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

Auteur : Differdange, Jarod
Promoteur(s) : Fontaine, Pascal
Faculté : Faculté des Sciences appliquées
Diplôme : Master : ingénieur civil en informatique, à finalité spécialisée en "computer systems security"
Année académique : 2024-2025
URI/URL : http://hdl.handle.net/2268.2/23374

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 Differdange Jarod

University of Liège
-
School of Engineering and Computer Science
Academic year 2024-2025

Acknowledgments

Firstly, I want to thank Professor Pascal Fontaine for being my promoter and for his support and guidance along this academic year.

Secondly, I would like to express my gratitude towards my co-promoter Stephan Merz, Senior Researcher at the INRIA Nancy research center, for his help on understanding TLA$^+$ early. This gratitude extends to the members of the VeriDis team whom I had the pleasure to meet during a week-long stay in Nancy.

Finally, I want to thank my family and friends for their support throughout this and previous academic years.

Abstract

The goal of this Master's thesis is to continue the effort of specifying and verifying safety properties of the different algorithms presented in the Parallel Programming course given by Professor Pascal Fontaine [1]. This endeavor is motivated by the creation of a formal companion to the course for students interested in the material. This companion will contain alternative descriptions of the algorithms presented, as well as their properties. This effort will also aim to identify potential errors or prove the correctness of the different algorithms presented in the lecture. Additionally, the lecture notes mention being able to exchange equivalent pieces of code when they have the same properties. The validity of this claim is also verified on an example.

The first chapter describes the principles of concurrent programming, the motivation behind its usage, and the challenges that come with it. The second chapter describes the Temporal Logic of Actions and the TLA$^+$ Toolbox. The TLA$^+$ environment allows the creation of specifications, their verification of properties on finite models, and the writing proofs. A description of all the different tools in the Toolbox is provided.

After the description of the tools, the barrier synchronization mechanism shown in [1] is first specified. Several invariant properties are described and proven invariant using the TLA$^+$ Toolbox. It is also shown that it behaves like an abstract barrier where processes advance synchronously. Secondly, by extending the specification of the abstract lock with auxiliary variables and using the refinement mechanism present in TLA$^+$, it is shown that the abstract lock and Peterson's algorithm are equivalent.

Contents

Chapter 1

Parallel Programming

1.1 Concurrent programming

Parallel programming is becoming increasingly attractive due to stagnating CPU clock frequencies and the increasing availability of multi-core processors. Using multiple cores to execute computations has the capacity to increase performance of programs compared to a sequential version.

However, introducing concurrency brings its own set of problems as the non-deterministic scheduling of different threads/processes may affect correctness of the program. An increase in performance is useless if the results of computations are not correct.

A program where n threads increment a shared variable, initially at 0, should be easy to write as it involves a simple operation. However, if the increment operation is not atomic, the shared variable might not contain n at the end of each execution. The non-determinism on the execution of concurrent programs could lead to two or more threads reading the same value of the shared variable and therefore putting the same increased value back into the variable.

To simplify the development of concurrent programs, there is a need for synchronization facilities to ensure programs behave as developers intend. Classical facilities like locks or semaphores are well known, but higher-level objects like barriers or message queues could further simplify writing code.

1.2 Interleaving semantics

To reason about concurrent programs, one needs semantics for the execution of concurrent/parallel programs. Such semantics are the interleaving semantics, which represent the execution of a concurrent/parallel program as any possible interleaving of the atomic steps of all processes.

The execution of a program can be seen as a sequence of states, which encompasses the values of all relevant variables. Such values are usually the values of program variables and the values of the program counter of each thread. The program counter indicates the next atomic operation the thread will execute. For a sequential algorithm, the succession of states is deterministic, as only one thread advances in the program. For a concurrent program, many threads can advance, and it is the computer's scheduler that decides which thread will advance.

The interleaving semantics state that even if threads execute in parallel, the execution of a parallel program can be considered as an interleaving of the threads' atomic steps. The simplest example would be two threads writing true into a thread-private variable initially set to false. The possible interleavings would be advancing the first thread then the second, or advancing the second thread, then the first thread. In this simple example, the two possible executions lead to the same result, but poorly designed parallel algorithms could yield different results depending on the interleaving. Therefore, parallel programs should be written with this problem in mind. Synchronization facilities allow programmers to control the non-determinism of interleavings. Locks ensure that only one thread can be in so-called critical sections where, for example, global variables are modified.

Parallel programs are hard to debug, as problems in the code might only appear for certain interleaving of processes. This coined the term "Heisenbugs", bugs that seem to disappear whenever one tries to debug them. Named after Heisenberg's indeterminacy principle, these bugs are hard to reproduce using conventional tools. The best way to reason about parallel programs is invariant properties, properties that are verified at each possible step of the execution. Such properties, when verified formally, can prove the correctness of the given program. Most invariants cannot be proven easily, as the only way to prove an invariant property is to show it is inductive or a logical consequence of other proven invariant properties. As the name suggests, an inductive invariant is one that is true initially, and such that any state (reachable or not) satisfying the property leads to another state satisfying the property after a step of the algorithm.

Chapter 2

The TLA$^+$ Language

2.1 Temporal Logic of Actions

The Temporal Logic of Actions, or TLA [2], created by Leslie Lamport, was a response to the difficulties of specifying a FIFO queue in Linear-time temporal Logic [3]. While LTL is particularly suitable for describing the properties of systems in terms of correctness and liveness, TLA may be seen as an extension of LTL with actions, i.e., predicates describing transitions rather than states.

2.1.1 Formalism of TLA

The Temporal Logic of Actions is a logic that describes discrete-time systems. It uses a linear-time approach, considering a single behavior and not a branching universe. A behavior is a sequence of states, and states are an assignment from variables to values. In TLA, variables are not typed and can therefore take any value. This mapping is denoted $s[\![x]\!]$ where s is a state and x a variable. A state can also assign a boolean value $s[\![P]\!]$ to a predicate P, by replacing each variable v with its value $s[\![v]\!]$. Any atemporal formula, which contains no temporal operators, has a meaning on a single state and is usually a first-order logic formula. On a behavior $\langle s_0, s_1, \ldots \rangle$, an atemporal formula F is evaluated on the initial state, e.g. $\langle s_0, s_1, \ldots \rangle [\![F]\!] \triangleq s_0[\![F]\!]$

The actions in TLA refer to a new type of predicate that assigns a truth value to an ordered pair of states. This ordered pair represents an initial state and a next state after a single transition. The notation for the evaluation of an action A on the pair of states (s, t) is $s[\![A]\!]t$. To distinguish between the values of variables in the two states, the variables in the initial state are written normally and the variables for the next state are primed, e.g. v'. A simple example of such an action predicate would be $y' = x$, which assigns true if for an ordered pair of states (s, t) $s[\![x]\!] = t[\![y]\!]$ holds. On a behavior, an action formula A is evaluated on the first two states (s_0, s_1), e.g. $\langle s_0, s_1, \ldots \rangle [\![A]\!] \triangleq s_0[\![A]\!]s_1$

The temporal operators present in TLA are the "always" $\Box$ and "eventually" $\Diamond$ operators, which also exist in LTL. The formula $\Box F$ holds if and only if F holds at each state of the behavior. The formula $\Diamond F$ holds if and only if F holds for at least one state of the behavior. Formally one can write :

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

If F is an atemporal formula, the right hand side reduces to $\forall n \in \mathbb{N} : s_n [\![F]\!]$, as only the first state is considered. If F is an action A, then only consecutive pairs of states need to be considered, e.g. $\forall n \in \mathbb{N} : s_n [\![F]\!] s_{n+1}$ The $\Diamond$ operator is defined with respect to $\Box$, as $\Diamond F \triangleq \neg\Box\neg F$. This operator is used to describe liveness properties. Since liveness is not the goal of this thesis, liveness properties will not be described.

2.1.2 Specification refinements

Using the logic to describe specifications, one can also define what it means to implement a specification. To verify this, one needs to verify if each behavior of an implementation $\mathcal{S}_I$ is also a valid behavior of the abstract specification $\mathcal{S}_A$. A simple example of refinement, as presented by Lamport himself many times, is the example of an hour clock and an hour-minute clock. It is evident that a clock showing the current hour and minute can function as a clock that only shows the hour by hiding the minute display. In the context of refinements, the specification of the hour clock would be $\mathcal{S}_A$, and the specification of the hour-minute clock $\mathcal{S}_I$. One would want to write $\mathcal{S}_I \Rightarrow \mathcal{S}_A$, but since both specifications might not operate on the same variables, this notation is not correct.

The last symbol that needs to be introduced is the so-called hiding operator $\exists$. It works similarly to the existential quantifier, but it is true if and only if there exists a sequence of values for the variables, for which the given TLA formula holds. It is called the hiding operator as it can bind the (internal) variables of a specification and allows one to reason on external variables only. It allows one to reason on the external behavior while hiding the internal implementation details. This operator is also called the temporal existential quantifier and can be used to correctly describe the meaning of refinement :

$$\exists \, \text{vars}_I : \mathcal{S}_I \Rightarrow \exists \, \text{vars}_A : \mathcal{S}_A$$

However, this general implication is not very useful when trying to prove the refinement from one specification to another. As a first step, we can remove the first quantifier, meaning that for any given behavior that satisfies $\mathcal{S}_I$, there exists a valid sequence of internal variables vars_A that satisfies $\mathcal{S}_A$. Finally, to prove the existence of such a sequence for the internal variables of $\mathcal{S}_A$, it suffices to find a mapping from the internal variables vars_I to vars_A. This would imply the definition of a new specification $\overline{\mathcal{S}_A}$ in which each variable v in vars_A is replaced by a mapping $\overline{v}$ from vars_I. Using the clocks defined above, the mapping is

trivial, as the hour variable of the hour clock can simply take the value of the hour variable in our hour-minute clock. The usage of a mapping of the variables lends its name to the name to the "refinement mappings". Finally, this results in a simpler formula to prove refinement mappings:

$$\mathcal{S}_I \Rightarrow \overline{\mathcal{S}_A}$$

To denote such a refinement in TLA$^+$, one can write :

```
Refinement == INSTANCE Abstract WITH var_A <- map(var_I)
```

Where

- `Refinement` will be used to refer to the refinement,
- `Abstract` refers to the module (defined later) containing the abstract implementation
- `var_A` and `var_I` will refer to some variable used in their respective specification.

Each variable will need to receive a mapping from the variables `var_I` and each variable that has the same name are implicitly assigned to each other.

2.2 The TLA$^+$ Toolbox

The TLA$^+$ Toolbox, accessible from the project repository,[1] is an IDE that integrates the different tools into one coherent graphical user interface.

The tools integrated into the IDE are the following:

- The *SANY* Synthetic Analyzer, which is a TLA$^+$ and PlusCal parser and syntax checker.
  This program is described in [4].
- The PlusCal Translator, which transpiles the easier-to-read PlusCal specification into a TLA$^+$ specification automatically.
  The translation is described in [5].
- The *TLATeX* Pretty-Printer, which uses the LaTeX engine to typeset ASCII specifications, creating a PDF file.
  This program is described in [4].
- The *TLC* model checker is an exhaustive model checker that allows the verification of safety or liveness properties on given TLA$^+$ specifications for fixed values of specification constants.
  This program is described in [4].

[1] `https://github.com/tlaplus/tlaplus/releases`

- The *TLAPS* Proof System [6] allows the writing of proofs in the TLA$^+$ language. The TLA Proof Manager takes TLA$^+$ assertions and converts them into an intermediary representation as proof obligations; these obligations can then be given to different backend provers.

  As TLAPS is an add-on created by the Microsoft Research-Inria Joint Center, it is not directly available inside the Toolbox. It must be downloaded, and installed separately and the installation path must be specified inside the toolbox.

2.3 Writing TLA$^+$ specifications

2.3.1 Pure TLA

To model digital systems, TLA$^+$ uses TLA and Zermelo-Fraenkel set theory. Variables are fundamentally sets and do not have types. However TLA$^+$ comes with modules to simplify working with the usual types, e.g. integers. Since TLA$^+$ uses logic to describe programs, the semantics of specifications are purely mathematical [4]. This allows the models to be free from the specifics of any particular programming language.

While this simplifies the creation of specifications, one must ensure that the created abstraction follows the reality of the program we want to verify. For example, if in a specification we write that two variables can be copied atomically at the same time, but the program behaves differently, some guarantees proven for the abstraction might not hold during execution.

A computer program executes a series of instructions and therefore runs one behavior. A TLA$^+$ specification, on the other hand, is a temporal logic formula, which can only assign boolean values to behaviors. In particular, it should only hold for the behaviors we want to describe.

In a first attempt, one could try to write a specification in the following form: Init $\wedge \Box$Next, where

- Init represents a state predicate that will assign true to all possible initial states of the model.
- Next represents an action which will assign true to each pair of states that form a valid transition.

The corresponding formula will assign truth values to every behavior $\sigma = \langle s_0, s_1, s_2, \ldots \rangle$, since s_0 must verify Init and each consecutive pair (s_n, s_{n+1}) must verify Next due to the $\Box$ operator.

This form is unfortunately insufficient to fully exploit the capacities of TLA$^+$, more precisely to allow the writing of refinements. Indeed, refinements might require specifications to perform stuttering steps, steps where the state does not change. Therefore, the values of variables does not change. One would

therefore need to write Next $\vee$ (vars$'$ = vars) to allow stuttering steps. Since stuttering steps are not part of a specification, one should not explicitly add this to the definition of Next. As stuttering steps are an essential part of TLA$^+$, a shorthand notation is used, which is $[\text{Next}]_{\text{vars}}$. With this addition, the canonical form of a TLA$^+$ specification would be :

$$\text{Init} \wedge \Box[\text{Next}]_{\text{vars}}$$

This formula does not include any liveness properties, as they are not needed for the specifications presented here, but TLA$^+$ has shorthands for weak and strong fairness constraints that can be added as further clauses in the formula.

Additional syntax is, of course, needed to fully write a TLA$^+$ file. A simple but complete TLA$^+$ module taken from [4] can be found in Figure 2.1. A `.tla` file contains a module that should start with `---- MODULE name ----`, to indicate the beginning of the file and end with `========`. These are designed to ensure that the ASCII version of the file looks like the pretty-printed version. Constants and variables need to be declared with the corresponding `CONSTANTS` and `VARIABLES` keywords. Both are variables in the mathematical sense of the word. However, constants are never modified, while variables are allowed to change during a behavior. Most logic symbols are written in ASCII or as their LaTeX equivalent. Definitions of formulas use the double equality sign `==`, standing for the LaTeX symbol $\triangleq$.

```
---------------------- MODULE HourClock ----------------------
  (*****************************************************)
  (* This module specifies a digital clock that displays  *)
  (* the current hour. It ignores real time, not          *)
  (* specifying when the display can change.              *)
  (*****************************************************)
EXTENDS Naturals
VARIABLE hr     \* Variable hr represents the display.
HCini == hr \in (1 .. 12)   \* Initially, hr can have any
                            \* value from 1 through 12.
HCnxt   (* This is a weird place for a comment. *)  ==
  (*********************************************)
  (* The value of hr cycles from 1 through 12.     *)
  (*********************************************)
hr' = IF hr # 12 THEN hr + 1 ELSE 1
HC == HCini /\ [][HCnxt]_hr
  (* The complete spec. It permits the clock to stop. *)
-------------------------------------------------------------
THEOREM HC => []HCini \* Type-correctness of the spec.
=============================================================
```

Figure 2.1: A simple TLA$^+$ specification taken from [4].

Additional constructs that are more reminiscent of programming languages can also be used. Associative arrays are defined as functions with a domain equal to the set of possible indices. To modify a single entry of an array inside an action, it is insufficient to only write `arr'[i] = e`, as the values for other indices are left undefined. A special notation for array modification is used, which follows the following format : `arr' = [arr EXCEPT ![i] = e]`. Another useful construct is the if-then-else construct using the `IF`, `THEN`, and `ELSE` keywords. As usual for

a conditional, it allows conditional assignment given a boolean value. Another shorthand for long chains of if statements is the `CASE` keyword, where only the value of the first matching condition is returned.

Further details of the TLA$^+$ syntax are not needed to understand the specifications presented here. The full syntax can be found in [4].

2.3.2 PlusCal

The PlusCal algorithm language [5] was created to both ease the creation of specifications and provide a systematic translation from pseudocode to TLA. As the name indicates, it borrows a lot of syntax from the Pascal language, such as the block definitions using `begin` and `end`, a variables section, and a code section to name a few. PlusCal also allows the usage of braces, more reminiscent of C.

PlusCal specifications must appear after the opening `MODULE` line, possible imports (using `EXTEND`), the declaration of constants, and possible TLA definitions that may be needed. A simple example can be found in Figure 4.1. The description must be put inside a multi-line comment that starts with a `--algorithm` line to notify a PlusCal section. Then the variables and their initial values are defined after the `variables` keyword. Each variable is separated by commas, and the list is ended with a semicolon. Processes in PlusCal are introduced with the `process` keyword, with the possibility of creating multiple types of processes that are distinguished by name. Furthermore, to instantiate the number of processes, a `name \in Set` notation is used, introducing a process for each element in the given set. The natural choice would be an interval of integers from 1 to some constant N, denoted `1..N`, but any set is accepted.

Finally, the operations done by a process can be described. Possible operations are assignments using `:=` similar to Pascal, conditional branches with `if` and `else`, and `while` loops. An important operator not usually present in programming languages is the `await` operator, which allows waiting for a condition to be true before continuing execution. PlusCal allows the creation of macros that work as one would expect; they behave like a shorthand for one or multiple operations.

The most important part of PlusCal is the control of atomic operations. Indeed, PlusCal allows multiple operations to be performed in one atomic step. The atomicity of operations is defined by putting labels in the pseudocode. All operations between one label and the next one are executed in a single step. These labels are used by the program counter created when translating the PlusCal specification. The translated PlusCal appears between comments `BEGIN TRANSLATION` and `END TRANSLATION`. The second comment also contains hashes of both specifications (PlusCal and translated TLA$^+$) to allow other tools to notify the user in case they forgot to translate the PlusCal specification after modifying it. The translation adds a `VARIABLE` clause with all declared variables and the program counter as `pc`. Then it will define an `Init` formula using

the initial values defined in PlusCal, and it will define a `Next` formula and subformulas for each atomic operation using the previously mentioned labels.

The PlusCal language has more features than those described here, and describing it further would be out of the scope of this thesis. This description should cover everything presented in the following sections.

2.4 TLATeX

As Leslie Lamport was involved in the creation of both LaTeX and TLA$^+$, the Toolbox includes a pretty-printer for TLA$^+$ specifications. An example is shown in Figures 2.2 and 2.3, which show the TLA$^+$ source in ASCII and the typeset version of a theorem, respectively. However, this tool generates a PDF directly and does not produce an intermediate LaTeX output, which makes extracting a certain section of a specification impossible.

The standalone tool `tla2tex.TLA` is available to generate such an intermediary TeX file. This tool is bundled with others inside a Java archive (`tla2tools.jar`) which is available on the tlaplus Github repository.[2]

Using the `tlatex` package (available through `tlatex.sty`[3]), specifications, or parts of them, can be embedded in any LaTeX document.

2.5 The TLC model checker

In TLA$^+$, in addition to specifications, one can write properties that should hold. The TLA Model Checker, or TLC, described in [4], takes a specification and properties and will verify these properties on a given model. By a model of a specification, one denotes a certain assignment of the constants used within it, e.g., a constant N for the number of processes. Properties can be written using a TLA formula and then given to the model checker.

TLC is an explicit state model checker and will compute the set of all reachable states from a TLA$^+$ specification. This is computed using the `Init` and `Next` formulas that respectively describe the initial states and the transition relation. TLC will compute all initial states from `Init`, and the other reachable states are computed by fixed point from already computed reachable states using `Next`.

More precisely, TLC performs a breadth-first search. It maintains a graph $\mathcal{G}$ which represents the state reachability graph computed by TLC and a queue $\mathcal{U}$ of states whose successors have not yet been computed. First, initial states are added to $\mathcal{G}$ and to $\mathcal{U}$. Second, until $\mathcal{U}$ is empty, TLC will pop a state s, compute the successors t_i, and add the t_i to $\mathcal{G}$ and $\mathcal{U}$ if they are not already present in $\mathcal{G}$. Edges are also added to represent the transition from state s to t_i.

[2] `https://github.com/tlaplus/tlaplus/releases`
[3] `https://seldoc.eecs.yorku.ca/doku.php/latex/sty`

Whenever a state is explored, a fingerprint is computed and added to a set of visited states. Successor states that have a fingerprint present in this set are not enqueued for further exploration. The fingerprint is the hash of the state. Since the hash space is potentially smaller than state space and the hash function is imperfect, there can be collisions. TLC shows the probability of collisions after execution. When verifying only safety properties, TLC can try to reduce the state space drastically using symmetries. If one works with named processes P_1, P_2, P_3, one could explicitly state their equivalence and let TLC reduce the state space in that way.

Since we can only verify finite models, one needs to instantiate constants and possibly limit the values variables can take. If in a specification a certain variable was allowed to take any integer value, TLC would not be able to completely verify the model. The usage of TLC should be limited to verifying a certain model and guaranteeing that the model behaves correctly, or to trying to find traces in which the algorithm does not behave correctly. To find invariant properties about the behaviors of a specification, one can use TLC as well, and try to find counterexamples before proving them using TLAPS.

To specify a model, a configuration file is used, in which the user states which `Init` and `Next` formulas to use, the values of constants, and which properties to verify. The configuration file does not accept TLA formulas, only definitions already present in the TLA$^+$ module. The TLA$^+$ Toolbox simplifies the creation of models inside a special menu that abstracts away from the configuration file.

2.6 TLAPS

To write proofs using TLA$^+$, The TLA Proof System (TLAPS) [6] and its companion, the TLA Proof Manager (TLAPM), were created. TLAPS allows writing hierarchical proofs in the TLA$^+$ language, meaning that it is independent of the backend solver actually used for verification. TLAPM will transform TLAPS proofs and generate proof obligations for the backend solvers. One of these backend solvers is Isabelle/TLA$^+$, which is an axiomatization of TLA$^+$ inside the Isabelle [7] proof assistant. Complete documentation of the usage of TLAPS can be found here [8].

Taking Figure 2.2 as a reference, the structure of a TLAPS proof is as follows. It starts with either the `THEOREM` or `LEMMA` keyword to indicate the start of a proof. It is optionally followed by a name that can be used to refer to the current theorem in other proofs. Then comes the formula we want to prove. This formula can be a TLA$^+$ formula like `Spec => Property` or as in the example, a TLAPS fact in the `ASSUME PROVE` form. Generally, the formulas are too complicated for the backend solvers to prove directly. Therefore, the proof should be separated into different steps. In TLAPS, a hierarchical, tree-like, structure is used, using `<i>` to indicate the proof level. A proof level can contain multiple steps, these are identified using `<i>j` where j denotes an identifier which might be an integer, but an alphanumerical sequence like `3a` is also accepted.

A step of level `<i>` might be decomposed into more steps in a new proof layer `<i+1>`. Like in the example, step `<1>2` is proven by a second proof level `<2>` in multiple steps. Since a proof level is created to prove a formula in the level above, the last step should prove that formula, also called the "goal". This is done using the `QED` keyword, which is a shorthand to avoid copying the formula to prove. (see line 23 of the example)

All leaf statements need to be proven by providing facts using the `BY` keyword. These facts can be previously proven steps inside of the current proof (line 24) or already proven theorems (see line 9). Another kind of fact, called "tactics", are indications on which backend solver to use when trying to prove a certain step. Facts can also be introduced into the current proof context using the `ASSUME` keyword. In the example, line 2 adds the facts that a variable S exists and that S is a finite set. These facts are present in each proof step, as they are introduced at the very beginning of the proof. Another way to introduce facts is the `USE` keyword. Facts introduced in that manner are usually assumptions of a specification, for example, that the number of processes is non-zero. Facts introduced using `USE` can be removed using the `HIDE` keyword. When a formula follows from all the facts already introduced in such a fashion, there might be leaf statements that do not require additional facts to be introduced using `BY`. In such cases, `BY` is replaced by the `OBVIOUS` keyword.

By default, definitions are not expanded in TLAPS. If we define `One` to be 1 and `Two` to be 2, TLAPS will not be able to prove `One + One = Two`; the definitions need to be expanded. This is done to avoid overwhelming backend solvers with possibly unneeded definitions. In the example, the definition of `IsFiniteSet(S)` is not known by the backend solvers. The unexpanded definition is sufficient to apply the different theorems about finite sets that are included by TLAPM (see Figure 2.4). To expand a definition in a leaf statement, the `DEF` keyword can be put at the end of the `BY`. If deemed appropriate, a definition can be expanded for a complete proof level using a `USE DEF` construct. Definitions can be added inside a proof, without being defined in the module, using the `DEFINE` keyword. An expanded definition can be hidden using a `HIDE DEF` construction.

Furthermore, TLAPS has many different constructs to organize proofs. The `SUFFICES` keyword is used to rewrite the proof goal in another form. More precisely, `SUFFICES F` indicates that F implies the current goal. In the example, lines 4 and 5 allow rewriting the proof of the implication at line 3 as proving the consequent by assuming the antecedent, simplifying the proof. Another keyword is `CASE` followed by a condition. It is a shorthand for `ASSUME condition PROVE QED`, which can make proofs more concise. The final keyword relevant for this thesis is the `PICK` keyword. It allows to reason about the existence of some elements without specifying the element. In line 7 of the example, the existence of an element inside a set needs to be proven, which is the exact situation `PICK` can be used. After proving that the element really exists inside a new proof level, the existence of the element can be used further down the proof.

```
 1  THEOREM FS_TwoElements ==
 2      ASSUME NEW S, IsFiniteSet(S)
 3      PROVE Cardinality(S) > 1 => \E x, y \in S: x # y
 4    <1> SUFFICES ASSUME Cardinality(S) > 1
 5                 PROVE \E x, y \in S: x # y
 6      OBVIOUS
 7    <1>1. PICK x : x \in S
 8      <2>1. Cardinality(S) > 0
 9        BY FS_CardinalityType
10      <2>2. QED
11        BY <2>1, FS_NonEmptySet
12    <1>2. PICK y : x # y /\ y \in S
13      <2> DEFINE T == S \ {x}
14      <2>1. /\ IsFiniteSet(T)
15            /\ Cardinality(T) = Cardinality(S) - 1
16        BY <1>1, FS_RemoveElement
17      <2>2. Cardinality(T) > 0
18        BY <2>1, FS_CardinalityType
19      <2>3. \E y : y \in T
20        BY <2>1, <2>2, FS_NonEmptySet
21      <2>4. QED
22        BY <2>3
23    <1>3. QED
24      BY <1>1, <1>2
```

Figure 2.2: The source of a short theorem used in the thesis.

THEOREM $FS\_TwoElements \triangleq$
  ASSUME NEW S, $IsFiniteSet(S)$
  PROVE $Cardinality(S) > 1 \Rightarrow \exists\, x,\, y \in S : x \neq y$
 $\langle 1 \rangle$ SUFFICES ASSUME $Cardinality(S) > 1$
            PROVE $\exists\, x,\, y \in S : x \neq y$
  OBVIOUS
 $\langle 1 \rangle 1$. PICK $x : x \in S$
  $\langle 2 \rangle 1$. $Cardinality(S) > 0$
   BY $FS\_CardinalityType$
  $\langle 2 \rangle 2$. QED
   BY $\langle 2 \rangle 1$, $FS\_NonEmptySet$
 $\langle 1 \rangle 2$. PICK $y : x \neq y \land y \in S$
  $\langle 2 \rangle$ DEFINE $T \triangleq S \setminus \{x\}$
  $\langle 2 \rangle 1$. $\land\, IsFiniteSet(T)$
      $\land\, Cardinality(T) = Cardinality(S) - 1$
   BY $\langle 1 \rangle 1$, $FS\_RemoveElement$
  $\langle 2 \rangle 2$. $Cardinality(T) > 0$
   BY $\langle 2 \rangle 1$, $FS\_CardinalityType$
  $\langle 2 \rangle 3$. $\exists\, y : y \in T$
   BY $\langle 2 \rangle 1$, $\langle 2 \rangle 2$, $FS\_NonEmptySet$
  $\langle 2 \rangle 4$. QED
   BY $\langle 2 \rangle 3$
 $\langle 1 \rangle 3$. QED
  BY $\langle 1 \rangle 1$, $\langle 1 \rangle 2$

Figure 2.3: The typeset version of the same theorem.

```
1  THEOREM FS_CardinalityType ==
2    ASSUME NEW S, IsFiniteSet(S)
3    PROVE  /\ Cardinality(S) \in Nat
4           /\ ExistsBijection(1..Cardinality(S), S)
```

Figure 2.4: The statement of the `FS_CardinalityType` theorem without proof. taken from the TLAPM source `https://github.com/tlaplus/tlapm/blob/main/library/FiniteSetTheorems.tla#L59`

Chapter 3

Reusable Barrier

3.1 Problem Setting

A barrier is a classical synchronization method in parallel computations. It must ensure that a group of processes arrives at the barrier before they can continue execution. Such a synchronization can be useful, for example, to allow processes to perform the first part of a computation, then wait for all processes to finish before being able to finish the computation because, for instance, the second part of the computation requires the results from all processes for the first part of the computation.

More formally and in all generality, considering N concurrent processes, the barrier synchronization will ensure that $m \leq N$ processes have reached the barrier before those m processes can proceed. Like the solution presented in the course [1], we will focus on the case where $m = N$. Furthermore, barriers are often used in loops such that the synchronization will happen repeatedly. The barrier implementation must allow repeated use, hence the name "reusable barrier".

3.2 Solutions

The first mechanism presented in the lecture notes includes a turnstile-like solution (analogy taken from the notes). Processes wait and then signal a binary semaphore to allow the other processes to pass. The semaphore is initially set to 0, and the last process entering the barrier signals the semaphore to let the waiting processes pass. Once all processes have passed that "turnstile", the semaphore still has value 1. To allow reusability, the last process leaving the barrier must wait on `Gate` to set it back correctly to 0. The pseudo-code for the solution can be found in Figure 3.1.

However, this implementation has a fatal flaw, since the first process to leave

```
   procedure Barrier()
1  |  lock.Lock()
2  |  RDV ← RDV + 1
3  |  if RDV = n then Gate.Signal()
4  |  lock.Unlock()
5  |  Gate.Wait()
6  |  Gate.Signal()
7  |  lock.Lock()
8  |  RDV ← RDV − 1
9  |  if RDV = 0 then Gate.Wait()
10 |  lock.Unlock()
```

Figure 3.1: First proposed solution found in [1].

the barrier can simply reenter that barrier and pass freely. Indeed, as once the `Gate` variable has been set to 1, processes are free to pass. To solve this issue, one needs to add a second independent gate, creating two sub-barriers. Since the first sub-barrier will be closed once the second is opened, a process leaving the (complete) barrier cannot pass through the whole barrier anymore. Another improvement to the design is the use of generalized semaphores instead of binary ones. The former enables exactly N processes to pass without the need to "close" the gate again once every process has passed through. The pseudo-code for this solution can be found in Figure 3.2.

```
   procedure Barrier()
1  |  lock.Lock()
2  |  RDV ← RDV + 1
3  |  if RDV = n then
4  |  |  Gate_1.Signal(n)
5  |  lock.Unlock()
6  |  Gate_1.Wait()
7  |  lock.Lock()
8  |  RDV ← RDV − 1
9  |  if RDV = 0 then
10 |  |  Gate_2.Signal(n)
11 |  lock.Unlock()
12 |  Gate_2.Wait()
```

Figure 3.2: Second proposed solution found in [1].

In the following, the barrier algorithm will be specified inside TLA$^+$, and the correctness of the barrier will be proven.

3.3 PlusCal Specification

The PlusCal specification of the reusable barrier of the `Barriers` module can be found in Figure 3.3.

First, the different variables are initialized to their respective values. Then, the `Lock`, `Unlock`, `Signal`, and `Wait` operations are defined as macros. To make sure that `Lock` and `Wait` are blocking, the keyword `await` is used. By awaiting that the lock variable must equal 1, we effectively disallow processes to advance unless the condition is met. The same reasoning applies inside `Wait`. Finally, the barrier implementation is described. The `process` keyword is used to define the number of processes to create. It is backed by the interval of integers $1..N$. To verify the reusability of the barrier, it is placed inside an infinite loop, and to simulate work unrelated to the barrier operations, a `skip` operation (a no-operation) is placed before the barrier.

As described in the previous section, the whole barrier is constructed using two sub-barriers. The variable `rdv` is used to count the processes that went through the barrier (label `a2`). The last process to pass, this is verified with a condition on `rdv` (label `a3`), will signal the `gate_1` semaphore (label `a4`), allowing all waiting processes to continue (label `a6`). Of course, the modification of the shared resource needs to be inside a critical section protected by `lock` (labels `a1` and `a5`). The second sub-barrier behaves in the same way, using `gate_2`.

3.4 Helper Inductive Invariants

This section will describe the simpler invariants used as building blocks for the main invariant

3.4.1 Typing Invariant

As TLA$^+$ is fundamentally untyped, the typing invariant `TypeOK` (Figure 3.4) specifies the value domains for every variable used in the specification. Using this invariant in other proofs allows backend solvers to restrict the types of the variables.

This invariant is inductive and has been proven as such using TLAPS in the `Typing` lemma. One can note that while `lock` could be proven to be in $\{0, 1\}$ at this stage, one cannot prove that `rdv` or the semaphores are in $0..N$. The typing invariant cannot itself inductively prove that rdv will not exceed N. For that, a more complete understanding of the algorithm is necessary.

3.4.2 Locking Invariant

Another simple property that can be verified at this stage is the mutual exclusion to the critical sections. The locking invariant (Figure 3.5) states that no two

processes can be inside a critical section, and if one process is in a critical section, the lock variable must be set to 0.

This invariant, together with the typing invariant, has been proven as such using TLAPS in the `LockExclusion` lemma. As mentioned in the previous section, the typing invariant is used to set the types of the different variables. In this particular case, it is used to tell TLAPS that the co-domain of `pc` must be a label of the algorithm and not any other value.

3.5 Main Invariant

The main invariant of the specification contains properties that need to hold to prove the correctness of the barrier.

3.5.1 Ghost variables

The invariant uses multiple ghost variables, variables not directly present inside the algorithm but that are related to the actual variables. In the invariant, three ghost variables are used. All are subsets of the set of processes based on values of `pc`. Their definitions can be found in Figure 3.6

- `ProcsInRdv` is the set of processes that are inside `rdvsection` at the current execution state, which is in between the increment and decrement of the `rdv` variable. Since `pc` represents the operation that is executed next, the labels that are inside the `rdvsection` are the ones strictly after `a2` and before (and including) `a8`.
- `ProcsInB1` is the set of processes that are inside `barrier1` at the current execution state, which is the first part of the algorithm. Label `a0` has been added to the first barrier as it behaves the same way as being at `a1`. Moreover, this choice also reflects the possibility of a process leaving the second barrier and trying to enter the barrier again.
- `ProcsInB2` is the set of processes that are inside `barrier2` at the current execution state, which is the second part of the algorithm.

Since these three sets are all subsets of `ProcSet`, they must be finite, and their cardinality must be bounded by the number of processes N. Since the number of processes does not change over time, the properties still hold after any action. These two facts are proven in the `ProcSetSubSetsBound` lemma and is used later.

3.5.2 Invariant description

The specification of the main invariant `Inv` can be found in Figure 3.7. Since the two sub-barriers are equal, except from their effect on `rdv`, many properties

about one will hold for the other, creating many "duplicated" clauses that state the same property for each barrier.

The first three clauses of the invariant are refinements of the typing invariant that could not be proven by themselves. Here we explicitly state that both semaphores `gate_1` and `gate_2`, as well as the rdv variable, are in $0..N$. The latter variable is related to the `ProcsInRdv` set in the invariant, the property can be proven using the lemma for the ghost variables. The fourth and fifth clauses state that processes must be waiting to leave a sub-barrier if the corresponding semaphore is strictly greater than 0. Indeed, in the algorithm, the semaphore is incremented once all processes have entered the sub-barrier, which gives an intuition as to why this property should hold. The sixth clause states that at least one barrier must be closed, alternatively, one can state that at most one barrier can be open. This clause matches the property described in Section 3.2. The seventh and eighth clauses ensure that the two barriers are mutually exclusive. Indeed, if at least one process is about to enter or is in the beginning of the second barrier, the start of the first barrier must be empty. Intuitively, since all processes must be ready to leave the first barrier to open `gate_1`, a process can only enter the second barrier if the beginning of the first barrier is empty. This must also hold the other way around, meaning that processes are at the start of the first barrier, the beginning of the second barrier must be empty.

However, these properties do not suffice to have an inductive invariant, additional clauses need to be added to make `Inv` inductive. The ninth and twelfth clauses indicate that the `gate_i` variables are bounded by the number of processes inside their respective sub-barrier. Since the number of processes in the first barrier decreases whenever `gate_1` is decreased and `gate_1` is increased to N whenever all processes are in the first barrier, this clause must hold. The same is true for the second barrier and `gate_2`. The tenth and thirteenth clauses describe that a process being at the entry of a barrier implies its gate being closed. Indeed, if one process exists at the start of a barrier, it means that not all processes are waiting at the corresponding gate, and therefore the gate must be closed. Finally, the eleventh and fourteenth clauses represent the intention between the `a4` and `a10` steps. It describes that if one process is at these labels, the condition in the `if` statements must be true, and all other processes must be waiting at the corresponding gate. Using all these properties, `Inv` can be proven inductive.

3.5.3 Proof details

To prove that an invariant is inductive, we must prove that it holds initially (proof step `<1>1.`) and that it stays true after each possible transition. (proof step `<1>2.`) Since the invariant contains many clauses, the backend solvers are not always able to prove the complete invariant at the same time. To prove that `Inv` holds initially, most clauses are proven without issues, but the clauses that involve the different introduced sets require theorems that involve

finite sets, which are shipped with TLAPS. For example, to prove that `rdv = Cardinality(ProcsInRdv)` holds initially, one needs to show that the cardinality of the empty set is 0. This fact is proven by the `FS_EmptySet` theorem.

Other theorems about finite sets used throughout the proof are :

- `FS_Interval`, an interval $a..b$ of integers is finite and its cardinality is $b - a + 1$
- `FS_AddElement`, the set that is the union of a finite set and an element is also finite, and its cardinality is the cardinality of the initial set, incremented if the element was not already present.
- `FS_RemoveElement`, the same as the previous theorem but for removing an element from a finite set.
- `FS_CardinalityType`, the cardinality of a finite set is a natural number

To prove that the invariant holds after each possible transition, the proof is again decomposed to first check if the invariant is verified after the execution of each label separately. Operations performed by labels `a0`, `a1`, `a5`, `a7`, `a11` having no effect on the properties inside `Inv`, the solvers are able to easily check the invariant validity. The other labels have at least one clause that the backend solvers cannot prove directly and/or require one of the previously mentioned lemmas or theorems about finite sets. In the following, the non-trivial proof steps for the remaining labels will be described. By symmetry, a non-trivial proof step for, e.g. `a8`, will be similar to the one for `a2` up to variable names.

For label `a2`, the `rdv` variable is increased, and due to TLAPS having difficulties handling cardinalities of sets, the `FS_AddElement` must be invoked to help the solver check that rdv = Cardinality(ProcsInRdv) stays true. Since the process executing `a2` will be entering `rdvsection`, the cardinality of set `ProcsInRdv` will indeed increase by one. For `a8`, since `rdv` is decreased, the dual theorem `FS_RemoveElement` is invoked instead.

For label `a3`, the solver cannot prove the validity of the eleventh clause on its own. Since the operation is a conditional, the proof will follow the two cases of going to `a4` or `a5`. In the case that `rdv` is equal to N, we can show that every other process must be waiting at `a6` by virtue of the critical section being mutually exclusive, all processes being in `rdvsection` and cannot be in the second barrier in virtue of the seventh clause of the invariant. Of course, if rdv = N, the process will move to `a4`, making the clause true. In the case that `rdv` is strictly smaller than N, the process will move to `a5` and by the mutual exclusion of the critical section, no other process can be in `a4`. Since this renders the precondition of the implication false for each process, the clause is also true in this case.

For label `a4`, the backend solver cannot directly see that the ninth clause still holds. By explicitly showing that all processes must be and stay inside `barrier1` by virtue of the eleventh clause, we can show that Cardinality(ProcsInB1) = N. Finally, using the tenth clause, we know that `gate_1` must be 0 when a process is about to execute `a4`, and the semaphore will be set to N after. Both variables being equal to N preserves the inequality, and the clause stays true.

For label `a6`, two clauses need more guidance to finish the proof. First, the ninth clause again cannot be proven directly. This is easily proven by showing `ProcsInB1` loses one element at the same time `gate_1` is decreased, preserving the inequality. Due to considering a set losing an element, the theorem `FS_RemoveElement` is added as justification for the solvers. Second, the fourth clause requires a more detailed proof. There are two cases, if `gate_1` becomes 0, the property is trivially verified. If `gate_1` stays strictly larger than 0, it means that `gate_1` was initially strictly greater than 1. Therefore, using clause 9, there must be at least two elements in `ProcsInB1`. Using the contrapositive of clause 10, we know that these two processes must be in either `a5` or `a6`. Since we are currently considering a process at `a6`, we have found a second process that will stay in `a5` or `a6`. Since both the antecedent and consequent are true, the clause is true.

With the TLAPS proof completely verified by the backend solvers, `Inv` is an inductive invariant.

3.5.4 More intuition strengthening properties

One of the properties required for the correctness of the barrier is that processes can only be in one sub-barrier as a whole. More particularly, we would like processes to all be inside the first barrier entry (and barrier waiting sections) or inside the second barrier entry (and barrier waiting sections). These facts can be proven easily as consequences of `Inv` and the typing invariant. See Figure 3.8 for the proof of two different representations of the statement presented above.

Since the property is a direct consequence of `Inv`, the solver can prove the theorem without any further guidance. The second version requires `TypeOK`, so that solvers understand the co-domain of `pc`. The co-domain is needed to complement the sets shown in the first version.

3.6 Refinement from an abstract barrier

3.6.1 Specification and refinement

More abstractly, a barrier should be a synchronization facility that waits for all processes to reach the barrier entry and then releases all processes simultaneously. Such a behavior cannot be described easily in PlusCal but can be

described in pure TLA. This is done in the `Barrier` module. (Figure 3.9)

From this specification, a refinement from the Reusable Barrier can be performed. Only the program counter variable needs to be translated from one specification to another. Since an abstract barrier requires a synchronous movement of processes and our barrier implementation allows processes to move independently, the translation for pc cannot depend on the program counter alone. A simple translation would need to involve the `gate` variables. The chosen translation can be found in Figure 3.10. The no-operation `a0` is chosen as `b0`, and nearly all the other labels correspond to `b1`. The last two labels `a11` and `a12` are chosen to be the barrier exit, and when `gate_2` is greater than 0, one can consider processes outside the barrier, else the processes are waiting and still considered inside the barrier.

3.6.2 Refinement proof

To prove that the proposed refinement is indeed valid, one can write a TLAPS proof. However, the proposed invariant `Inv` is not strong enough, and thus new clauses must be derived. The problem with the previous invariant `Inv` is that it does not describe the fact that once a gate is opened, it has the capacity to let all the processes leave. This property can be expressed in the invariant `FlushInv` (Figure 3.11) and can be proven to be inductive, using the previous invariants in theorem `FlushInvariant`.

The refinement proof follows quite succinctly from all the invariants presented. The only step requiring guidance is the last step `a12`, as one needs to split the proof into the case where `gate_2` becomes 0 and where it stays non-zero. The former is problematic, as if some processes remained at `a11` or `a12` when the variable becomes zero, their label would switch to `b1`. But using `FlushInv`, we can safely assume that once `gate_2` becomes zero, all processes must have advanced from `a12`.

```
--algorithm Barrier{
    variables
      lock = 1,
      gate_1 = 0,
      gate_2 = 0,
      rdv = 0 ;

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

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

    macro Wait( s ) {
        await s > 0 ;
        s := s − 1 ;
     }

    macro Signal( s ) {
        s := s + N ;
     }

    process ( proc ∈ 1 .. N ) {
a0:   while ( TRUE ) {
        skip ;  Some code
a1:     Lock(lock) ;
a2:     rdv := rdv + 1 ;
a3:     if ( rdv = N ) {
a4:       Signal(gate_1) ;
        } ;
a5:     Unlock(lock) ;
a6:     Wait(gate_1) ;
a7:     Lock(lock) ;
a8:     rdv := rdv − 1 ;
a9:     if ( rdv = 0 ) {
a10:      Signal(gate_2) ;
        } ;
a11:    Unlock(lock) ;
a12:    Wait(gate_2) ;
      }
    }
 }
```

Figure 3.3: The PlusCal specification of the Reusable Barrier.

$$
\begin{array}{l}
\mathit{TypeOK} \triangleq \\
\quad \land \mathit{lock} \in \{0, 1\} \\
\quad \land \mathit{gate\_1} \in \mathit{Nat} \\
\quad \land \mathit{gate\_2} \in \mathit{Nat} \\
\quad \land \mathit{rdv} \in \mathit{Int} \\
\quad \land \mathit{pc} \in [\mathit{ProcSet} \to \\
\qquad \{\text{“a0”}, \text{“a1”}, \text{“a2”}, \text{“a3”}, \text{“a4”}, \text{“a5”}, \text{“a6”}, \\
\qquad\quad \text{“a7”}, \text{“a8”}, \text{“a9”}, \text{“a10”}, \text{“a11”}, \text{“a12”}\}]
\end{array}
$$

Figure 3.4: The type invariant for the Reusable Barrier.

$$
\begin{array}{l}
\mathit{lockcs}(p) \triangleq \mathit{pc}[p] \in \{\text{“a2”}, \text{“a3”}, \text{“a4”}, \text{“a5”}, \text{“a8”}, \text{“a9”}, \text{“a10”}, \text{“a11”}\} \\
\mathit{ProcsInLockCS} \triangleq \{p \in \mathit{ProcSet} : \mathit{lockcs}(p)\}
\end{array}
$$

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

Figure 3.5: The locking invariant for the Reusable Barrier.

$$
\begin{array}{l}
\mathit{rdvsection}(p) \triangleq \mathit{pc}[p] \in \{\text{“a3”}, \text{“a4”}, \text{“a5”}, \text{“a6”}, \text{“a7”}, \text{“a8”}\} \\
\mathit{ProcsInRdv} \triangleq \{p \in \mathit{ProcSet} : \mathit{rdvsection}(p)\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{barrier}1(p) \triangleq \mathit{pc}[p] \in \{\text{“a0”}, \text{“a1”}, \text{“a2”}, \text{“a3”}, \text{“a4”}, \text{“a5”}, \text{“a6”}\} \\
\mathit{ProcsInB}1 \triangleq \{p \in \mathit{ProcSet} : \mathit{barrier}1(p)\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{barrier}2(p) \triangleq \mathit{pc}[p] \in \{\text{“a7”}, \text{“a8”}, \text{“a9”}, \text{“a10”}, \text{“a11”}, \text{“a12”}\} \\
\mathit{ProcsInB}2 \triangleq \{p \in \mathit{ProcSet} : \mathit{barrier}2(p)\}
\end{array}
$$

Figure 3.6: The definition of the three ghost variables used in the main invariant.

$Inv \triangleq$

the semaphore values are kept between 0 and N

$\land\ gate\_1 \in 0\,..\,N$

$\land\ gate\_2 \in 0\,..\,N$

rdv is the amount of processes in $]a2\ ;\ a8]$

$\land\ rdv = Cardinality(ProcsInRdv)$ proves that $rdv \in 0\,..\,N$

open gates mean that at least one process must be in the correct waiting section

$\land\ gate\_1 > 0 \Rightarrow \exists\, p \in ProcSet : pc[p] \in \{\text{“a5”}, \text{“a6”}\}$

$\land\ gate\_2 > 0 \Rightarrow \exists\, p \in ProcSet : pc[p] \in \{\text{“a11”}, \text{“a12”}\}$

at least one gate must be closed

$\land\ (gate\_1 = 0) \lor (gate\_2 = 0)$

if one process in the first barrier (or about to enter), then second barrier must be empty

$\land\ (\exists\, p \in ProcSet : pc[p] \in \{\text{“a0”}, \text{“a1”}, \text{“a2”}, \text{“a3”}, \text{“a4”}\})$
$\quad \Rightarrow \neg(\exists\, p \in ProcSet : pc[p] \in \{\text{“a7”}, \text{“a8”}, \text{“a9”}, \text{“a10”}\})$

if one process in second barrier, then first barrier must be empty

$\land\ (\exists\, p \in ProcSet : pc[p] \in \{\text{“a7”}, \text{“a8”}, \text{“a9”}, \text{“a10”}\})$
$\quad \Rightarrow \neg(\exists\, p \in ProcSet : pc[p] \in \{\text{“a0”}, \text{“a1”}, \text{“a2”}, \text{“a3”}, \text{“a4”}\})$

The value of $gate\_1$ is bounded by the count of processes in the first barrier

$\land\ gate\_1 \leq Cardinality(ProcsInB1)$

if one process arrives at the first barrier, the first gate is locked

$\land\ (\exists\, p \in ProcSet : pc[p] \in \{\text{“a0”}, \text{“a1”}, \text{“a2”}, \text{“a3”}, \text{“a4”}\})$
$\quad \Rightarrow gate\_1 = 0$

if one process is in $a4$, that means rdv is equal to N

and all other processes are waiting on $gate\_1$

$\land\ \forall\, p \in ProcSet : pc[p] = \text{“a4”} \Rightarrow ($
$\quad \land\ rdv = N$
$\quad \land\ \forall\, q \in ProcSet : (p \neq q) \Rightarrow pc[q] = \text{“a6”}$
$)$

The value of $gate\_2$ is bounded by the count of processes in the second barrier

$\land\ gate\_2 \leq Cardinality(ProcsInB2)$

if one process arrives at the second barrier, the second gate is locked

$\land\ (\exists\, p \in ProcSet : pc[p] \in \{\text{“a7”}, \text{“a8”}, \text{“a9”}, \text{“a10”}\})$
$\quad \Rightarrow gate\_2 = 0$

if one process is in $a10$, that means rdv is equal to 0

and all other processes are waiting on $gate\_2$

$\land\ \forall\, p \in ProcSet : pc[p] = \text{“a10”} \Rightarrow ($
$\quad \land\ rdv = 0$
$\quad \land\ \forall\, q \in ProcSet : (p \neq q) \Rightarrow pc[q] = \text{“a12”}$
$)$

Figure 3.7: The specification of the main invariant `Inv`.

```
THEOREM BarrierExclusion ≜
    Inv ⇒ ∨ ¬(∃ p ∈ ProcSet : pc[p] ∈ {"a0", "a1", "a2", "a3", "a4"})
          ∨ ¬(∃ p ∈ ProcSet : pc[p] ∈ {"a7", "a8", "a9", "a10"})
  BY N_Assumption DEF Inv

THEOREM BarrierExclusion2 ≜
    TypeOK ∧ Inv ⇒
      ∨ (∀ p ∈ ProcSet : pc[p] ∈
            {"a5", "a6", "a7", "a8", "a9", "a10", "a11", "a12"})
      ∨ (∀ p ∈ ProcSet : pc[p] ∈
            {"a11", "a12", "a0", "a1", "a2", "a3", "a4", "a5", "a6"})
  BY N_Assumption DEF TypeOK, Inv
```

Figure 3.8: The two different representation of barrier exclusion.

```
CONSTANTS
  N

VARIABLES pc

vars ≜ ⟨pc⟩

ProcSet ≜ (1 .. N)

Init ≜
  ∧ pc = [p ∈ ProcSet ↦ "b0"]

b0(self) ≜
  ∧ pc[self] = "b0"
  ∧ pc′ = [pc EXCEPT ![self] = "b1"]

b1 ≜
  ∧ ∀ p ∈ ProcSet : pc[p] = "b1"
  ∧ pc′ = [p ∈ ProcSet ↦ "b0"]

Next ≜
  ∨ ∃ p ∈ ProcSet : b0(p)
  ∨ b1

Spec ≜ Init ∧ □[Next]_vars
```

Figure 3.9: The specification of an abstract barrier.

```
pc_translation(self) ≜
  IF pc[self] ∈ {"a1", "a2", "a3", "a4", "a5", "a6", "a7", "a8", "a9", "a10"}
    THEN "b1"
  ELSE IF pc[self] = "a0"
    THEN "b0"
  ELSE IF gate_2 > 0
    THEN "b0"
  ELSE "b1"

B ≜ INSTANCE Barrier WITH pc ← [p ∈ ProcSet ↦ pc_translation(p)]
```

Figure 3.10: The refinement from the Reusable Barrier to an abstract barrier.

```
FlushInv ≜
  ∧ gate_1 > 0 ⇒ gate_1 = Cardinality(ProcsInB1)
  ∧ gate_2 > 0 ⇒ gate_2 = Cardinality(ProcsInB2)
```

Figure 3.11: The definition of `FlushInv`.

Chapter 4

Two-way Refinement using Auxiliary Variables

Refinement only states that all traces of the implementation are possible traces of the abstract specification.

One would want to also prove that all traces of the abstract specification are possible traces of the implementation. This would then prove that the two specifications are equivalent. This is difficult, since the implementation might contain more information than the abstract specification, or the implementation might take more steps, to consider non atomic read-and-writes for example.

To allow two-way refinement, one needs to add information to the abstract specification. This can be achieved by adding auxiliary variables. This methodology is well known and described in [9]. The same paper shows that adding auxiliary variables creates an equivalent specification to the original.

This section of the thesis aims to find a two-way refinement on the concrete example of an abstract lock and Peterson's algorithm.

4.1 The different kinds of auxiliary variables

[9] shows three kinds of auxiliary variables. History variables that remember the past, prophecy variables that guess the future, and stuttering variables that force a specification to take stutter steps. Prophecy variables are not needed for this thesis and are therefore not described further.

4.1.1 History variables

Such variables are used to remember past actions taken by an algorithm that it has forgotten, since they are not useful for the correctness. However this infor-

mation might be needed to achieve a refinement to a lower-level specification.

Since the history variable contains only information about the current and past state and does not influence the behavior, one can convince oneself that adding such a variable creates an equivalent specification.

4.1.2 Stuttering variables

As already mentioned, stuttering is an important part of TLA$^+$ specifications, even more so in refinements.

A low-level specification may take multiple steps to implement a single step of a high-level specification. To allow refinement in such cases, the high-level specification allows stuttering steps that do not change the values of the different variables.

To achieve the refinement from a high-level to a low-level specification, one needs to introduce this stuttering so that a possible trace for the low-level specification can be constructed. Using the `Stuttering` module that is presented in [10], one can easily add such stuttering variables. The `PostStutter` definition allows automatically adding a certain number of stutter steps after an action is taken. The number of steps is based on a finite set with total order, a decrement operation that must eventually reach the smallest element $\bot$, and an initial value.

4.2 Specifications

In this section, we describe the specifications, and in particular, we explain the addition of auxiliary variables as well as the different refinements.

4.2.1 `Lock` specification

This specification (Figure 4.1) models a lock using a single variable and an atomic acquisition of the variable. This will be our high-level implementation in the context of refinement. As it is a very high-level specification, proving mutual exclusion is easily done in the `MutualExclusion` theorem. A typing invariant is used to ease the proving process. The invariants are shown in Figure 4.2

4.2.2 `Peterson` specification

The `Peterson` specification (Figure 4.3) transcribes Peterson's algorithm as described in [1] and verifies the invariant presented (Figure 4.4). The usual typing invariant is also defined, proven, and used in the other proofs.

Using the invariant, one can easily find the refinement from `Peterson` to `Lock` (Figure 4.5). The translation of `pc` is done by mapping each action of `Peterson`

to an action in `Lock`. The only notable quirk of this refinement is that the value of `lock` cannot be instantly derived from the variables found within `Peterson`. The only way is to get a value for `lock` is to verify if any process is in the critical section. TLAPS can easily prove this refinement, as done in theorem `Refinement`. It suffices to break the proof down to each atomic step for the backend solvers to prove it.

4.2.3 `LockHS` specification

As described before, to create a refinement from the high-level `Lock` towards the low-level `Peterson` specification, auxiliary variables need to be added. Lock acquisition takes 3 atomic steps in `Peterson` while it takes only one in `Lock`, therefore a stutter variable that adds 2 stuttering steps needs to be added. Moreover, `Peterson` contains the `turn` variable that remembers which process' turn it is. The `Lock` specification does not have such a variable, and therefore a history variable needs to be added as well.

Adding the history variable

The history variable `h_turn` will contain the expected value of `turn` for the `Peterson` specification. To emulate the assignment of the `turn` variable, `h_turn` is modified during the first stutter step. (explained in Section 4.2.3) Otherwise, the history variable is unchanged. As a shorthand for steps that do not change the history variable, a `NoHistoryChange(A)` definition is created that simply adds an (UNCHANGED h_turn) clause to the given action A.

Adding the stuttering variable

The stuttering variable `s` is created, and the `Stuttering` module is imported to allow the usage of `NoStutter` and `PostStutter`. The former is a shorthand to specify that actions are not allowed to happen during stutter steps and they do not modify the stuttering variable. The latter is used to introduce the stuttering after an action is performed. When no stuttering occurs, the variable `s` is equal to $\top$. To start stuttering after executing the given action, `PostStutter` sets `s` to a counter `val` with the initial value `initVal`. This counter is decremented after each stuttering step, and when the last step is taken, meaning that the counter is equal to `bot`, `s` is set back to $\top$.

When multiple actions may use stutter steps, one needs to ensure only the correct stutter action ends or respectively continues the stuttering. This is even more important when using `PreStutter`, which executes the actions after the stuttering steps. This is achieved by saving additional context inside `actionId` and `ctxt` which represent the associated action and the possible variables (e.g. the process ID).

To ensure correctness of `LockHS`, the `Stuttering` module needed to be modified,

since the `ctxt` field is not verified at every stutter step. This could allow another process, not the one who started the stuttering, to decrement `val`. In particular, this has the negative side-effect of allowing another process to modify `h_turn`, which would break the refinement.

The addition is not a necessary modification, since there are multiple ways to fix the problem. The first way would be to add the process ID inside the `actionId` field, which is always verified. Another way would be to change `h_turn` using `s.ctxt` instead of self. These two other solutions were discovered after the writing of the proofs and were therefore not tested. The modified `Stuttering` module has the advantage of simplifying proofs, as described in the next sections.

4.2.4 Invariants used and lemmas

Two invariants have been introduced in this specification to ease the refinement proof. (Figure 4.7)

- `TypeOKHS` extends the typing invariant of `Lock` and adds the type of the auxiliary variables.
- `InvHS` contains two clauses. The first clause explicitly states the value of `pc` whenever a process is performing stuttering steps. The second expresses that the value of `h_turn` is fixed inside the critical section and after the correct stuttering step. For this invariant, the fact that `s.ctxt` contains the process ID greatly eased the description of the invariants

Four lemmas are detailed and proven before the refinement proof.

- `TypingHS` proves that `TypeOKHS` is indeed inductive. TLAPS could prove this easily by decomposing the proof towards the different sub-actions.
- `AddingVariables` proves that `LockHS` is indeed a refinement of `Lock`. This proof is mainly used to be able to prove that `LockInv` holds for `LockHS` as well.
- `MutualExclusionHS` proves that `LockInv` still holds, using the previous lemma and the `MutualExclusion` theorem of `Lock`
- `IndInvHS` proves that `InvHS` is indeed inductive. As `h_turn` is modified during stuttering steps, the proof needed to be split between the different possibilities of `s`. This splitting will be explained for the proof of the refinement.

4.2.5 Refinement towards `Peterson` and proof

The refinement from `LockHS` towards `Peterson` using the auxiliary variables is now quite straightforward. (Figure 4.8)

- The history variable `h_turn` was created and is updated to exactly equal `turn`
- The variable `pc` of `Peterson` can be derived from `pc` and the value of the stuttering variable `s`
- The variable `c` of `Peterson` is easy to compute once we have a way to compute `pc`, given the invariant `Inv` of `Peterson`

The proof for the refinement uses all the invariants previously mentioned and therefore all their proofs. The non-trivial parts of the proof correspond to the action(s) performed by `l1HS`. Like mentioned above, we needed to split the proof for each possible value of `s` and then show that it would correspond to an action of `Peterson`. As the definition of `PostStutter` is quite large, it is more efficient for the backend solvers to first extract all the necessary information from the premises and then work with this reduced set of clauses.

To exemplify, I will focus on the sub-proof where `s = [id|->"l1", ctxt |-> self, val |-> 2]`. This means that `LockHS` is performing the first stuttering step of `l1HS(self)`. From the premises, we can say that the current process is already in the critical section of `Lock` and that `pc` will not be changed (as we are stuttering). We can also easily derive the next value of `s` as the `val` field will be decremented from 2 to 1, as well as the new value of `h_turn` as it is within the first stutter step that `h_turn` is modified. From these extracted clauses and the different invariants, one can easily prove that this stuttering step exhibits the same behavior as the action `a2` of `Peterson`.

Since we need to show what happens for each possible value of `s`, we also need to consider the case where `s.ctxt` is not equal to `self`, which is impossible in this case. This is trivial with the additional clause in `PostStutter`, since we can also derive `s.ctxt = self` from the premises.

4.3 Two-way refinement

Having proven the refinement from `Peterson` to `Lock` and from `LockHS` to `Peterson`, we have proven that `Lock` and `Peterson` are indeed equivalent and can be used interchangeably.

To the best of my knowledge, such a proof had not been done on a concrete example in TLA$^+$ before, and might encourage further endeavors to use auxiliary variables as well as two-way refinements when necessary.

```
--algorithm Lock{
    variables lock = 1;

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

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

    process ( proc ∈ 1 .. 2 ) {
l0:   while ( TRUE ) {
        skip;  non-critical section
l1:     Lock(lock);
cs:     skip;  critical section
l2:     Unlock(lock);
      }
    }
}
```

Figure 4.1: The PlusCal specification of `Lock`.

$$
\begin{aligned}
&\mathit{TypeOK} \triangleq \\
&\quad \land \mathit{lock} \in \{0, 1\} \\
&\quad \land \mathit{pc} \in [\mathit{ProcSet} \to \{\text{“l0”}, \text{“l1”}, \text{“cs”}, \text{“l2”}\}]
\end{aligned}
$$

$$
\mathit{lockcs}(i) \triangleq \mathit{pc}[i] \in \{\text{“cs”}, \text{“l2”}\}
$$

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

Figure 4.2: The invariants of `Lock`.

```
Other(p) ≜ IF p = 1 THEN 2 ELSE 1

--algorithm Peterson{
    variables
      c = [self ∈ ProcSet ↦ FALSE],
      turn = 1 ;

    process ( proc ∈ 1 .. 2 ) {
a0:   while ( TRUE ) {
        skip ;
a1:     c[self] := TRUE ;
a2:     turn := Other(self) ;
a3:     await ¬c[Other(self)] ∨ turn = self ;
cs:     skip ;
a4:     c[self] := FALSE ;
      }
    }
}
```

Figure 4.3: The PlusCal specification of `Peterson`.

```
TypeOK ≜
  ∧ c ∈ [ProcSet → BOOLEAN ]
  ∧ turn ∈ ProcSet
  ∧ pc ∈ [ProcSet → {"a0", "a1", "a2", "a3", "cs", "a4"}]

lockcs(i) ≜ pc[i] ∈ {"cs", "a4"}
Inv ≜
  ∧ ∀ p ∈ ProcSet : c[p] ≡ pc[p] ∈ {"a2", "a3", "cs", "a4"}
  ∧ ∀ p ∈ ProcSet : pc[p] ∈ {"cs", "a4"}
      ⇒ (turn = p ∨ pc[Other(p)] ∈ {"a0", "a1", "a2"})
  ∧ ∀ i, j ∈ ProcSet : (i ≠ j) ⇒ ¬(lockcs(i) ∧ lockcs(j))
```

Figure 4.4: The invariants of `Peterson`.

```
pc_translation(label) ≜
  CASE (label = "a0") → "l0"
    □  (label ∈ {"a1", "a2", "a3"}) → "l1"
    □  (label ∈ {"cs"}) → "cs"
    □  (label ∈ {"a4"}) → "l2"

lock_translation ≜ IF ∃ p ∈ ProcSet : pc[p] ∈ {"cs", "a4"} THEN 0 ELSE 1

L ≜ INSTANCE Lock WITH
      pc ← [p ∈ ProcSet ↦ pc_translation(pc[p])],
      lock ← lock_translation
```

Figure 4.5: The refinement from `Peterson` to `Lock`.

```
Other(p) ≜ IF p = 1 THEN 2 ELSE 1

InitHS ≜ Init ∧ (h_turn = 1) ∧ (s = top)

Adding 2 stuttering steps after an l1(self)transition
Updating the history variable during the right stutter step
l1HS(self) ≜
  ∧ PostStutter(l1(self), "l1", self, 1, 2, LAMBDA j : j − 1)
  ∧ h_turn′ = IF s′ ≠ top THEN IF s′.val = 1 THEN Other(self)
                                             ELSE h_turn
                          ELSE h_turn

procHS(self) ≜
  ∨ NoStutter(NoHistoryChange(l0(self)))
  ∨ l1HS(self)
  ∨ NoStutter(NoHistoryChange(cs(self)))
  ∨ NoStutter(NoHistoryChange(l2(self)))

NextHS ≜ (∃ self ∈ 1 .. 2 : procHS(self))

SpecHS ≜ InitHS ∧ □[NextHS]⟨vars, h_turn, s⟩
```

Figure 4.6: The TLA specification of `LockHS`.

```
TypeOKHS ≜
  ∧ TypeOK
  ∧ h_turn ∈ 1 .. 2
  ∧ s ∈ {top} ∪ [id : {"l1"}, ctxt : {1, 2}, val : 1 .. 2]

InvHS ≜
  ∧ ∀ p ∈ ProcSet :
      ∧ IF s ≠ top THEN s.ctxt = p ELSE FALSE
      ⇒ pc[p] = "cs"
  ∧ ∀ p ∈ ProcSet :
      ∨ pc[p] = "l2"
      ∨ pc[p] = "cs" ∧ s = top
      ∨ IF s ≠ top THEN s.ctxt = p ∧ s.val = 1 ELSE FALSE
      ⇒ h_turn = Other(p)
```

Figure 4.7: The invariants of `LockHS`.

```
pc_translation(self, label, stutter) ≜
  CASE (label = "l0") → "a0"
    □  (label = "l1") → "a1"
    □  (label = "l2") → "a4"
    □  (label = "cs") → IF stutter = top THEN "cs"
                        ELSE IF stutter.ctxt ≠ self THEN "cs"
                        ELSE IF stutter.val = 2 THEN "a2"
                        ELSE IF stutter.val = 1 THEN "a3"
                        ELSE "error"
c_translation(alt_label) ≜ alt_label ∈ {"a2", "a3", "cs", "a4"}

P ≜ INSTANCE Peterson WITH
      pc ← [p ∈ ProcSet ↦ pc_translation(p, pc[p], s)],
      c ← [p ∈ ProcSet ↦ c_translation(pc_translation(p, pc[p], s))],
      turn ← h_turn
```

Figure 4.8: The refinement from `LockHS` to `Peterson`.

Chapter 5

Conclusion

The goal of this Master's thesis was to continue the effort of specifying and verifying the different algorithms presented in the Parallel Programming course given by Pascal Fontaine using the TLA$^+$ Toolbox.

First, the turnstile-like barrier presented in the lecture notes was translated to PlusCal. Using three invariants, it could be proven that each process could only be waiting at a single sub-barrier, and therefore processes respected the barrier property. To further reinforce this claim, an abstract barrier that moves all processes synchronously has been specified, and a refinement mapping from the turnstile barrier to this abstract barrier could be proven. This provides a formal proof that the algorithm shown in the lecture is correct.

Second, the equivalence of two lock implementations could be proven with the usage of auxiliary variables. Peterson's algorithm was specified in PlusCal, and a refinement mapping to an abstract lock could be achieved. By extending the abstract lock with a history and a stutter variable, a refinement mapping to Peterson's algorithm was also achieved. These two refinements prove that both algorithms are equivalent and can be used interchangeably.

Future work could focus on verifying other algorithms presented in the course, as well as continuing the work on the equivalence of different algorithms. This thesis proves the equivalence on a lock for two processes, but the course also presents the generalized Peterson's algorithm or Filter algorithm, which works for an arbitrary number of processes. One could try to show the equivalence of this algorithm with the abstract lock. Moreover, this thesis does not address liveness properties. Future work could analyze such properties of the different algorithms, in particular locks, under some fairness constraint.

Bibliography

[1] P. Fontaine. (2025) Info0912 parallel programming lecture slides.

[2] L. Lamport, "The temporal logic of actions," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 3, pp. 872–923, 1994.

[3] S. Merz, "On the logic of tla+," *Computing and Informatics*, vol. 22, no. 3-4, pp. 351–379, 2003.

[4] L. Lamport, *Specifying systems: the TLA+ language and tools for hardware and software engineers*, 2002.

[5] ——, "The pluscal algorithm language," in *International Colloquium on Theoretical Aspects of Computing*. Springer, 2009, pp. 36–60.

[6] K. C. Chaudhuri, D. Doligez, L. Lamport, and S. Merz, "A tla+ proof system," *arXiv preprint arXiv:0811.1914*, 2008.

[7] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[8] M. R.-I. J. Centre. (2008) Tlaps documentation. [Online]. Available: https://proofs.tlapl.us/doc/web/content/Documentation/Tutorial/The_example.html

[9] L. Lamport and S. Merz, "Prophecy made simple," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 44, no. 2, pp. 1–27, 2022.

[10] ——, "Auxiliary variables in tla+," *arXiv preprint arXiv:1703.05121*, 2017.