional primitive is required to express such a formula. It is indeed sufficient to write $var_1' = var_1 \land var_2' = var_2 \land \dots$ for all the variables of a specification to produce an action that repeats the current state. The philosophy behind stuttering step is thus not to consider them as transitions of the system and to explicitly incorporate them into *Next*, but rather to always keep them as an alternative beside the actions of *Next*. Once again, TLA provides a shorthand notation of the form $[\mathcal{A}]_f$ to indicate that the stuttering step $f' = f$ over variable f can be taken as an alternative to action $\mathcal{A}$. In other words, such a formula simply constitutes syntactic sugar for $\mathcal{A} \vee (f' = f)$.

The motivation for the addition of stuttering steps to a TLA specification will be perfectly illustrated in Section 3.8 with the refinement of Barz's algorithm. In order to understand it at this stage, the example given by Lamport in [9] will be used. Simply put, he considers two clocks, one includes a hand for seconds while the other one does not. He then supposes that any behavior of the former should also be accepted by the latter. In order to relate the behaviors of both systems, it suffices not to look at the second hand of the first clock. Physically, the two systems would then be equivalent, the hour and minute hands of the both of them being in perfect sync.

However, when constructing TLA specifications without stuttering steps, the two systems seem to accept radically different behaviors. Indeed, the second clock accepts behaviors that increment the minute variable at each step until loopback. On the other hand, the first one requires 60 steps to increment the same variable by 1, as it must deal with its additional second variable beforehand. Hence, when trying to relate both specifications by ignoring the second variable just like for the physical systems, the first clock systematically stutters 59 times before taking a step, creating a mismatch with the behavior of the second one. This argument is sufficient to determine that the second specification does not accept all the behaviors of the first one like expected.

Intuitively, this issue is due to the distorted notion of temporality of the TLA logic. Indeed, the unit of time on its abstract timeline is one action, regardless of it being a stuttering step or not. Thus, as corresponding steps are not taken at the same point in the sequence of states of both clocks, TLA considers they do not happen "at the same time". The solution is thereby simply to leave the possibility for the second clock to stutter unconditionally, like described previously. As a result, the specification accepts infinitely many more behaviors because an arbitrary number of stuttering steps can be inserted at any point of a previously accepted sequence of states. With the addition of stuttering steps, it becomes clear that the second specification accepts all behaviors of the first one as initially intended.

In order to preventively inhibit similar problems, the canonical form of a TLA specification with the addition of stuttering steps is given below, using the previously defined notation:

$$\begin{aligned} \mathit{Spec} &\triangleq \mathit{Init} \ \wedge \ \Box\,[\mathit{Next}]_{\langle \mathit{var}_1, \mathit{var}_2, \dots \rangle} \\ &\equiv \mathit{Init} \ \wedge \ \Box\big(\mathit{Next} \vee \big(\mathit{var}_1' = \mathit{var}_1 \wedge \mathit{var}_2' = \mathit{var}_2 \wedge \dots\big)\big) \end{aligned}$$

2.4 PlusCal

In a similar way to the original TLA paper published in 1994, PlusCal was formally presented to the public in the 2009 paper [8] by Leslie Lamport. PlusCal can be categorized as a Domain Specific Language which is intended to be transpiled to a TLA$^+$ specification. The abstract of Lamport's paper clearly states the reasons that incited him to add PlusCal to the capabilities of the TLA$^+$ suite:

> Algorithms are different from programs and should not be described with programming languages. The only simple alternative to programming languages has been pseudo-code. PlusCal is an algorithm language that can be used right now to replace pseudo-code, for both sequential and concurrent algorithms ...

In other words, PlusCal is meant to bridge the gap between pseudo-code, which is the preferred way to express an abstract algorithm in a language-independent manner, and TLA$^+$ specifications that results from a radically different approach. This is of course a crucial step because, as pointed out previously, incoherence between the source algorithm and the TLA$^+$ specification would render any subsequent work useless.

The goal of this section is not to exhaustively lay out the syntax of the PlusCal language, as it does its job of looking like plain pseudo-code very well, but rather to present the general structure to adopt when specifying concurrent algorithms. First, the PlusCal "code" must be enclosed in a TLA$^+$ comment block using the `(* *)` delimiters. Indeed, once the specification has been transpiled from PlusCal to TLA$^+$, the translation is appended at the end of the same file by the Toolbox. The original PlusCal formulation does thereby not play any role in the following steps and is "hidden" as a comment. Upon parsing the specification after a file modification, SANY however checks if both version are still coherent by using a checksum associated to the TLA$^+$ translation. In order to set a PlusCal section apart from a regular comment, the `--algorithm` token must be used at the beginning and followed by an arbitrary name for the algorithm. Its content must then be delimited like any construct that has a "body" section like *while* loops, $if - then - else$ conditional jumps or *macro* definitions, by either using the C-style curly braces or an equivalent $begin - end$ formulation.

Note, however, that some necessary TLA specification elements must be added before the PlusCal section. First, the `---- Module name ----` line signals the beginning of the specification and matches the closing `====`. Secondly, any required external library must be included with the `EXTENDS` statement. Finally, the specification can be parameterized by declaring constants without assigning them a precise value yet using the `CONSTANTS` TLA statement. Similarly, other preliminary TLA definitions can be given in order to make the corresponding symbols available to the PlusCal specification.

Now diving in the body of a PlusCal algorithm, the usual structure for a simple sequential program is to first declare the variables along their initial value, under a `variables` section, and then the algorithm itself. However, as the subject of this Master's thesis is concurrent programs, the PlusCal `process` construct will be required. When coming across this statement, PlusCal conceptually spawns processes that will perform the job described within the `process` body concurrently. One should keep in mind that this should be understood abstractly, from the point of view of the interleaving semantics model.

The `process` construct takes an expression of the form `name \in set` as an argument, where `name` can be chosen arbitrarily but will appear in the TLA$^+$ translation while `set` can be any finite mathematical set. Basically, this statement expresses that a process should be created for each value in `set` and that they should each have a local variable named `name` holding said value. Equivalently, this value can be referred to using the reserved keyword `self` which has the benefit of being recognized in the body of macros although they are defined outside the process environment. Finally, a second `variables` statement can be used at the beginning of the `process` body to declare variables that should be private to each process. The ones previously defined outside `process` are thus shared by all processes.

With the previous remarks in mind, one is pretty much able to translate word for word every pseudo-code algorithms present in this Master's thesis to PlusCal. However, the retrospectively trickiest aspect of PlusCal has not be touched on yet, namely labeling the statements. Although labels usually provide no more than a handy reference by numbering the lines of a pseudo-code algorithm, they sometimes convey the semantics of atomicity, as in PlusCal. Simply put, all statements encountered while following the control flow of the algorithm fall under the current label until a new label is stumbled on. All instructions under a same label are then considered as a single transition and should thus correspond to an atomic step in the original algorithm. In other words, when writing a PlusCal specification, labels must be manually positioned in order to delimit the desired starting and ending points of actions. In terms of interleaving semantics, labels are the points were a process can be suspended in its job to interleave an arbitrary number of atomic steps from other processes before resuming.

As can probably already be foreseen, an atomic PlusCal step will be translated to a unique TLA action, meaning labels directly control the grain of atomicity of the final specification. Beside making sure the specification matches the behavior of the original algorithm, there are thus a few constraints to respect when placing labels. First of all, there must be a label at the beginning of the algorithm to satisfy a first great principle which is that statements should always belong to one and only one label. In line with this principle, PulsCal's label rules mostly affect constructs that mess up with the control flow as they are responsible for label obfuscation in some cases. For example, the `goto` statement unconditionally jumps to a new label, effectively "killing" the current one. As a result, the statement that would normally follow `goto` if the jump was not applied does not naturally fall under any label because it is simply not reachable in this setting. PlusCal thereby requires any statement that immediately follows `goto` to be labeled.

In a similar fashion, although the content of the branches of `if-then-(elsif)-else` statements can be labeled freely, a problem may arise if the statement immediately following the whole block is not labeled. Indeed, the branches may set up different labels on the multiple paths leading to said statement, resulting in an ambiguous label assignment. As for `goto`, any statement immediately following an `if` block must thus be labeled explicitly, as soon as any of the branches contains at least one label.

Finally, the second great principle states that a same variable should be submitted to at most a single assignment within any given label. Considering atomic steps are the smallest unit of time when reasoning about concurrent systems, it would not make sense to allow a variable to hold different values over the span of it. Moreover, multiple assignments to a same variable could not be translated to a valid TLA action anyway, as the corresponding primed variable can obviously not be mathematically equal to different values.

2.5 From PlusCal to TLA$^+$ Specifications

Once an appropriate PlusCal version of the algorithm has been written, the first step is to translate it to its TLA$^+$ equivalent. As stated in the previous section, the latter will completely replace the former in subsequent steps. From a practical point of view, the Toolbox allows to transpile a PlusCal algorithm to TLA$^+$ under the *File* tab. Being one of the most basic operations within the TLA$^+$ environment, this translation feature is also distributed by most third-party plug-ins.

Given the canonical specification formula construction process described in Section 2.2 and the description of PlusCal specification given in the previous section, the translation is rather straightforward. Essentially, the only element missing in PlusCal specifications is the auxiliary program counter variable. Fortunately, beyond the management of atomicity, the goal of the instruction labeling system of PlusCal is to unambiguously account for the control flow of the algorithm without having to introduce said variable.

The first step of a TLA$^+$ translation is thus to declare all the variables that were present in its PlusCal counterpart with the addition of `pc`, the abstract program counter. Because the TLA mathematical model does not draw any distinction between subcategories of variables, concurrent specifications that rely on process-local variables make no exception. Such variables are thus turned into a mathematical function that associates the elements of a finite domain to a value. In this case, the domain is the finite `set` given as part of the `process` statement which is referred to as `ProcSet` by default. This trick allows to define a single TLA$^+$ variable which retains the name given in the PlusCal specification while holding the values of the different instances, each corresponding to a process. Coincidentally, even if no process-private variable is used, this is exactly how the program counter variable `pc` is dealt with for multiple processes.

Then, the initial value assignment of all variables is converted to the *Init* TLA predicate with the structure $Init \triangleq var_1 = x \land var_2 = y \land \ldots$. Note that variables may instead belong to a set rather than hold a precise value or have an initial value dependent on constants parameterizing the specification. Those constructs typically allow the *Init* predicate to accept multiple initial states for the behaviors of the specification.

From there, a TLA action is defined for each PlusCal label, keeping the same name. All mutations of variable values are described using primed variables expressions and an additional *Unchanged* clause is added to explicitly indicate that remaining variables are unmodified. Among these variables, `pc` reflects the control flow of the original algorithm and takes values in the set of labels that appear in the PlusCal specification. It is thus used within an action as a postcondition indicating the destination label(s) the process may reach as a result of said action.

On the other hand, actions also allow to express conditions on the current value of variables, resulting in restrictions on which states the corresponding steps can be taken in. Typically, the `pc` variable is thereby also used to ensure actions are only available at the right time with regard to the control flow of the PlusCal specification. For example, one can consider an instruction labeled a unconditionally followed by an another one labeled b in PlusCal. The corresponding TLA$^+$ action is also called a and requires the current value of `pc` to be "a" and the next value `pc'` to be "b". This effectively guarantees that action a can not be taken unless the program is currently at label a and that the program will not reach another point than label b. As will be illustrated in Section 3.4, preconditions on other variables can be expressed as well in order to introduce additional restrictions on transitions.

As a result, most TLA actions corresponding to a PlusCal label will be similar to the following structure:

$$\begin{aligned}
action_{source} \triangleq\ & \land\ pc = label_{source}\ \land\ \underbrace{var_{foo} = x\ \land \ldots}_{additional\ precondtions} \\
& \land\ pc' = label_{destination}\ \land\ \underbrace{var'_{bar} = y\ \land \ldots}_{additional\ postcondtions} \\
& \land\ Unchanged\ \langle\langle\ variables\ unused\ in\ postcondtitions\ \rangle\rangle
\end{aligned}$$

Note that the destination label is not necessarily unique and may be a more complex formula describing how to choose between multiple label values. This is typically the case for branching instructions like `if` constructs. When dealing with concurrent specifications, the action is additionally parameterized by an element of `ProcSet` in order to distinguish the process that is taking the action. This argument is in turn used to index the right instance of local variables and particularly `pc`. The other local instances of those variables are specified as unmodified, which in the case of `pc` guarantees only one process can take an action at a time. The resulting modification of the previous formula is given below, using a special notation for pc' to maintain the same value except for p:

$$\begin{aligned}
action_{source}(p) \triangleq\ & \land\ pc[p] = label_{source}\ \land\ \ldots \\
& \land\ pc' = [pc\ EXCEPT\ ![p] = label_{destination}]\ \land\ \ldots \\
& \land\ Unchanged\ \langle\langle\ variables\ unused\ in\ postcondtitions\ \rangle\rangle
\end{aligned}$$

With a TLA$^+$ action for each PusCal label, the *Next* formula can be built by expressing the disjunction of all of them, as defined in Section 2.3. In the case of concurrent programming, the PlusCal translator calls the resulting expression by the `name` given as part of the `process` statement. This formula could indeed not be used as *Next* under the current form, the reason being that it contains the free variable previously referred to as p, standing for the process taking the action. In order to solve this, *Next* is in this case the existential quantification of the disjunction of all actions over the `ProcSet`. As a result, the *Next* formula states there exists (at least) one process among the ones of the specification for which at least one of the actions is verified. Given that each individual action has a different corresponding source label and that they all modify the value of `pc` for a single process only, *Next* in fact expresses that exactly one process takes exactly one of the actions.

Finally, the top-level *Spec* formula can be written by the PlusCal translator exactly like the one at the end of Section 2.3, including stuttering steps.

2.6 The TLC Model Checker

With the current tooling, one is able to practically express a specification for a system of interest under the form of a TLA temporal formula of actions by either writing it directly in TLA$^+$ or as a PlusCal intermediate representation. Now, other TLA$^+$ formulas can be appended at the end of the specification to express expected properties of the system. The idea is to somehow verify if any behavior σ allowed by the specification formula *Spec* indeed respects a property represented by another formula F. A first intuitive approach to doing so is to infer behaviors for which the *Spec* formula is true and to evaluate the F formula over them. If F is not true for one of the behaviors, it is clear that the specification as a whole does not respect the corresponding property and the failing behavior constitutes a perfect counterexample to prove it. This process is analogous to a programmer trying to imagine execution traces in order to uncover faulty edge cases in his or her program.

The above approach is the one behind explicit state model checking, which in the TLA$^+$ ecosystem is available through the TLC tool. As described in [11], TLC first looks at the *Init* formula and deduces the set of possible starting states for the behaviors. Those states are then explicitly added to a conceptual (multiple implementation are presented) FIFO queue *sq* of states left to explore and a fingerprint of them is added to a *seen* set. Then, TLC can iterate by picking the first state from *sq* and checking if it satisfies safety properties. The *Next* formula is subsequently examined to determine all the states that can be reached from the current one, by taking one of the available actions. If the fingerprint of one of these sates does not appear in the *seen* set, it must consequently be explored and is added to the *sq* queue. This process is in fact no more than a breadth-first search of the state space for states that would violate safety properties. The *seen* set allows to prune repeated exploration of a same state and thereby makes it possible to check infinite behaviors as long as they contain a finite number of states.

It is important to note that the previous description corresponds to the original version of TLC which can only check safety properties of specifications. The current version partially supports checking liveness properties which requires a radically different approach. It is indeed not sufficient to check every state once in order to determine if a statement will eventually be true at some point of the behavior. However, such properties will not appear in the following of this Master's thesis and the mechanisms required to check them are thus not developed.

As its name indicates, the TLC model checker does not work on complete TLA$^+$ specifications but on models derived from them. As described in [11], traditional model checking approaches rely on the assumption that the number of reachable states of their model is finite which makes it possible to explicitly enumerate and check them. On the other hand, the full expressiveness of the TLA logic implemented in TLA$^+$ does not provide any such guarantee. It is thus necessary to derive a finite model from the full specifications before being able to check it with TLC.

First, an explicit value must be assigned to any unspecified constant introduced to parameterize the specification. This is necessary for TLC to be able to instantiate the specification by giving an initial value to all symbols. Unspecified constants are therefore reported as errors, before even trying to start a TLC check. It is also possible to provide a finite set of possible values for a constant, instead of a single one, which will result in a series of consequent model checks, one for each given value.

All the possible values for a given constant are generally not equivalent when it comes to this assignment. Although they may be conceptually considered as acceptable values for the specified system to work properly, they might indeed result in drastically different state space sizes. For example, this is particularly true regarding the number of processes in a concurrent system specification. Due to the number of possible interleavings of the actions of said processes, the size of the state space often explodes very quickly when the number of processes increases. Hence, it is often possible for TLC to fully check a model within reasonable time for a few values of the constants of the specifications while other ones make the process impractical or even impossible because of memory exhaustion.

Now that a value has been assigned to all constants, it is of course still possible for a specification to result in an infinite number reachable states. This problem will notably arise in Section 3.4, when trying to check the specification of Barz's algorithm. Simply put, the specification described up to that point allows one of its variables to eventually reach any natural value. Hence, as each individual sate holds the value of said variable among others, there is necessarily an infinite number of them. Even if TLC could foresee such problems like for an explicit nondeterministic assignment "The next value of x can be any element of $\mathbb{Z}$", it could still do nothing about it without arbitrarily denaturing the specification. It is thus the specifier's responsibility to identify similar situations, to make sure they are not due to a mistake in the specification and possibly to introduce auxiliary state constraints to allow checking by TLC. The latter is often done by creating an alternative specification adding new statements on top of the original one like "The value of x can now never be greater than *threshold*". Refer to Section 3.8 for an explanation on relating two specification files with the *INSTANCE* keyword. Instead of doing so for Barz's algorithm, the specification will simply be directly patched to limit behaviors resulting in an infinite state space.

Given the two previous points, it seems obvious that TLC generally does not check the full span of the original specification for the simple reason it is not able to do so. The first constant parameterization requirement already reduces the complete abstract specification to a model that is only valid for a given set of constant values. Moreover, in the second case, the newly introduced constraints artificially cut off a possibly infinite part of the reachable state space of said model. Hence, passing a TLC model check does, in most cases, not translate to any form of guarantee on the original specification. However, the insight provided by such checks can contribute to building confidence in that there is no easily detectable fault in the specification with respect to expected properties. Successful TLC model checks thus constitute a good indication that it is worth going further and investing time in trying to formally prove the specification respects said properties. On the other hand, when a model check fails, TLC provides a behavior that leads to the violation of the problematic property. This constitutes an invaluable counterexample that precisely shows how the specification does not respect the expected property. The error trace is thus the perfect starting point for correcting either the specification or the property.

From a practical standpoint, when using the TLA$^+$ IDE, the values of constants must simply be inputted in the relative boxes after creating a new model for the specification. Any atemporal formula can be checked as a safety property under the *invariants* section. The definition of these formulas can either be given on the spot or in the specification file in which case they can handily be referred to by their name. When working in a different setting that uses TLC, the underlying configuration file automatically produced by the TLA$^+$ IDE for TLC must usually be provided manually. Note that in both environments, deadlocks are checked for by default.

2.7 TLAPS

After having successfully checked safety properties for some models, one can be hopeful that the specification always respects those properties but still has no formal proof that the assertion is true. It is thus necessary to introduce a new part of the TLA$^+$ language to allow to write proofs and to subsequently mechanically check them. This is the role of the TLA Proof System - TLAPS and its TLA Proof Manager - TLAPM. As described in [3], the benefit of this proof system is that the expression of the proof does not rely on the underlying verification tool. The user is thus able to write the proof by only knowing TLA$^+$ as developed in the previous sections as well as a few basic primitives of TLAPS. It is TLAPM's job to translate this form of proof to obligations geared towards a particular back-end solver.

First of all, TLAPS proofs follow a hierarchical structure. This implies that the original statement to prove sits at the top level. If it can not be proven as it is, simpler scaffolding statements must be provided in order to help TLAPM. The problem is that these new statements must be proven as well for the overall scheme to hold. As a result, the complete proof can be seen as a tree of statements. The leaves are "simple" enough to be proven without additional support and serve as the basis for the proofs of the statements on top of them.

For any of the statements of the tree, the objective of the user is to bring the right known facts to the table in order for TLAPM to be able to prove it. The goal statement and the associated context of known facts are referred to as an obligation. The obligation is said to be verified if the facts invoked in the context are sufficient to logically imply the goal assertion.

In practice, the statements are hierarchically organized with a numbering system. The ones right below the top-level assertion are given labels of the form $\langle 1 \rangle 1$, $\langle 1 \rangle 2$ and so on. The first number indicates the level of the statement in the tree and the second one allows to refer to the particular statement. Statements added below a level $\langle 1 \rangle$ step to justify it will thus be considered at level $\langle 2 \rangle$. The last statement at a given level should be the reserved *QED* keyword. It is used to prove that the statements developed on this level are sufficient to prove the parent assertion.

Now that the structure of a proof can be laid out, it is necessary to justify all the leaf obligations. The main way to do so is to add a BY clause right after the line containing the statement. The facts that are cited after the BY keyword must of course be verified themselves and are added to the context. There are nonetheless a few more rules to determine the facts that are available in the context at a given point of the proof. First, lower-level obligations inherit the context of their parent. Then, unnumbered statements are implicitly added to the context of following obligations on the same level, whereas numbered one must be explicitly invoked. Finally, a *USE* statement can be introduced to bring facts into context in a similar way to BY. This statement does of course not need to be verified and makes the facts available to the following statements if it is unnumbered. In a similar way, *HIDE* statements allow to remove facts from the current context, which is sometimes useful to avoid burdening TLAPM with useless facts.

Another important aspect of TLAPS is the availability of definitions. TLAPM is indeed not allowed to expand the definition of tokens appearing in facts unless explicitly asked to. This default behavior allows to limit the size of obligations by not automatically unrolling all nested definitions. The set of available definitions is thus part of the context of obligations, just like known facts. Therefore, the three BY, *USE* and *HIDE* constructs are allowed to be followed by an optional *DEF* field to respectively add or remove definitions from the current context.

2.8 TLATeX

As the creator of both TLA$^+$ and LaTeX, Leslie Lamport included a pretty-printer for TLA$^+$ based on LaTeX. It is available through the Toolbox to output a PDF document based on a whole specification module. In order to produce the figures of this Master's thesis, TLATeX however had to be fed with precise sections of Pluscal or TLA$^+$ directly, without going through the Toolbox. Basically, a standalone LaTeX document must be created and the desired "code" must be included within the `pcal` or `tla` environment respectively. The `tla2tex.TeX` program shipped as part of the `tla2tools.jar` archive of the Toolbox must then be called with the `.tex` file containing the document as an argument. As a result, the file should be replaced by a new version in which tla2tex appended a `tlatex` environment containing the LaTeX typesetting directives. In order to be able to compile this document to a PDF output, the `tlatex.sty`[3] LaTeX package must be included. When producing the output, the original `pcal` or `tla` environments is ignored. If the obtained document is to be subsequently included as part of a larger one like for this Master's thesis, one should not forget to remove the surrounding `document` environment manually.

2.9 General Methodology

This section is mostly derived from advice given by Stephan Merz, Director of Research at the INRIA Nancy research center. The goal is to present a step-by-step general methodology that has been proven useful for all the problems that will be discussed in the following chapters. It is important to note this is not the universal way the TLA$^+$ toolset should be used but rather a kind of roadmap laying out the important milestones to go through when specifying parallel programming algorithms. Hence, additional steps or tuning will be required depending on the specificities of the different algorithms that must be tackled.

First, once a satisfying PlusCal specification is written and translated to TLA$^+$, the first step is to check a few models while varying some parameters. At this stage, TLC already allows to rule out many potential problems that would be way more costly to discover further on. Typically, TLC offers to check for deadlock by default and supposed invariants or temporal properties of the algorithm can be submitted manually through the IDE or a configuration file before running a model check. Such a process may allow to find out that the specification written up to this point does not meet the

[3] Package available from: https://lamport.azurewebsites.net/tla/tlatex.sty

expected behavior of the modeled algorithm. It is also worth inspecting the TLA$^+$ translation of the PlusCal specification to be sure there is no clear inconsistencies. Moreover, as this translation is the basis all subsequent developments will be built upon, it is important to understand its intricacies before going further.

Then, with a convincing specification that passed the preliminary checks, one may start writing TLA$^+$ statements about the algorithm and theorems to state that respecting the specification guarantees they are true. The final step will be to write proofs for those theorems (or lemmas) to guide TLAPS (or rather let TLAPS guide us) in verifying them formally. The most pervasive form of statement that can be made about algorithms like the ones that will be presented is the invariant. Such a statement only concern individual states reached by the system and declares verifiable facts about the value of the variables that make up these states. In other words, the invariant expresses a propositional formula that must evaluate to $\top$ in each and every state that may be reached by following the specification from its initial state(s). The standard way to formally prove that the invariant is indeed verified by the specification is to show that the invariant is **inductive**. Practically speaking, proving an invariant as inductive simply consists in two steps, the first and usually easiest of which being to verify all initial states of the system respect the invariant. Then, for every possible transition of the system, one must prove the invariant still holds in the new state reached after the transition, assuming it held before said transition transition. Intuitively, these two steps are sufficient for the induction to unfold as expected because the invariant is known to hold for any initial state so it must also hold for any state reached from the initial state and for any state reached from this new state and so on. Hence, the truth of the invariant being verified in the initial state is able to "propagate" along all the transitions of the system in such a way that one can be sure the invariant inductively holds for any reachable state of the system. Not only do invariants translate expected facts about the studied algorithms, they also comprehensibly express constraints on the reachable state space of the system. This accounts for one of the most important lesson learned while producing this work: even though it may seem unrelated at first, having a solid invariant is usually the key to prove other properties of the algorithm as it essentially encapsulates the logic behind it.

Going back to TLA$^+$, most of the proofs presented in this Master's thesis will involve inductive invariants. Having multiple invariants rather than a single bigger one often proves helpful as it allows to express increasingly complex facts about the system while using the previously proven easier ones as "stepping stone" assumptions while trying to write a proof for the harder ones. Additionally, this approach saves time when running the TLA$^+$ proof manager as it has to work with smaller statements and does not need to verify the previously proven theorems again when they are used as assumptions in subsequent proofs.

With this strategy in mind, the first step should de facto be a form of typing invariant as TLA$^+$ is not a typed language. Such an invariant thus reduces the space of possible values for the variables that appear in the specification from pretty much anything to a determined set of values. Note that the set might be infinite such as the set of all natural numbers which would then indeed appear as a type like one would expect from a programming language. At this stage, the invariant should be as restraining as possible while keeping the proof process simple. It should contain one clause for each variable that appears in the algorithm and if, it can not be proven by itself, it is better to weaken the constraint for now rather than introducing new clauses to try to prove it. For example, many algorithms rely on a counter variable that should be a natural number at any point. However, when looking at individual steps, TLAPS is only able to ascertain that such a variable is sometimes incremented and sometimes decremented. Therefore, it is simpler to indicate in the type invariant that the counter takes its value in the whole set of integers rather than limiting it to the set of natural numbers. Indeed, in order to be proven, this stricter claim would generally require to dive in the logic of the algorithm which is the heavy-duty work one would like to avoid at this point.

Once the type invariant has been determined and ts adjoining proof has been verified, the next step is to check for parts or mechanism the algorithm relies on but which are relatively independent of the rest of the algorithm. The best example is the lock synchronization primitive that will appear in all the presented algorithms. Indeed, locks are used to protect a critical section only one process should be able to access at anytime and usually do so regardless of the logic of the algorithm. In such cases, the mutual exclusion property provided by the lock can be proven prematurely without further understanding of the whole algorithm. Moreover, said proof will have the additional benefit to be very similar from one algorithm to the other. However, it is important to note that the previous explanation does not state in anyway that it is valid for any algorithm were locks appear. Practically, it suffices to enter or exit the critical section of the lock conditionally for the mutual exclusion and the rest of the logic of the algorithm to intertwine.

Finally, once as much preliminary work as possible has been tackled, it is time to proceed with the main inductive invariant which is usually the hardest to find and to prove. Compared to the previous ones, this invariant requires some ingenuity as well as a deep understanding of the algorithm as a whole. The starting point is typically a set of properties that are obviously inferred from the expected behavior of the algorithm. As a rule of thumb, it is also a good idea to reintroduce stricter bounds on the value of variables that have been loosened up when trying to prove the type invariant, if applicable. Supposing a first candidate invariant has been settled on, it is overwhelmingly important to check it on a model. TLC is by definition unable to provide a formal guarantee that the invariant will be true or sufficient. However, if TLC finds a counterexample, under the form of an error trace leading to a state where the candidate invariant is violated, one can be sure the invariant is somehow incorrect. Moreover, the error trace typically provides the insight needed to improve the candidate invariant. Therefore, it is crucial to run TLC again every time the invariant is modified, as it allows to detect a faulty invariant as soon as it is proposed, thus saving a lot of time in the long run. Indeed, proceeding to trying to prove such a faulty invariant as inductive will result in the back-end SMT solvers failing without further indication.

Now, the schematic of a general inductive proof for the resulting invariant should be laid out. From this point on, unless said proof can be verified trivially, one should proceed with the following iterative invariant completion process. The main idea is to "unroll" the second inductive part of the proof (as the first one relative to the initial states most probably does not cause any problem) by first decomposing it in all the cases corresponding to the labels of the initial specification. Hopefully, this step can be performed semi-automatically by making use of the "Decompose Proof" feature of TLAPS (ctrl-G ctrl-D default shortcut). Now, rerunning TLAPM on the expanded proof allows to identify for which parts of the algorithm the conservation of the invariant can not be proven. Furthermore, the invariant itself can be decomposed in its different clauses for those failing cases. The objective of this whole process is to pinpoint which part of the invariant can not be proven after taking some action if it was assumed to be true before said action. The difficult part is now to look at all the assumptions available to TLAPM for proving a failing obligation and try to infer what information is missing in this situation that would allow the solver to succeed. A trivial cause for such shortcoming is simply to forget to expand the definition of a formula that is available. Although this kind of mistake can be very frustrating to spot, it is usually eliminated without further effort. On the other hand, the second category of failure has to do with the invariant itself. Indeed, the information encapsulated in the invariant, which is made available as an assumption when trying to prove its own invariance to the transitions of the system, may not be sufficient to "hold itself" at some point of the algorithm. In this case, some clauses must be modified or added to the invariant to fill the identified gaps.

Then, before starting a new round of the iterative process, one should be careful to run TLC again to check the new invariant and to rollback any previous expansion of the definition of the invariant as they have de facto become inconsistent with the updated one. Repeating the previously described process, the solver might fail to prove some obligations which will in turn raise previously unforeseen shortfalls of the invariant preventing it from being fully inductive. It is worth noting that although it may seem counterintuitive at first, some failing obligations may have been successfully proven in the previous round since the invariant was different. Hopefully, this process will eventually lead to finding a complete enough invariant to be inductive.

Chapter 3

Barz's Algorithm

3.1 Semaphores

Building upon the simple locking mechanism, it seems natural to extend the principle of exclusion to more than a single process. Typically, the resource to be shared might be simultaneously accessible by multiple processes without causing a problem up to a certain defined limit. Using a lock in such a situation obviously enforces the exclusion principle but appears like a waste of resources since any potential for parallelism is unexploitable due to the systematic sequentialization.

The usual solution to this issue is to introduce a semaphore mechanism made of a variable used as a counter and an interface made of two methods generally named `Wait()` and `Signal()`. The counter is initially set to the maximum number of processes that may safely access the shared resource simultaneously (also referred to as the **capacity** of the semaphore) and is decremented each time one of them calls the `Wait()` method. Once the counter variable reaches 0, the `Wait()` method becomes blocking, forcing the processes that try to use it to wait until "a slot" is available, meaning the counter is greater than 0. On the other hand, processes can call the `Signal()` method to increment the counter by 1, thus signaling to another process that the slot is available and that it does not need to wait anymore.

It is important to recall that those operations on the semaphore variable must be safe themselves. Therefore, practical implementations typically rely on a lock to protect the variable, ensuring a single process can access it at any time. Those implementations often include other methods in the interface such as `Get()` to read the value of the counter variable without tampering with it, `Signal(`n`)` to increment to counter by an arbitrary amount n or `Signal_all()` to free all processes that are currently waiting.

Another property that is often wished for in semaphores is the transfer of permissions. This means that, in contrast to locks, semaphores should allow any process to increment the counter, not necessarily the one that performed the "corresponding" decrement. The `Signal()` name for the method results from this property because it allows any process to signal a waiting process that it is free to go whereas locks only allow a single process to lock, perform its critical task and unlock them while the others are forced to wait.

3.2 Historical Context

Dijkstra, who introduced the concept of semaphore for the first time in [5] around 1962, later devoted a full section to the [Superfluity of the General Semaphore 4, p.35] in the relative chapter in which he states:

> In this section we shall show the superfluity of the general semaphore and we shall do so by rewriting the last program of the previous section, using binary semaphores only. (Intentionally I have written "we shall show" and not "we shall prove the superfluity". We do not have at our disposal the mathematical apparatus that would be needed to give such a proof and do not feel inclined to develop such mathematical apparatus now. Nevertheless I hope that my show will be convincing!)

In other words, he considered the need for standalone general semaphores (i.e. semaphores whose capacity may take any natural value) beside simpler binary semaphores (whose capacity is 1) as irrelevant because he was able to "show" that the former can be simulated using the latter only. As a little sneak peek, note that TLA$^+$ will provide the "mathematical apparatus" required to produce a formal proof Dijkstra "did not feel inclined to develop" at the time.

In 1983, Barz looked back at candidate solutions that had been published since Dijkstra's statement to simulate the general semaphore using binary ones only. In [1], he shows that one is straightup faulty, providing an error trace. Others, although proven valid as substitutions for the general semaphore, are deemed suboptimal compared to the final original solution he provides. This substitution that is nowadays referred to as "Barz's Algorithm", only requires two binary semaphores (or equivalently a binary semaphore and a lock).

3.3 The Algorithm

Figure 3.1 lays out both procedures that make up Barz's algorithm as presented in [2], the graphics being taken from the Parallel Programming

```
procedure delay.Wait()
    delay.Wait();
    lock.Lock();
    counter ← counter − 1;
    if counter > 0 then
        delay.Signal()
    lock.Unlock();
```

```
procedure delay.Signal()
    lock.Lock();
    counter ← counter + 1;
    if counter = 1 then
        delay.Signal()
    lock.Unlock();
```

Figure 3.1: Barz's algorithm, taken from [2]

course's slides [6]. As already hinted above, one can notice the algorithm relies on three variables called delay, lock and counter. The two first ones are the two binary semaphores required by Barz's solution who presented it as "optimal with respect to the number of additional semaphores and variables" [1]. It can however be noticed that the lock variable is used with the methods suiting a lock rather than a semaphore like delay. This can be explained by the fact that the transfer of permissions which is the characteristic feature that sets binary semaphores apart from locks is in fact needed for only one of the semaphores. Barz's algorithm can thereby indeed be expressed equivalently using a lock and a single binary semaphore since the purpose of the second semaphore is to simulate a lock anyway.

When it comes to the role of the lock variable, it is to guarantee that the increment or decrement operations on the counter and the following comparison are atomic in both methods. Intuitively, the intent is to initially set counter as the desired capacity N greater than 0 for the general semaphore and to increment and decrement it through `Signal()` and `Wait()` calls respectively in such a way that it remains in the $[0, N]$ interval. However, although counter always being greater than 0 might appear as a trivial, this property will still need to be proven formally with the help of TLAPS. On the other hand, one can already point out at this stage that there is no guarantee at all when it comes to counter not exceeding the initial capacity N. Indeed, no assumptions have been made yet on how processes will behave outside of the body of the `Wait()` and `Signal()` methods. There is thus no mechanism preventing a process from calling `Signal()` when the value of counter is already N, getting it past the initial capacity.

Finally, the "trick" behind Barz's algorithm is to harness the transfer of permissions provided by the delay binary semaphore while coupling it to the counter variable to compensate for its limited capacity. This idea very roughly boils down to "if counter is greater than 0 the value of delay should be 1 so that waiting processes are not blocked on the first delay.`Wait()` instruction" and, conversely, "if counter reaches 0, the value of delay should be 0 in order to block waiting processes". Hence, a process is responsible for calling `delay.Signal()` to set `delay` back to 1 after incrementing `counter` from 1 to 0 in `Signal()` or after entering `Wait()` without decrementing `counter` down to 0.

3.4 TLA$^+$ Specification

As can be seen in Figure 3.2, the PlusCal specification is very close to the original formulation of Barz's algorithm 3.1. The two procedures `Wait()` and `Signal()` are clearly identifiable in the two branches of the **either-or** statement. The label names have been chosen to reflect this, with w and s prefixes for `Wait()` and `Signal()` respectively and the numbers roughly matching those of the pseudo-code lines while respecting the constraints of PlusCal. The whole body of the process is shrouded in a **while** statement which represents the processes potentially indefinitely looping performing an arbitrary job (**skip**) until they need synchronization through either of the methods of the semaphore. It is crucial to restate that, as TLA$^+$ is a specification language rather than a programming one, the choice introduced by the **either-or** statement is not solved by an arbitrary coin flip each time it arises. Instead, the `b1` label gets two non-deterministic outgoing transitions towards the $w2$ and $s2$ labels, meaning that the two scenarios are always possible (as long as corresponding transitions are not disabled, see later) and must be considered.

While the preceding explanation lays out the general idea behind **either-or** statements, it must be taken with a grain of salt in this case. As already explained when presenting the algorithm, there is nothing preventing a process from calling `Signal()` repetitively. Keeping `counter` in the $[0, N]$ interval thus never exceeding the initial capacity of the general semaphore might be a desired property. However, the elephant in the room is the resulting state space which is necessarily infinite thus preventing exhaustive state space exploration by the TLC model checker. Indeed, any reachable configuration of the system may also be reached with any positive value for `counter` as an arbitrary number of calls to `Signal()` can be inserted in the execution trace unconditionally. Therefore, the decision was made to prevent the processes from calling `Signal()` if `counter` already holds the value N equal to the initial capacity.

In TLA$^+$, such a restriction is specified through the **with** or **await** keywords which are roughly equivalent for this use case. As **with** is a more powerful keyword with multiple meanings, **await** has been used event though it might be misleading coming from a programming background. Indeed, whereas **await** usually means "wait until the given condition becomes true", TLA$^+$ considers it as a condition under which a possible transition is enabled.

```
CONSTANTS
    $P$,  number of processes
    $N$   initial capacity of general semaphore

--algorithm Barz{
    variables
        $delay = 1$,
        $lock = 1$,
        $counter = N$ ;

     Request a binary semaphore.
    macro wait(s){
        await $s = 1$ ;
        $s := 0$ ;
    }

     Release a binary semaphore.
    macro signal(s){
        $s := 1$ ;
    }

    process $(proc \in 1 \,..\, P)${
b0:   while (TRUE){
        skip ;    do whatever
b1:     either {
            wait(delay) ;
w2:         wait(lock) ;
w3:         $counter := counter - 1$ ;
w4:         if $(counter > 0)${
w5:           signal(delay)
            } ;
w6:         signal(lock) ;
        } or {
            await $counter < N$ ;
            wait(lock) ;
s2:         $counter := counter + 1$ ;
s3:         if $(counter = 1)${
s4:           signal(delay)
            } ;
s5:         signal(lock) ;
        }
      }
    }
}
```

Figure 3.2: PlusCal Specification for Barz's Algorithm

In other words, **await** effectively has the power to block a transition if its condition is not met. Hence, if a process is chosen to take an action while all its outgoing transitions are deactivated, it has no other choice than taking a stuttering step until other processes hopefully change the system to a state where the condition of one of the blocking **await** is met. Such a behavior is akin to what one might expect from regular programming languages. However, even if an outgoing transition is deactivated, a state may have one or multiple other ones that are still active. In this case, the process is free to take one of these transitions (or it can always choose to stutter) thus removing the notion of "waiting" completely.

Going back to the specification of Barz's algorithm, there are two uses of **await**. The first one is hidden in the body of the $wait(s)$ macro that specifies the corresponding method on binary semaphores, namely delay and lock. It expresses the fact that a process should not be able to acquire the binary semaphore s if it is not available in the first place. In terms of value of s, 1 means the semaphore is free to acquire and the process who does so accordingly sets s to 0 with $wait(s)$. PlusCal macros can not define new labels which means that their content always specifies an atomic operation that falls under the current label, at the line when they are called. Such a behavior is of course desired in this case because processes should not be able to check the value of s and then stop at an intermediary label before setting it to 0 as this pattern obviously suffers from a potential race condition.

The second **await** statement can be found at the very beginning of the second branch of the **either-or** block. It is not present in the original pseudo-code formulation of the algorithm and has deliberately been added to solve the issue with `Signal()` described at the beginning of this section. Indeed, there is no label between $b1$ and $s2$, meaning that a process nondeterministically branching at the **either** statement of $b1$ can only jump to $s2$ if the $counter < N$ condition of **await** is met. Such a behavior effectively corresponds to the original intent of preventing processes to call `Signal()` on the simulated general semaphore if counter's value has already reached N.

Finally, another slight difference that might be noted between the original algorithm and this specification is the that lock is considered as a binary semaphore rather than a proper lock. This reflects the comment made previously on the equivalence of lock being a binary semaphore or a lock while having the benefit of defining a single pair of macros common to delay and lock rather than two separate ones. Furthermore, the definitions of those macros would be the exact same for a lock anyway and the proper mutual exclusion property of lock will be proven in the next section.

3.5 Inductive Invariants

The invariants that have been found and proven inductive for Barz's algorithm are presented in Figure 3.3. In order to obtain those results, the approach proposed in Section 2.9 has been applied in an almost unaltered way.

First of all, the typing invariant is rather straightforward beside the fact that it states `counter` belongs to the set of all integers $\mathbb{Z}$. Intuitively, `counter` should indeed remain in the more restrictive set of naturals $\mathbb{N}$, and even more precisely in the $[0, N]$ interval. As already hinted previously, this is due to the impossibility of proving such a statement at this stage, without diving headfirst in the complexity of the algorithm. Indeed, when trying to verify those stricter invariants as valid, the TLAPM solvers fail to prove the inductive step. Therefore and to keep the typing invariant simple, one should fall back to the looser domain $\mathbb{Z}$. On the other hand, it is trivial for TLAPM to prove that `counter` and `delay` are indeed binary as they are never assigned a value different from 0 or 1.

Then, the second inductive invariant *LockInv* expresses the mutual exclusion property implemented by the use of *lock*. The associated critical section *lockCS* spans from $w3$ to $w6$ and from $s2$ to $s5$, covering both branches of the algorithm. Indeed, after a process has acquired *lock* through $wait(lock)$, the other ones should not be able to enter the protected section until *lock* has been released by $signal(lock)$. Hence, the first clause of the second invariant states that there can not be two different processes in *lockCS* at any time. However, such a clause is not sufficient to hold inductively by itself. The second one has thereby been added to express the link between mutual exclusion and the value of the *lock* variable. Simply put, if there is a process in *lockCS*, this clause ensures $lock = 0$, which in turn guarantees that no other process can go past $wait(lock)$. The resulting invariant is sufficient to be proven inductive.

Finally, the main invariant of Barz's algorithm was fully uncovered after multiple rounds and backtrackings of the iterative procedure explained in Section 2.9. The initial intent was to express the mutual exclusion property due to `delay` in the same way as the one due to `lock`, which explains the similarity of the first two clauses to the locking invariant.

As can be noticed in the PlusCal specification 3.2, any process choosing to `Wait()` (i.e. transitioning to label $w2$ rather than $s2$) at the **either** branching corresponding to label $b1$ must atomically acquire the `delay` semaphore. Intuitively, this semaphore may only be freed by $signal(delay)$ at label $w5$, thus making the $w2$ to $w5$ *delayCS* section "critical" as only one process can proceed at a time. The other $signal(delay)$ statement at $s4$ should nonetheless not be disregarded, as it may seem to be able to break the de-

$$
\begin{array}{l}
\mathit{TypeInv} \;\triangleq \\
\quad \land\ \mathit{delay} \in \{0, 1\} \\
\quad \land\ \mathit{lock} \in \{0, 1\} \\
\quad \land\ \mathit{counter} \in \mathit{Int} \\
\quad \land\ \mathit{pc} \in [\mathit{ProcSet} \rightarrow \{\text{“b0”}, \text{“b1”}, \text{“w2”}, \text{“w3”}, \\
\qquad\qquad \text{“w4”}, \text{“w5”}, \text{“w6”}, \text{“s1”}, \text{“s2”}, \text{“s3”}, \text{“s4”}, \text{“s5”}\}]
\end{array}
$$

$$
\begin{array}{l}
\mathit{lockCS}(p) \;\triangleq \\
\quad \mathit{pc}[p] \in \{\text{“w3”}, \text{“w4”}, \text{“w5”}, \text{“w6”}, \text{“s2”}, \text{“s3”}, \text{“s4”}, \text{“s5”}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{LockInv} \;\triangleq \\
\quad \land\ \forall\, i, j \in \mathit{ProcSet} : (i \neq j) \Rightarrow \neg(\mathit{lockCS}(i) \land \mathit{lockCS}(j)) \\
\quad \land\ (\exists\, p \in \mathit{ProcSet} : \mathit{lockCS}(p)) \Rightarrow \mathit{lock} = 0
\end{array}
$$

$$
\begin{array}{l}
\mathit{delayCS}(p) \;\triangleq \\
\quad \mathit{pc}[p] \in \{\text{“w2”}, \text{“w3”}, \text{“w4”}, \text{“w5”}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{Inv} \;\triangleq \\
\quad \land\ \forall\, i, j \in \mathit{ProcSet} : (i \neq j) \Rightarrow \neg(\mathit{delayCS}(i) \land \mathit{delayCS}(j)) \\
\quad \land\ (\exists\, p \in \mathit{ProcSet} : \mathit{delayCS}(p)) \Rightarrow \mathit{delay} = 0 \\
\quad \land\ \mathit{counter} \in 0 \,..\, N \\
\quad \land\ \ \lor\ \mathit{counter} = 0 \\
\qquad\ \ \lor\ \ \land\ \mathit{counter} = 1 \\
\qquad\qquad\ \land\ \exists\, p \in \mathit{ProcSet} : \mathit{pc}[p] \in \{\text{“s3”}, \text{“s4”}\} \\
\qquad\ \ \Rightarrow \mathit{delay} = 0 \\
\quad \land\ (\exists\, p \in \mathit{ProcSet} : \mathit{pc}[p] \in \{\text{“w2”}, \text{“w3”}, \text{“w5”}, \text{“s3”}, \text{“s4”}, \text{“s5”}\}) \\
\qquad \Rightarrow \mathit{counter} > 0 \\
\quad \land\ (\exists\, p \in \mathit{ProcSet} : \mathit{pc}[p] \in \{\text{“w4”}, \text{“w5”}, \text{“w6”}, \text{“s2”}\}) \\
\qquad \Rightarrow \mathit{counter} < N \\
\quad \land\ (\exists\, p \in \mathit{ProcSet} : \mathit{pc}[p] = \text{“s4”}) \\
\qquad \Rightarrow \mathit{counter} = 1 \\
\quad \land\ (\exists\, p, q \in \mathit{ProcSet} : \mathit{pc}[p] \in \{\text{“s3”}, \text{“s4”}\} \land \mathit{pc}[q] = \text{“w2”}) \\
\qquad \Rightarrow \mathit{counter} > 1
\end{array}
$$

Figure 3.3: TLA$^+$ Invariants and Adjoining Definitions for Barz's Algorithm

scribed mutual exclusion through transfer of permissions. However, one can be convinced this is in fact impossible because lock prevents a process from reaching $s4$ while another one is in *delayCS* unless said second process is exactly at $w2$. In that case, *delay* must have been free for this process to acquire in order to reach $w2$ from $b1$, indicating *counter* should have been greater than 0. Therefore and because no other process than the first one can alter *counter*, its value is necessarily incremented past 1 at label $s2$ which prevents reaching $s4$. This argument will appear again later, when discussing the last clause of the invariant.

Practically speaking, when decomposing the proof, TLAPM is able to verify the invariance for all steps expect $s4$ because the solver sees *delay* being set to 1 with no way to be sure there is no process in *delayCS*, thus possibly violating the second clause of the invariant. Retrospectively, the preceding argument constitutes a crystal-clear indication that the invariant is not strong enough as it is and must incorporate reasoning linking the values of *counter*, *delay* and the program counters together.

Consequently, the third and fourth clauses were added to the invariant. The third one is obviously derived from the failed typing invariant attempt and expresses the strictest possible bounds on the value of *counter*. When it comes to the fourth one, its purpose is to express the relationship between the values of *counter* and *delay* also roughly described as the "trick" of Barz's algorithm at the end of Section 3.3. The main intent behind this can be summarized by saying *delay* is responsible for blocking processes that try to call `Wait()` when *counter*'s value is 0 and its value must thus reflect this. Basically, $counter = 0 \Rightarrow delay = 0$ should always hold because delay is always set to 0 in between $b1$ and $w2$ before *counter* could even have a chance to be decremented to 0 at $w3$. In that case, *delay* is not reset thanks to the condition at $w4$. Additionally, there exists a transition phase during which *counter*'s value is 1 but *delay* is still unset with value 0. Such a phase occurs when a process has just incremented *counter* from 0 to 1 at $s2$ but not yet reached $s4$ to reset *delay*. Both cases are covered by the fourth clause of the invariant.

Since this invariant is still not strong enough to be inductive, three additional clauses were introduced. A simple reason to this is that although the third clause could previously not be proven within the typing invariant, the current material does not really provide any more useful assertions on the value of *counter*. The issue is that, when looking individually at steps $w3$ and $s2$, this clause can not hold. Assuming the invariant is true for the current state, one indeed knows $counter \in [0, N]$ but sees the value unconditionally decremented or incremented respectively and can thereby not be sure the assertion will continue to hold in this new state.

The new clauses thus express additional constraints on the value of *counter* resulting from the conditions of the **await** statements combined with mutual exclusion. For example when a process goes down the `Wait()` branch, it acquires *delay*, preventing other processes to do the same. Doing so implies that the value of *delay* was 1. Yet, by the contrapositive of the fourth clause of the invariant, one can sate $delay \neq 0 \Rightarrow counter \neq 0$. To sum things up, once any process reaches $w2$, it is the only one to be able to decrement *counter* whose value can not be null, which combined with the third clause corresponds to $counter \in \,]0, N]$.

The above assertion still holds when the process transitions to $w3$ at which point it is the only one to be able to modify *counter* at all, through full mutual exclusion due to *lock*. When consequently going on to $w4$, the process decrements *counter* by 1, trivially turning the constraint into $counter \in \,[0, N[$. To reach $w5$ the $counter > 0$ condition guarding the **if** statement must obviously be met as well, resulting in $counter \in \,]0, N[$. On the other hand, it can not be assumed for $w6$ as both execution paths are possible, leaving the constraint from $w4$ untouched.

Now applying the same form of reasoning to the `Signal()` branch, *lock* is already acquired and **await** enforces $counter < N$ when any process reaches $s2$. Incrementing *counter* by 1 results in the $counter \in \,]0, N]$ assertion when the $s3$ label is reached. This statement can be propagated to $s5$ without modification but $s4$ benefits from the stricter $counter = 1$ guarantee provided by the **if** guard.

All the previous constraints have been directly expressed as the fifth, sixth and seventh clauses of the invariant. Note however that similar arguments can not be formulated for the $b0$ and $b1$ labels because they are obviously not covered by mutual exclusion. Knowing a process is at one of those labels does thus not provide any more information than the general statement $counter \in \,[0, N]$.

Once more, as can be inferred from the eight clause that has not been discussed yet, the current invariant is still not sufficient. By decomposing the proof, TLAPS indicates only the second clause of the invariant can not be proven inductive when taking the $s4$ step. There is indeed a very precise state accepted by the current invariant that prevents it from holding over transition $s4$ although the states is in fact not reachable through the specification. Quite ironically, when roughly describing how Barz's algorithm could effectively block processes with `Wait()` like a general semaphore at the beginning of this section, this exact problem with $s4$ had been foreseen. Looking back at it, it is obvious that the argument formulated at that stage to show $s4$ was not violating the exclusion over the *delayCS* should have been incorporated in the invariant at some point. Fortunately, as all clauses are required anyway, the order is which they were added does not matter beyond this slight inconvenience. This issue nevertheless highlights the importance of decomposing obligations or use the TLC model checker to find error traces that violate presupposed properties in order to try to pin down remaining edge cases preventing TLAPS from proving the overall property.

The starting point of the problematic scenario lies in a deceiving loophole with the previous reasoning about the value of *counter*. Although a process, when at label $w2$, was said to be the only one able to *decrement counter*, it is not the only one to be able to *modify counter*. It is indeed possible for a process to reach $w2$, acquiring *delay* but not *lock* yet while another one is in the `Signal()` branch. Note that the order in which those events occur does not matter but the important point is that it is possible to encounter a state in which the values of *pc* for two of the P processes are $w2$ and $s2$ respectively. At this point, because there exists a process at label $w2$, *counter* is known to be strictly greater than 0 thanks to the fifth clause of the invariant.

Now, if the second process takes the $s2$ step to reach label $s3$, *counter* will be incremented and thus strictly greater than 1. In this situation, the current invariant would however only indicate that *counter* is greater than 0. The facts that there exist both a process in $w2$ and a process in $s3$ can indeed not been combined to "cumulatively" deduct $counter > 1$ from the fifth clause. The invariant being too loose on this point allows the state in which $counter = 1$ while two processes are in $w2$ and $s3$ to slip trough the cracks and be included among the set of satisfying states.

Taking the next step $s3$, the process branches to $s4$ if $counter = 1$. This condition can not be met by following the specification as explained with the previous argument but verifies the current invariant regardless. From the point of view of TLAPS trying to prove the obligation, the only thing that matters is proving that the invariant is inductive over any action permitted by the specification. Hence, when dealing with step $s4$, TLAPS does the same as for any other action, by assuming the invariant is true in the current state and verifying if it holds up when taking the step. When doing so, the second clause of the invariant states $delay = 0$ in the current state because the process at $w2$ is within the *delayCS* critical section. As this process is obviously not the one taking the action in this case, $delay'$ must still be equal to 0 in the next state for the invariant to hold. However, step $s4$ explicitly sets $delay'$ to 1, de facto invalidating the invariant in the next state.

As a result, the eighth and final clause has been appended to the invariant to make it inductive. It simply strengthens the constraints on the value of *counter* by fixing the loophole described hereabove. In order to understand how this allows TLAPS to prove the remaining obligation, it is important to state that the wording *"assuming the invariant is true in the current state"* is extremely misleading. As anything else in TLA$^+$, this process to prove induction is nothing more than a formula, namely $Inv \land \cdots \Rightarrow Inv'$. This formula relies on the logical implication connector $\Rightarrow$ which does in fact not require the left-hand side to be true for the overall formula to be evaluated as true. As often colloquially put, falsehood indeed implies anything.

In the case where two processes are at the $w2$ and $s4$ labels respectively, the invariant says $counter = 1$ because there is a process at $s4$ but the new clause also states $counter > 1$ which is an obvious contradiction. The overall invariant is consequently evaluated to $\bot$ for the current state, right before taking the previously problematic $s4$ step. In other words, the problem has been eliminated because the invariant must only hold in the next state for the cases in which it holds in the current state in order to be globally proven as inductive.

Considering the case in which the two processes are at $w2$ and $s4$ is nonetheless not sufficient. Only introducing the corresponding clause effectively transfers the burden of proof onto $s3$ as TLAPS now fails over this step although it did not cause problem beforehand. The $s3$ action is indeed the only one that leads to the $s4$ label and is thus now responsible for ensuring $counter > 1$ in the case there is another process at $w2$. However, the increment of *counter* that guarantees this statement has already happened at the preceding step $s2$. It is thus necessary to extend the clause to $counter > 1$ if there is a process at $w2$ and another one at either $s3$ or $s4$. This statement can be proven inductive over $s2$ because $counter > 0$ is already ensured by the fact there is a process at $w2$ and TLAPS can observe *counter* getting incremented by 1 while the *pc* value of the process is modified from $s2$ to $s3$. The previous problem over the $s3$ step is trivially solved because $counter > 1$ can be assumed in the current state thanks to the modified invariant and *counter* being unchanged by the action.

As a conclusion to this section, an inductive invariant has been found and verified for the specification of Barz's algorithm. Note that the complete invariant is in fact the conjunction of the typing, mutual exclusion and "final" invariants because all the clauses from these formulas were needed for the last proof to hold. It is also important to keep in mind that the process required to produce this inductive invariant has been very far from being as smooth as it may seem from the above description. Such exercise indeed requires going back and forth many times between the definition of the invariant, the model checker and the proof in order to find and fix logical fallacies that may be due to a very precise and unforeseen behavior of the algorithm.

3.6 Introduction to Refinement

The inductive invariant resulting from the previous section is a wonderful tool that characterizes each and every state reachable by respecting the specification and thus conveys the inner workings of Barz's algorithm. Nevertheless, if one goes back to the introduction of this part, in Section 3.2, the defining claim originally introduced by Dijkstra in [4], that in turn motivated Barz, was the superfluity of the general semaphore. In other words, the main goal of the specification was to prove that Barz's algorithm is indeed a valid substitution to simulate the general semaphore, only using binary ones. While the inductive invariant will come in handy to do so, it does not express anything about the relationship of Barz's algorithm to the expected behavior of the general semaphore, the reason being that said behavior has not even been defined yet.

In order to achieve the aforementioned objective, a particular type of specification and proof referred to as *refinement* will be used. As described in [10], the first step will be to write a second *abstract specification* that expresses the intended behavior of the general semaphore regardless of any implementation. Then, a *refinement mapping* will be defined to map the state space resulting from the Barz specification onto the one of this new abstract specification. The mapping will thus take the form of a function expressing how to derive the values of the variables of the abstract specifications from the values of the variables appearing in Barz's algorithm.

With such tooling available, for each action described by the Barz specification, it will be possible to translate the origin and destinations states of the transition to their equivalent in the abstract state space and to subsequently check if the resulting transition is permitted by the abstract specification. Provided that the initial states of Barz are all mapped to initial states of the abstract specification, it becomes clear that any possible execution trace of Barz's algorithm would correspond to a valid abstract alter ego.

Put differently, this means that one would be guaranteed to emulate the abstract behavior of the general semaphore faithfully by running Barz's algorithm and converting the sequence of states thanks to the refinement mapping. As can be noticed, the previous sentence exactly describes the objective defined earlier. Formally, such a scheme is described as the Barz specification *implementing* the abstract one *under the refinement mapping.*

3.7 Abstract Specification for the General Semaphore

As laid out in Figure 3.4, the proposed abstract specification for the general semaphore is extremely similar to the one of Barz's algorithm in Figure 3.2. Essentially, the only difference from the latter one is the absence of the *delay* and *lock* variables which consequently cuts out most of the lines of Barz's algorithm as they rely on said missing variables. The only instructions left thereby constitute the outer logic of the algorithm (*while* loop and *either−or* branching) or have to do with *counter* variable, which at this stage is totally

```
CONSTANTS
    P,      number of processes
    N       initial capacity of general semaphore

--algorithm semaphore {
    variable counter = N ;

    process (proc ∈ 1 .. P){
s0:   while (TRUE){
        skip ;          do whatever
s1:     either {        wait on semaphore
          await counter > 0 ;
          counter := counter − 1 ;
        } or {          signal on semaphore
          await counter < N ;
          counter := counter + 1 ;
        }
      }
    }
}
```

Figure 3.4: PlusCal Abstract Specification for the General Semaphore

independent of the one of Barz's algorithm despite having the same name.

It goes without saying that *counter* can not be removed in the same way as *delay* and *lock* when abstracting the specification. Indeed, the latter two variables result from a choice of implementation taken by Barz whereas the former is inherent to the concept of a general semaphore as a count of the number of available resources must necessarily be maintained at all time.

It is also important to note the absence of intermediary labels which results from the idea that the operations on the abstract general semaphore should appear as atomic. There are thus only two possible steps for a process to take in this specification, either it transitions from label $s0$ to $s1$, performing an arbitrary job that does not rely on the general semaphore or from $s1$ to $s0$ executing one of the two methods on said semaphore atomically.

The above scheme might appear as any given process having to respect strict alternation between the two labels when taking a step. However, it is crucial to recall the TLA logic always allows a process to take a stuttering step, leaving every single variable untouched, including the program counter. Therefore, it will be possible to take the *Next* action of the abstract specification without having to flip the label of the active process from $s0$ to $s1$ or conversely. The previous reasoning may seem irrelevant for the time being but will come in handy down the line, when trying to relate the two specifications.

There is however an additional step compared to the Barz specification, in the form of **await** *counter* > 0 at the beginning of the `Wait()` branch. Obviously, this statement is required to prevent a process from proceeding with the `Wait()` method while there is no resource available. It is nonetheless interesting to note such restriction was not explicitly needed in the Barz specification thanks to the coupling of *delay* and *counter*. Indeed, as stated by the fourth clause of the main inductive invariant in Figure 3.3, *delay* is necessarily set to 0 if the value of *counter* reaches 0, thus forbidding the entry to the `Wait()` branch thanks to the **await** statement hidden in the $wait(s)$ macro.

3.8 Refinement Mapping

Up to this point and although sharing rather obvious conceptual similarities, the two specification are totally independent from TLA$^+$'s point of view for the simple reason they constitute two modules that are not related in any way. A new operator of TLA$^+$ must thus be introduced, namely *INSTANCE* optionally followed by a *WITH* clause to express *partial parametrization.* In order for the following to work, the files containing the two specification modules must be under a common directory (or globally available through the defined TLA$^+$ library path).

Contrary to the *EXTENDS* operator used at the beginning of specifications to include standard libraries, *INSTANCE* makes the content of the imported module available under a separate namespace. Referencing a statement defined in the included module thereby requires a namespace lookup, expressed under the form *SpecName*!*ObjectName*, using the ! operator.

Then, if the imported module contains constant definitions, the value to attribute to said constant must be specified after the *WITH* parameterization keyword. Fortunately, this is not needed for the general semaphore specification as TLA$^+$ implicitly ports the value of constants over as long as they are given the same name in both modules.

In the same fashion, the variables that appear in the imported module can be parameterized. Rather than a one-time assignation like constants, the "value" that is given to a variable is a textual substitution that essentially works like a TLA$^+$ assignment whose definition can be extended to compute the value of said variable at any time. This is the key feature that allows to achieve the refinement mapping as described in Section 3.6.

The goal is thus now to find a relationship expressing how to compute the values of the variables of the abstract specification from the ones of Barz's algorithm. As one may remember, there is only one variable explicitly defined in the PlusCal abstract specification that appears in Figure 3.2, namely *counter*. Without originality, this variable will simply be set to mimic the value of the *counter* of Barz's algorithm at all times. Both variables indeed play the same role in their respective specification and are used in the same way, being incremented and decremented under the same conditions.

There is nonetheless a second variable *pc* representing the program counters of all the processes that should not be forgotten as it is implicitly added when translating the PlusCal algorithm to TLA$^+$. This *pc* "translation" constitutes the tricky part of this process because one has to map all the labels of the Barz specification onto the only $s0$ and $s1$ of the abstract one in a way that works out. To understand the choice of mapping, it may be useful to have a look at the TLA$^+$ translation of the abstract specification. Simply put, taking the $s0$ action only alters the program counter of the corresponding process whereas taking $s1$ modifies both *counter* and *pc*.

The main constraint to notice at this stage are the two actions $w3$ and $s2$ that modify the value of *counter* in the Barz specification. Indeed, the label mapping must be chosen to always match those actions to $s1$ in the abstract realm. Otherwise, a valid behavior of the Barz specification could take an action modifying its *counter* variable which would not satisfy the $s1$ action as the label transition from $s1$ to $s0$ for the active process would not be respected. Any other available action, namely $s0$ or a stuttering step, do not allow to modify *counter*, which is the same as the one from Barz's algorithm, and would thus be eliminated. As a result, all the possible actions would be depleted, meaning the *Next* disjunction of said actions would be false as well, leading to the abstract *Spec* being violated. The conclusion to this argument is that the labels corresponding to $w3$ and $w4$ in the refinement mapping must be $s1$ and $s0$ respectively, in order to ensure the $s1$ action is verified in the abstract realm when taking the $w3$ action in Barz's algorithm. Same goes for the $s2$ and $s3$ labels relative to the $s2$ action.

Regarding the rest of the refinement mapping, it results from the previous reasoning that no other transition from abstract labels $s1$ to $s0$ can occur while following the control flow of Barz's algorithm. Beside this, there is no particular constraint because all the remaining actions appearing in Barz's algorithm only modify the value of *delay* and *lock*. Incidentally, such variables, that have no impact on the abstract behavior of the specification, are referred to as internal variables by Lamport and Merz in [10].

Consequently, once either of the $w3$ or $s2$ actions has set the abstract program counter of a process to $s0$, the label mapping must ensure this program counter is flipped back once and only once to $s1$ before the process reaches one of the those actions again. Intuitively, this makes sense as going through one iteration of the while loop in the Barz algorithm specification should result in exactly one iteration of the abstract loop. The switching point from

```
apc(p) ≜
    IF p ∈ {"b1", "w2", "w3", "s2"}
     THEN "c1"
     ELSE "c0"

Sema ≜
    INSTANCE Semaphore
    WITH counter ← counter,
            pc ← [p ∈ ProcSet ↦ apc(pc[p])]

ASpec ≜ Sema!Spec

THEOREM Spec ⇒ ASpec
```

Figure 3.5: TLA$^+$ Refinement Mapping and Definitions for Barz's Algorithm

label $s0$ to $s1$ has thus been chosen as the $b0$ action of the Barz specification.

Finally, the remaining label mappings will simply be assigned in such a way as to maintain a constant *pc* value in between the predefined flipping actions. As already anticipated in Section 3.7, it may appear as these actions taken in Barz's algorithm are somehow "lost" as they do not result in any change from the abstract point of view. However, they do not cause any problem to TLA$^+$ as the abstract specification provides for such stuttering steps. On the intuition side, one should simply think about it as setup operations on the internal variables. In the end, it is not that surprising that an implementation requires more steps than the abstract description of the behavior it is trying to implement. Fixing mismatches in the behaviors of abstract specification and their implementation by the way constitutes the original reason presented by Lamport in [9, p. 10] in order to motivate the addition of stuttering steps in his logic.

Practically speaking, the TLA$^+$ expression of the preceding refinement mapping as well as adjoining definitions are given in Figure 3.5. First, the $apc(p)$ operator, whose name stands for *"abstract program counter"* defines the label mapping as expressed above, $b1$, $w3$ and $s2$ being the destination or origin labels of the flipping actions $b0(p)$, $w3(p)$ and $s2(p)$ respectively. The remaining $w2$ simply is an intermediary state between $b1$ and $w3$. Then, the *Sema* instance of the abstract specification is defined and parameterized so that the abstract *counter* is the one from Barz's algorithm and the abstract program counter array *pc* is obtained by applying the $apc(p)$ operator to the program counter label of every process.

```
THEOREM $Spec \Rightarrow ASpec$
  ⟨1⟩ USE  DEF ASpec, apc, ProcSet, Sema!ProcSet
  ⟨1⟩1 $Init \Rightarrow Sema!Init$
    BY  DEF Init, Sema!Init
  ⟨1⟩2 $[Next]_{vars} \land Inv \land TypeInv \Rightarrow [Sema!Next]_{Sema}!vars$
    ⟨2⟩1 $Next \land Inv \land TypeInv \Rightarrow [Sema!Next]_{Sema}!vars$
        ⟨3⟩ USE  DEF Sema!proc, apc
        ⟨3⟩ SUFFICES ASSUME Inv, TypeInv, Next, NEW $p \in ProcSet$, proc(p)
        PROVE  $[Sema!Next]_{Sema}!vars$
        BY  DEF Next
        ⟨3⟩1.CASE b0(p)
            BY ⟨3⟩1  DEF b0, Sema!Next, Sema!c0, TypeInv
        ⟨3⟩2.CASE b1(p)
            BY ⟨3⟩2  DEF b1, Sema!vars, TypeInv
        ⟨3⟩3.CASE w2(p)
            BY ⟨3⟩3  DEF w2, Sema!vars
        ⟨3⟩4.CASE w3(p)
            BY ⟨3⟩4  DEF w3, Sema!Next, Sema!c1, Inv, TypeInv
        ⟨3⟩5.CASE w4(p)
            BY ⟨3⟩5  DEF w4, Sema!vars, TypeInv
        ⟨3⟩6.CASE w5(p)
            BY ⟨3⟩6  DEF w5, Sema!vars
        ⟨3⟩7.CASE w6(p)
            BY ⟨3⟩7  DEF w6, Sema!vars
        ⟨3⟩8.CASE s2(p)
            BY ⟨3⟩8  DEF s2, Sema!Next, Sema!c1, Inv, TypeInv
        ⟨3⟩9.CASE s3(p)
            BY ⟨3⟩9  DEF s3, Sema!vars, TypeInv
        ⟨3⟩10.CASE s4(p)
            BY ⟨3⟩10  DEF s4, Sema!vars
        ⟨3⟩11.CASE s5(p)
            BY ⟨3⟩11  DEF s5, Sema!vars
        ⟨3⟩13. QED
            BY ⟨3⟩1, ⟨3⟩2, ⟨3⟩3, ⟨3⟩4, ⟨3⟩5, ⟨3⟩6, ⟨3⟩7, ⟨3⟩8, ⟨3⟩9, ⟨3⟩10, ⟨3⟩11  DEF Next, proc
    ⟨2⟩2 UNCHANGED vars $\Rightarrow$ UNCHANGED Sema!vars
        BY  DEF vars, Sema!vars
    ⟨2⟩ QED
    BY ⟨2⟩1, ⟨2⟩2
  ⟨1⟩ QED
    BY ⟨1⟩1, ⟨1⟩2, InductiveInvariant, Typing, PTL DEF Spec, Sema!Spec
```

Figure 3.6: TLAPS Refinement Proof for Barz's Algorithm

3.9 Refinement Proof

Regarding the TLAPS proof, as can be seen in Figure 3.6, the top level obligation $\mathit{Spec} \Rightarrow \mathit{ASpec}$ formally states every sequence of states respecting the temporal formula of action of the Barz specification should make the formula corresponding to the parametrized instance of the abstract specification *ASpec* true as well. Decomposing the obligation to the next level first expresses that the initial states of the behaviors allowed by the Barz specification make the *Init* formula true. $\mathit{Sema}!\mathit{Init}$ should thus hold for the equivalent abstract states. Then, the main obligation states that any action (stuttering step included) that can be taken under the *Next* formula of Barz should make its corresponding abstract action true. The typing and main invariants are added as assumptions and justified by the previous theorems as they are required to provide guidance to TLAPS when proving some of the subsequent obligations.

Note that although the overall structure of the inductive proof is very similar to the previous ones used for the invariant, there is a fundamental difference in that the main statement now evaluates actions against actions whereas they were only present on the left-hand side of $\Rightarrow$ connector. As explained in Section 2.2, the TLA logic does not care about this distinction thanks to simple state predicates being considered a subcategory of actions but there is nevertheless a conceptual difference to keep in mind.

The first obligation on initial states can be proven by TLAPS without further effort but the main one requires decomposition over the possible actions that are part of the *Next* disjunction. In order to do so, stuttering steps of the Barz specification must be treated separately. They are trivially proven to always correspond to a stuttering step in the abstract specification. Then, as expected from the label refinement mapping elaborated in the previous Section 3.8, the subobligations in which the active process takes the step $b0$, $s3$ or $s4$ are the only ones whose proof requires the definition of $\mathit{Sema}!\mathit{Next}$. This means that all the other steps correspond to a stuttering step in the abstract specification as planned. Moreover, the mapping precisely matched the $b0$ action to $s0$ as well $w3$ and $s2$ to $s1$ which turned out successful in the current proof.

Finally, one can observe the main invariant is required for the $w3$ and $s2$ steps, the reason being that the unique $s1$ abstract action they correspond to comprises both of their behaviors (incrementing or decrementing *counter* by 1) with appropriate conditions to "pick the right one". However, in Barz's algorithm, these conditions are not directly expressed under the $w3$ or $s2$ actions. For example, the $\textbf{await}\mathit{counter} < N$ check before incrementing *counter* is done by the $b1$ action but the overall logic of Barz's algorithm guarantees this condition is still true when the process eventually takes the

$s2$ step. Unfortunately, when looking at the $s2$ action and the abstract $s1$ one it is supposed to emulate, one notices that the former performs the increment operation unconditionally while the latter only allows it if $counter < N$. Introducing the previously proven invariant conveniently allows TLAPS to deduce $counter < N$ from the fact that there exists a process at label $s2$ thanks to its sixth clause. Any pair of states satisfying the $s2$ action can thereby be proven to abstractly satisfy $s1$. Indeed, although the $counter < N$ statement is not explicitly checked by $s2$ on the first state of the pair, it is nevertheless ensured by the assumption that the invariant holds for any state appearing in a behavior satisfying the Barz specification.

Chapter 4

Readers-Writers

4.1 Problem Setting

The readers-writers exclusion scheme is a classical synchronization problem that has many applications in computer systems and beyond. It builds upon the simple locking scheme by softening the mutual exclusion constraint which can be summed up as “only one user can access the protected resource at any given time”. Indeed, the readers-writers setting differentiates a subcategory of users called the readers that may share concurrent access to the protected resource. The remaining users, de facto called the writers, behave like previously by considering that only one user may access the protected resource at a time, regardless of its subcategory. Such a mutual exclusion scheme calls for an asymmetrical corresponding access policy. The writers must indeed be forbidden from accessing the protected resource if any user is currently holding it whereas readers are prevented access if and only if a writer is holding it. The main goal of the readers-writers problem is to provide a data structure and associated methods that would allow users to be guaranteed to follow the above mutual exclusion rules at all times.

Beyond these primary exclusion constraints, Professor Fontaine’s lecture [6] requires two additional properties for proposing a solution. First, a candidate solution should be generalized to an arbitrary number of readers and writers. Rigorously speaking, both the total number of users and the allotment of those user to the two subcategories are unspecified. Secondly, the candidate solution should be starvation free for the user regardless of the category it belongs to.

The following solutions considers all involved users have access to the shared resource to be protected in the first place. Therefore, the readers-writers exclusion property can only be guaranteed if all users respect the algorithm ”honestly”. No strict “computer security” guarantee could thus be stated when it comes to denying access to the resource to a malevolent user. Because access to the resource is a prerequisite to this setting, such a user could simply decide not to follow any algorithm and head straight to the resource in the absence of any other constraining mechanism.

Similarly, no guarantee to respect any form of semantics associated to the status of reader or writer could be given. The proposed methods allow any user to request access as a reader or a writer and to synchronize with the other users accordingly. However, once the user enters the critical section and gets access to the resource, the following template algorithms do not state anything about the behavior the user should adopt depending on its status. Therefore, it might be useful to consider other mechanisms on top of this one to guarantee users respect what it means to be a reader or writer in a particular use case.

4.2 Proposed Solutions

Professor Fontaine's lecture [6] first presents a bit of a context on how the candidate solutions will be used which allows to infer a general structure. Three solution are subsequently proposed by increasing degree of complexity.

To begin with, one can consider an interface for the readers-writers implementation similar to the one of a regular lock. This schematically boils down to processes asynchronously performing an arbitrary job up to the point they require access to the shared resource to be able to continue. At this point, they execute one method to wait until they are granted access to the shared resource which is often described as entering the critical section. This step corresponds to the `lock()` method of a basic locking mechanism or, in the readers-writers case, a pair of `read_enter()`, `write_enter()` methods to enter the critical as a reader or writer respectively. Once in its critical section, the process executes the part of its job for which access to shared resource was required and then exits the critical section with a dedicated method such as `unlock()`, `read_exit()` or `write_exit()` respectively. Finally, one can express a general process by assuming this sequence will be repeated continuously.

It is already worth noting a slight difference between [6] and the specification that will follow. Indeed, the first considers two separate process types for readers and writers, meaning reader processes only ever require reader access to the resource and likewise for writer processes. On the other hand, the next specification uses a single process type that may arbitrarily choose between reading and writing every time it requires access to the shared resource. The latter approach has the benefit of making less working assumptions and obviously covers the former as a particular instance of process behavior.

The first naive solution is to use the readers-writers interface as a wrapper around a traditional lock. This scheme trivially holds as a solution to the readers-writers problem by piggybacking on all the properties already proven for the lock. However, the purpose of differentiating the readers-writers problem from the strict mutual exclusion one in the first place is wholly defeated. Indeed, such an approach implements full sequentialization which results in readers never being able to share the resource and thus not taking advantage of potential parallelization.

The second solution relies on a binary semaphore, a lock and a counter that form a data structure. To understand this solution, one can refer to Figure 4.1, putting aside the fair_lock variable. Basically, the algorithm revolves around the binary semaphore write_lock. On the writer side, it is used exactly like a lock, which results in strict mutual exclusion between writers. On the other hand, multiple readers may collectively hold write_lock. The first reader to access the critical section is responsible for acquiring the write_lock in order to check there is no writer already accessing the resource. However, thanks to read_count having been incremented, following readers know they do not need to acquire write_lock as there is already another reader in the critical section. Conversely, writers must systematically acquire write_lock to enter the critical section. They must thus wait for the readers to free the binary semaphore. When a reader exits the critical section, it decrements read_count to indicate it to the other ones. If it realizes that it was the last reader, it releases the binary semaphore since there is no process accessing the shared resource left. As a side note, this means that a reader that acquired write_lock may transfer the burden of freeing it to the other ones. This corresponds to the notion of transfer of permissions of the binary semaphore developed for Barz's algorithm. Finally, read_lock is simply required to ensure local mutual exclusion between the readers when handling the read_count variable.

The third and final solution is the one presented in Figure 4.1, is the same as the second one with the addition of the fair_lock variable. The role of this lock is to prevent the possible starvation of writers by readers taking turns without freeing write_lock. If this semaphore is assumed to be fair regarding the processes that try to acquire it, then the readers-writers should be fair as a whole. This would require a formal proof based on fairness properties that is beyond the scope of this Master's thesis. This solution has nevertheless be used in the following to allow the possible addition of these properties and associated proofs in the future.

```
procedure read_enter
    fair_lock.Lock();
    read_lock.Lock();
    read_count ← read_count + 1;
    if read_count = 1 then
        write_lock.Wait()
    read_lock.Unlock();
    fair_lock.Unlock();
```

```
procedure write_enter
    fair_lock.Lock();
    write_lock.Wait();
    fair_lock.Unlock();
```

```
procedure read_exit
    read_lock.Lock();
    read_count ← read_count − 1;
    if read_count = 0 then
        write_lock.Signal()
    read_lock.Unlock()
```

```
procedure write_exit
    write_lock.Signal()
```

Figure 4.1: The Readers-Writers Algorithm adapted from [2]

4.3 PlusCal Specification

The PlusCal specification given for the readers-writers in Figure 4.2 shares the same outer structure as the one for Barz's algorithm given in Figure 3.2. The body of the algorithm is indeed made of a **process** statement to indicate concurrency, then an infinite **while** loop to allow processes to repeat the behavior multiple times. Each iteration contains an **either-or** branching to nondeterministically choose to behave like a reader or a writer. Just like for Barz's algorithm, processes are thus free to change the branch to follow at each iteration. There is however no restriction similar to **await** $counter > 0$ that could block one branch under certain circumstances. Two macros are also defined for the instructions on locks. They are exactly the same as the *wait* and *signal* macros from the Barz specification but are given the name *lock* and *unlock* since, strictly speaking, the variables are used as locks and not binary semaphore in this case.

The read and write branches contain the respective enter and exit procedures with a *skip* in between. This statement represents the arbitrary job performed by the process for which the respective access mode was required in the first place. Note that there is an unwritten assumption in that the process does not modify the variables present in the specification as part of this job. A second implicit condition allowed by the structure of this specification is that the processes use the readers-writers interface as intended, by calling the enter and exit procedures corresponding to the access mode they require. Putting corresponding enter and exit methods in the same branch

```
--algorithm Readers_Writers {
    variables
      fair_lock = 1,
      read_lock = 1,
      read_count = 0,
      write_lock = 1 ;

    macro lock( l ) {
      await l = 1 ;
      l := 0 ;
     }

    macro unlock( l ) {
      l := 1 ;
     }

    process ( proc ∈ 1 .. P ) {
rw0:    while ( TRUE ) {
         either {
re1:        lock(fair_lock) ;
re2:        lock(read_lock) ;
re3:        read_count := read_count + 1 ;
re4:        if ( read_count = 1 ) {
re5:          lock(write_lock)
            } ;
re6:        unlock(read_lock) ;
re7:        unlock(fair_lock) ;
            skip ;
rx1:        lock(read_lock) ;
rx2:        read_count := read_count − 1 ;
rx3:        if ( read_count = 0 ) {
rx4:          unlock(write_lock) ;
            } ;
rx5:        unlock(read_lock) ;

           } or {
we1:         lock(fair_lock) ;
we2:         lock(write_lock) ;
we3:         unlock(fair_lock) ;
             skip ;
wx1:         unlock(write_lock) ;
           }
         }
    }
 }
```

Figure 4.2: PlusCal Specification for the Readers-Writers

prevents behaviors in which a process tries to mix and match incompatible methods or to acquire the resource again without having released it.

When it comes to the labeling choice, nothing particular is to be said. Beside the body of the macros, there is no tricky grouping of instructions to form a single atomic operation, unlike Barz algorithm. The lines are thus simply labeled with the same numbers as in the pseudo-code of Figure 4.1, with additional prefixes *re*, *rx*, *we*, *wx* for `read_enter`, `read_exit` and so on. Finally, when it comes to the instructions contained in both branches, they are straightforward adaptation from the pseudo-code ones presented in Figure 4.1.

4.4 Simple Inductive Invariants

Similarly to the process presented in Section 3.5 for Barz's algorithm, three preliminary simpler invariants have been formulated and proven inductive with the help of TLAPS before proceeding with the main one. All the invariants are given in Figure 4.3 along accompanying definitions. The first of them is a loose typing invariant stating the tree lock variables *fair_lock*, *read_lock* and *write_lock* can obviously only take the values 0 or 1. On the other hand, *read_count* is said to be able to take any integer value for the exact same reason *counter* could not initially be strictly bounded to $[0, N]$ in the Barz specification.

The two other preparatory invariants have to do with the basic mutual exclusion provided by the two *fair_lock* and *read_lock* variables. They are perfectly analogous to the lock encountered in Barz's algorithm and the invariant are thus exactly the same, putting the definition of their respective critical section aside. The first of these critical sections is defined as *fairLockCS* and covers the whole body of the `read_enter` and `write_enter` sections past their first line. The second one is similarly defined as *readLockCS*. It expresses strict mutual exclusion from re3 to right after re6 in `read_enter` and from rx2 to the end of `read_exit`.

4.5 Main Inductive Invariant

The starting idea for the main invariant presented as *Inv* in Figure 4.3 is simply to state the readers-writers exclusion scheme as a safety property. In order to do so, two additional critical sections of the algorithms have been identified. The first one is called *readerInCS* and ranges from re6 to rx4 and thus covers parts of both `read_enter` and `read_exit` as well as the **skip** statement in between. As its name indicates, a process p of *ProcSet* verifying the *readerInCS*(p) predicate is to be interpreted as a reader which has been granted access to the shared resource. The exact same thing could be

$$\begin{array}{l}
\mathit{fairLockCS}(p) \triangleq \\
\quad \mathit{pc}[p] \in \{\text{``re2''}, \text{``re3''}, \text{``re4''}, \text{``re5''}, \text{``re6''}, \text{``re7''}, \text{``we2''}, \text{``we3''}\} \\
\mathit{readLockCS}(p) \triangleq \\
\quad \mathit{pc}[p] \in \{\text{``re3''}, \text{``re4''}, \text{``re5''}, \text{``re6''}, \text{``rx2''}, \text{``rx3''}, \text{``rx4''}, \text{``rx5''}\} \\
\mathit{readerInCS}(p) \triangleq \\
\quad \mathit{pc}[p] \in \{\text{``re6''}, \text{``re7''}, \text{``rx1''}, \text{``rx2''}, \text{``rx3''}, \text{``rx4''}\} \\
\mathit{writerInCS}(p) \triangleq \\
\quad \mathit{pc}[p] \in \{\text{``we3''}, \text{``wx1''}\} \\
\mathit{readCountCS}(p) \triangleq \\
\quad \mathit{pc}[p] \in \{\text{``re4''}, \text{``re5''}, \text{``re6''}, \text{``re7''}, \text{``rx1''}, \text{``rx2''}\}
\end{array}$$

$$\begin{array}{l}
\mathit{ProcsInReadCountCS} \triangleq \\
\quad \{p \in \mathit{ProcSet} : \mathit{readCountCS}(p)\}
\end{array}$$

$$\begin{array}{l}
\mathit{TypeInv} \triangleq \\
\quad \land\ \mathit{fair\_lock} \in \{0, 1\} \\
\quad \land\ \mathit{read\_lock} \in \{0, 1\} \\
\quad \land\ \mathit{write\_lock} \in \{0, 1\} \\
\quad \land\ \mathit{read\_count} \in \mathit{Int} \\
\quad \land\ \mathit{pc} \in [\mathit{ProcSet} \rightarrow \\
\qquad \{\text{``rw0''}, \text{``re1''}, \text{``re2''}, \text{``re3''}, \text{``re4''}, \text{``re5''}, \text{``re6''}, \text{``re7''}, \\
\qquad\quad \text{``rx1''}, \text{``rx2''}, \text{``rx3''}, \text{``rx4''}, \text{``rx5''}, \text{``we1''}, \text{``we2''}, \text{``we3''}, \text{``wx1''}\}]
\end{array}$$

$$\begin{array}{l}
\mathit{FairLockInv} \triangleq \\
\quad \land\ \forall\, i, j \in \mathit{ProcSet} : (i \neq j) \Rightarrow \neg(\mathit{fairLockCS}(i) \land \mathit{fairLockCS}(j)) \\
\quad \land\ (\exists\, p \in \mathit{ProcSet} : \mathit{fairLockCS}(p)) \Rightarrow \mathit{fair\_lock} = 0
\end{array}$$

$$\begin{array}{l}
\mathit{ReadLockInv} \triangleq \\
\quad \land\ \forall\, i, j \in \mathit{ProcSet} : (i \neq j) \Rightarrow \neg(\mathit{readLockCS}(i) \land \mathit{readLockCS}(j)) \\
\quad \land\ (\exists\, p \in \mathit{ProcSet} : \mathit{readLockCS}(p)) \Rightarrow \mathit{read\_lock} = 0
\end{array}$$

$$\begin{array}{l}
\mathit{Inv} \triangleq \\
\quad \land\ \forall\, i, j \in \mathit{ProcSet} : \\
\qquad\quad (i \neq j) \Rightarrow \neg(\mathit{writerInCS}(i) \land (\mathit{writerInCS}(j) \lor \mathit{readerInCS}(j))) \\
\quad \land\ (\exists\, p \in \mathit{ProcSet} : \mathit{writerInCS}(p) \lor \mathit{readerInCS}(p)) \Rightarrow \mathit{write\_lock} = 0 \\
\quad \land\ \mathit{IsFiniteSet}(\mathit{ProcsInReadCountCS}) \\
\quad \land\ \mathit{read\_count} = \mathit{Cardinality}(\mathit{ProcsInReadCountCS}) \\
\quad \land\ \mathit{read\_count} > 1 \Rightarrow \exists\, p \in \mathit{ProcSet} : \mathit{pc}[p] \in \{\text{``re7''}, \text{``rx1''}\} \\
\quad \land\ (\exists\, p \in \mathit{ProcSet} : \mathit{pc}[p] = \text{``rx4''}) \Rightarrow \mathit{read\_count} = 0
\end{array}$$

Figure 4.3: TLA$^+$ Invariants and Other Definitions for the Readers-Writers

said about *writerInCS*(p) for writers, corresponding to the critical section composed of labels `we3` and `wx1`. The choice of starting and ending points for both sections is determined by the locking and unlocking operations on *write_lock* respectively. This lock indeed acts as the underlying synchronization primitive allowing the two categories of users to agree on which of them has access to the shared resource.

Using the two critical sections defined hereabove, the first clause of the invariant can be written. It states that, for any given two process, if one of them is a writer currently in its critical section then the other one should not be in a critical section, regardless of it being a reader or a writer. This clause effectively expresses the readers-writers exclusion pattern as intended. It indeed implements mutual exclusion around writers while allowing multiple readers to access their critical section concurrently. In the same way as all the locks previously presented, the resulting mutual exclusion property is not sufficient as an inductive invariant. The second clause was therefore added in an effort to relate the variable used to implement the locking mechanism to the property. In the current case, *write_lock* should hold the value 0 as soon as there is a reader or a writer in the corresponding critical section.

As could be expected, the two previous clauses do not constitute an inductive invariant. They are quite similar to the first clauses of the main invariant developed for Barz's algorithm in Section 3.5 in that they express more complex patterns in the same way as basic locks. In order to hold over the whole algorithm, such invariants however need to encompass the overall logic behind it. All attempts to reinforce the invariant with clauses relating the values of the different variables like for Barz's algorithm nonetheless failed to produce an inductive invariant. Such clauses were not violated by the specification when checking models with the help of TLC, but they were however not sufficient to hold inductively. This is due to a fundamental difference in the use of the respective counter variable between both algorithms.

In Barz's algorithm, *counter* acts a history variable, keeping track of the difference in the number of calls to `Signal()` and `Wait()` since the beginning. The value of *counter* can thus not be precisely inferred from the ones of the other variables of the algorithm in a given state. On the other hand, the *read_count* variable of the readers-writers exactly counts the number of readers currently accessing the shared resource. As a result, the value of *read_count* should be equal to the number of processes whose program counter falls in the corresponding critical section. Due to the manipulation of *write_lock* and *read_count* not being part of the same atomic step, this critical section does not perfectly overlap with the previously defined *readerInCS*. A new predicate called *readCountCS* is introduced to delimit the section in between the increment and decrement of *read_count*, thus ranging from label re4 to rx2. Then, the set of processes of *ProcSet* verifying this predicate is given the alias *ProcsInReadCountCS*.

In order to reason on the number of processes in the *ProcsInReadCountCS* set, the notion of cardinality must be introduced. This well-known property of sets is not available by default in a TLA$^+$ module. To introduce its definition, the module must extend the `FiniteSets` standard library. As its name indicates, this standard library limits the notion of cardinality to finite sets, which is sufficient since the total number P of processes in *ProcSet* is assumed to be a natural number. A predicate called $IsFiniteSet(S)$ is handily provided as well, in order to state that a given set S is finite. The third clause of the invariant relies on this predicate to express that *ProcsInReadCountCS* is a finite set. The $Cardinality(S)$ operator consequently allows to consider the cardinality of said set in the fourth clause which states that it is equal to *read_count* at all times.

When trying to prove that the current invariant is inductive, one faces a new kind of issue. Simply put, every statement relying on seemingly trivial properties of the notion of cardinality can not be proven by using the provided definition of *Cardinality*. For example, TLAPS is not even able to verify that the cardinality of *ProcsInReadCountCS* is initially null although it should be obvious that there is no process in *readCountCS*. This phenomenon is due to the fact that the definition of *Cardinality* is impractical to use as part of proofs. A recursive reasoning is indeed used to define the cardinality of S as 1 plus the cardinality of a subset of S obtained by removing an arbitrary element. As a result, a number of nested susbets equal to the final cardinality must be developed before reaching the empty set and stopping the process. Due to the necessity to reason by induction, this form of definition is difficult to infer useful properties from and TLAPM struggles as soon as it is introduced. Fortunately, the TLAPS standard library contains a set of well-known already proven theorems regarding the notion of cardinality within the `FiniteSetTheorems` file.

Among all the provided theorems, 4 are necessary to the proof of the invariant and are detailed in the following. First, *FS_EmptySet* expresses that a finite set which has a null cardinality is and can only be the empty set. Then, *FS_CardinalityType* is used, among other things, to type the cardinality of a finite set as a natural number. Finally, the two symmetric theorems *FS_AddElement* and *FS_RemoveElement* allow to reason on finite sets when adding or removing an element respectively. In both cases, the new set is guaranteed to remain finite and its cardinality is related to the one of the original set. If a new element that did not belong to the set beforehand is added, then the cardinality is incremented by 1. Conversely, if an element that was indeed part of the set is removed, its cardinality is decremented by 1. On top of these 4 provided theorems, a new one was defined under the name *FS_NonEmptySet*. This theorem states that, for a finite set, the cardinality being greater than 0 means that (at least) one element can be found in this set and conversely.

Using the previous theorems, one can try to write the proof for the invariant. First, the definitions of *Inv*, *ProcsInReadCountCS*, *readCountCS*, *writerInCS* and *readerInCS* as well as the assumption that constant P is a natural number are made available to the solvers for the whole proof. Then, following the natural decomposition of the top level statement $Spec \Rightarrow \Box Inv$, the first step is to prove that *Init* respects the invariant. As already indicated previously, TLAPS is able to prove that *ProcsInReadCountCS* is the empty set but not that its cardinality is null and matches the value of *read_count*. Instead of trying to justify this by the definition of *Cardinality*, one can now use the theorem *FS_EmptySet* to verify the fourth clause of the invariant.

Once $Init \Rightarrow Inv$ is verified, it is time to proceed with the main endeavor of proving the invariance of *Inv* with respect to all the possible actions of the specification. As for every inductive proof encountered until now, the automatic decomposition feature of TLAPS is used. The invariance must thus be proven for each action making up *Next*, while assuming a given process *self* is taking said action. The possibility of a stuttering step must also be covered by the proof. Most of the steps can be trivially proven by TLAPS by only relying on the definition of the corresponding action and the soft typing invariant *TypeInv*. Regarding the failing steps, some simply require the help of the theorems on finite sets or of the mutual exclusion invariants. However, the information conveyed by the invariant is still not sufficient for the proof to hold completely.

Similarly to the Barz invariant, one should add clauses to account for the guardian conditions of the **if** blocks. For example, when looking at step $re5$ only, TLAPS is not able to determine that the process is within the body of the **if** block. Combined with the mutual exclusion of *read_lock*, the fact there is a process at the corresponding label is however sufficient to infer $read\_count = 1$. In order to spare TLAPS this more complex proof, the fifth and sixth clause of the invariant have been added.

The sixth clause corresponds directly to the **if** conditional in `read_exit`. It constitutes the missing piece to prove that it is safe for a process in $rx4$ to unlock *write_lock* without risking to violate the second clause of the invariant. Simply put, one knows *read_count* is null in this situation which means there is no reader left in *readCountCS* thanks to the fourth clause. Moreover, no other process than the considered one can be at labels $rx3$ or $rx4$ thanks to the mutual exclusion on *read_lock*. Therefore, there is no other process verifying *readerInCS* left and the second clause is verified.

The fifth clause of the invariant is a consequence of the second one as well as the mutual exclusion on *read_lock*. Basically, when there is more than a single reader in *readCountCS*, one can be sure there will be at least one process at label $re7$ or $rx1$. This is due to the fact that those are the only two labels part of *readCountCS* which are not affected by the mutual exclusion on *read_lock*. By explicitly stating this point, this clause is useful at $re4$ to justify it is safe to jump directly to $re6$ if $read\_count \neq 1$. One should indeed be sure the readers have already acquired the *write_lock* before entering *readerInCS*, otherwise violating the second clause of the invariant. Due to the considered process being at $re4$, it is obvious that $read\_count > 0$ thanks to the fourth clause. Therefore, the precondition $read\_count > 1$ is guaranteed to be true if the process skips the body of the **if** block. In that case, the newly introduced clause states there is at least one other process in *readerInCS*, at label $re7$ or $rx1$. Thanks to the second clause, TLAPS can be sure the readers are collectively already holding *write_lock* in order for this process to have reached that point. As a side note, one may notice the fifth and sixth clauses have symmetrical roles in that the former guarantees there is already an active reader while the latter states there is no reader left.

The current invariant can finally be proven inductive for all steps of the specification. The following describes the information to add for TLAPS to be able to verify the non-trivial steps.

The first problematic action is $re3$ because it increments the *read_counter* variable. The *FS_AddElement* theorem must thus be used to ensure the cardinality of *ProcsInReadCountCS* stays equal to this updated value in the new state. Practically speaking, manual statements had to be added beside the ones of the automatic proof decomposition. TLAPS indeed struggles to use the finite set theorems if their assumptions are not explicitly given under the exact expected form. Those additional proof steps have been numbered with letters to differentiate them. Additionally, *FS_NonEmpySet* as well as mutual exclusion must be invoked to prove the fourth clause of the invariant. Simply put, as *read_count* is incremented and may get past the value 1, TLAPS must check there is indeed a process at labels $re7$ or $rx1$ in this case. *FS_NonEmpySet* effectively states there is at least one process in *readCountCS* and the mutual exclusion restricts the section to these two labels.

Secondly, the two first clauses of the argument do not hold for action $re4$. In the case where the process jumps directly to $re6$ and enters the critical section as a reader, TLAPS can indeed not guarantee there is not already an active writer. This case nevertheless implies $read\_count > 1$ as explained when introducing the fifth clause. It is thus sufficient to invoke *FS_NonEmptySet* for TLAPS to understand that the cardinality of *ProcsInReadCountCS* being greater than 1 means there is already a reader in the critical section.

Because the two first clauses of the invariant are assumed to be verified in the current state, it is obvious that there is no writer in the critical section and that the readers already collectively hold *write_lock*.

Then, the fifth clause does not hold for action $rx1$. This is due to the fact that the active process exits the section composed of labels $re7$ and $rx1$. TLAPS must thus ascertain there is another process in said section in the case where $read\_count > 1$. This argument is stated as step $\langle 3 \rangle\, d$ and requires to reason on the set *ProcsInReadCountCS* when removing the active process from it. Because the cardinality of the original set is supposed to be strictly greater than 1, *FS_RemoveElement* states the updated cardinality after removing the process is still strictly greater than 0. *FS_CardinalityType* is required for TLAPS to transfer this basic reasoning on integer values to cardinalities. Lastly, *FS_NonEmptySet* is used to state there is still at least one element in the resulting set. Thanks to this chain of basic elements of reasoning, the original argument is verified and the invariance holds for $rx1$.

The $rx2$ action is symmetrical to $re3$, decrementing *read_count* instead of incrementing it. Hence, although not trivially solved, its proof is analogous to the one for $re3$ but relies on *FS_RemoveElement* instead of *FS_AddElement*.

Finally, the proof for action $rx4$ requires to declare that *ProcsInReadCountCS* is empty before using the sixth clause of the invariant along mutual exclusion. As one can remember, this was the motivation behind the introduction of said clause.

Chapter 5

Conclusion

The goal of this Master's thesis was to specify and verify safety properties of parallel programming algorithms taken from Prof. Pascal Fontaine's lecture, using the TLA$^+$ Toolbox.

Regarding Barz's algorithm, a formal PlusCal specification has first been written. Three inductive TLA$^+$ invariants have then been developed, checked with the TLC model checker and complete TLAPS proofs were produced and verified. The safety properties associated to those invariants respectively express type restrictions on the variables, mutual exclusion due to a lock and the overall logic of the algorithm. A second abstract specification was then written to lay out the expected behavior of a general semaphore regardless of its implementation. Finally, a refinement mapping was found to relate both specifications and a proof of refinement has been formulated. This proof relies on the previous invariants and formally shows that Barz's algorithm is a valid implementation of the general semaphore. As a result, any additional property that could be proven for Barz's algorithm would also hold for the general semaphore specification, under the defined refinement mapping.

A similar process has been carried out for the readers-writers algorithm. A formal PlusCal specification compliant with the original pseudo-code formulation was produced. Three prerequisite typing and mutual exclusion inductive invariant were checked and proven. Finally, the main inductive invariant describing the expected behavior of the algorithm required a way more complex proof. This proof writing process provided an opportunity to experiment with the theory of finite sets of the TLAPS standard library. Compared to the other proofs of this work, the main inductive invariant of the readers-writers required extensive manual fine-tuning to be verified.

In order to take this work further, many more algorithms are available to specify in the Parallel Programming course and beyond. For example, the Dining Philosophers' and reusable barriers for n processes are similar problems to the ones presented in this Master's thesis and could be a good starting point. On the other hand, it would also be possible to find and prove other properties for the presented algorithms. Beside additional safety properties, liveness and fairness properties are good candidates as they have not been touched upon. Even though they are typically harder to prove, TLAPS is nonetheless equipped with primitives to express and write proofs for such properties.

Bibliography

[1] Hans W. Barz. "Implementing Semaphores by Binary Semaphores". In: *ACM SIGPLAN Notices* 18.2 (1983), pp. 39–45. DOI: 10.1145/948101.948103. URL: https://doi.org/10.1145/948101.948103.

[2] Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentice Hall international series in computer science. Addison-Wesley, 2006. ISBN: 9780321312839. URL: https://books.google.be/books?id=oP-2hpMEdb8C.

[3] Kaustuv Chaudhuri et al. "Verifying Safety Properties with the TLA + Proof System". In: *Automated Reasoning.* Ed. by Jürgen Giesl and Reiner Hähnle. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 142–148. ISBN: 978-3-642-14203-1.

[4] Edsger W. Dijkstra. "Co-operating sequential processes". English. In: *Programming languages : NATO Advanced Study Institute : lectures given at a three weeks Summer School held in Villard-le-Lans, 1966 / ed. by F. Genuys.* United States: Academic Press Inc., 1968, pp. 43–112. ISBN: 0-12-279750-7.

[5] Edsger W. Dijkstra. "Over de sequentialiteit van procesbeschrijvingen [On the sequentiality of process descriptions]". circulated privately, translation available in Texas Archive. undated, 1962 or 1963. URL: http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD35.PDF.

[6] Pascal Fontaine. *INFO9012-1 Parallel Programming Lecture Slides.* University of Liège. 2024.

[7] Leslie Lamport. "Specifying Concurrent Program Modules". In: *ACM Trans. Program. Lang. Syst.* 5.2 (Apr. 1983), pp. 190–222. ISSN: 0164-0925. DOI: 10.1145/69624.357207. URL: https://doi.org/10.1145/69624.357207.

[8] Leslie Lamport. "The PlusCal Algorithm Language". In: *Theoretical Aspects of Computing-ICTAC 2009, Martin Leucker*

and Carroll Morgan editors. Lecture Notes in Computer Science, number 5684, 36-60. (Jan. 2009).
URL: `https://www.microsoft.com/en-us/research/publication/pluscal-algorithm-language/`.

[9] Leslie Lamport. “The temporal logic of actions”. In: *ACM Trans. Program. Lang. Syst.* 16.3 (May 1994), pp. 872–923. ISSN: 0164-0925. DOI: `10.1145/177492.177726`.
URL: `https://doi.org/10.1145/177492.177726`.

[10] Leslie Lamport and Stephan Merz. *Auxiliary Variables in TLA+.* 2017. arXiv: `1703.05121 [cs.LO]`.
URL: `https://arxiv.org/abs/1703.05121`.

[11] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. “Model Checking TLA+ Specifications”. In: *Correct Hardware Design and Verification Methods.* Ed. by Laurence Pierre and Thomas Kropf. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 54–66. ISBN: 978-3-540-48153-9.