UNIVERSITY OF TRENTO

Department of Information Engineering and Computer Science

Master's Degree in
Computer Science

Final Dissertation

Proving the Filter-Lock correct in PVS

Supervisor
Prof. Paola Quaglia

Student
Alberto Ercolani

Academic year 2020/2021

Contents

Abstract

Concurrent software is characterized by complex actions interleaving and is consequently challenging to devise and prove correct. Formal proofs are subject to a variety of fallacies, ranging from rewriting to logical errors that represent additional difficulties. Theorem provers are formal tools whose purpose is to help the human demonstrator to construct an error-free proof. Theorem proving is inherently hard and requires the user to be acquainted with many details not required by other formal techniques. In this thesis, we approach theorem proving developing two complete formalizations where Peterson-Lock and Filter-Lock, two famous and well-known mutual exclusion algorithms, are proved correct. To achieve this result we employ the Input/Output Automaton formalism and PVS, an actively maintained industrial-strength theorem prover.

Chapter 1

Introduction

Concurrent software is hard to devise and prove correct, however sound ways to achieve these results are required to generate performant solutions. In this thesis, we are interested in the second point, that is, to learn how to effectively prove concurrent software correct. Theorem Proving (TP) is hard but comes with one major advantage: the user that proves some solution correct obtains insights about correctness, consequently learning sound ideas that can support new solutions.

TP is a structured synthesis process, namely, the user discovers and proves several properties that together support the correctness proof. We are interested in providing a proof by invariants, that is, those properties that hold in all reachable states of the model. This kind of proof is generally carried out by induction and is characteristic: the basis case reduces to show that the property holds in some initial state, the inductive step involves the preservation of the property that is assumed to hold in some generic state and is maintained by all transition steps. Invariant properties on software are in general mathematically shallow and preserved by the majority of the transitions. The prover must develop the proof manually and this approach is subject to a variety of fallacies: incomplete case analyses, wrong reasoning, rewrite errors, and incorrect steps unfolding. Hence, pencil-and-paper proofs tend to be subject to several mistakes that computer-aided verification can rule out, forcing the human prover to analyse all missing cases, rejecting incorrect logical developments, rewriting or type-checking formulae for the user and accepting tautologies only. The tool that supports this process is a mechanical theorem prover or Proof Assistant (PA), that is a software whose purpose is to ease proof construction while shielding the user from aforementioned mistakes. Mechanical TP can be fully automatic or interactive: in the first case the user declares the property and the computer searches for a proof, in the second case the user declares the property and provides the proof. We are interested in the second approach because it is a convenient way to analyse properties with the aid of the PA.

Formal and mechanical proofs are remarkably different and so is their development. Several works proving concurrent algorithms correct are publicly available [2, 9, 10, 14, 15, 22]; however, to the best of our knowledge, when proofs are provided these results are supported by mechanical proofs only. Mechanized proofs tend to be of little value to inexperienced users because they do not explain why the proof works and how the mechanical proof relates to its formal version. Also, the high amount of low-level details tends to distract the reader from the intuition that supports the argumentation. Therefore, the user having no experience with the use of some PA can not rely on any intuitive proof, not even sketched.

Our work serves the purpose to introduce the novice user to the use of the Prototype Verification System (PVS) [33] for proving that concurrent software satisfies Mutual Exclusion (ME). Cited works consider many safety properties, we restrict our discussion to ME only: it proves itself to be a formalization challenge. We formalize two well-known and famous algorithms that satisfy the property of mutual exclusion: Peterson-Lock and Filter-Lock [30]. The second is a generalized version of the first, so we consider them in this order. Our main target is Filter-Lock; we choose to formalize this algorithm because we deem it sufficiently simple from a conceptual perspective and sufficiently challenging from a formal one. To make our formalization as accessible as possible, we provide both the formal and mechanical version of our formalizations [17]. The user can inspect our formal proofs and compare them against their mechanical counterparts, this way, abstract elements are conveyed

and mechanical proofs are made intuitive. Our work does not focus on the actual operation of PVS: material provided by SRI International [11], NASA [5] and three operative manuals [33] are publicly available and cover this topic. The strategy we plan to follow to achieve the result maps on next chapters:

In chapter 2, we introduce the reader to the formalization discussing several topics: ME, the concurrency problem and the property we want to prove; the role of Model Checking (MC) and TP in this formalization effort; the pseudocode and correctness proofs of the two algorithms we prove correct; the Input Output Automaton (IOA) model, the formalism we employ; eventually, the formal system upon which PVS, the PA we use, is based.

In chapter 3, we cover preliminaries supporting both formalizations: we explain why we have chosen PVS; we discuss "invariant properties", the type of property we wish to employ; we provide the formalization for Peterson-Lock with the purpose to introduce the reader to the formalization, giving the change to familiarize with the reasoning required to formalize Filter-Lock. Finally, we explain six mechanical strategies that aid us in the proof construction of both formalizations. At this point the user should be able to compare formal and mechanical proof scripts for the Peterson-Lock.

In chapter 4, we develop the main formalization challenge: we show and discuss the IOA modelling Filter-Lock and we introduce the reader to the set of intermediate results that support the ME proof. Eventually, we state and prove each property.

In chapter 5, we gather the conclusions we have drawn by the completion of the present work, in that chapter we discuss our experience with PVS, we discuss the role of MC and sketch a strategy for becoming effective at TP.

Chapter 2

State of the Art

Concurrent algorithms are difficult to prove correct because hand-written proofs are subject to many errors that computer-aided verification can rule out. We are interested into gaining experience in mechanical verification of concurrent algorithms, we do so formalizing and proving correct two simple and well-known mutual exclusion algorithms. Our formalization effort takes advantage of the Input Output Automaton (IOA) model and the Prototype Verification System (PVS).

In section 2.1, we describe Mutual Exclusion (ME), the coordination mechanism implemented by the two algorithms we prove correct, Peterson-Lock and Filter-Lock. We recall and discuss the algorithms in question, we provide their pseudocode and comment it.

In section 2.2, we describe the role of formal verification. We first compare advantages and disadvantages of the two major analysis methods, Model Checking (MC) and Theorem Proving (TP), then we contextualize the reader about modern TP. Eventually, we explain the additional difficulties the employment of a theorem prover involves and means to support TP with MC.

In section 2.3, we describe the IOA model, the formalism we adopted during our verification effort; we provide notation, definitions and an example model.

In section 2.4, we describe Sequent Calculus (SC), the proof system upon which the theorem prover we use is based. We first expose its formal syntax, its inference rules and explain how their usage generates proofs, then we provide an example proof to show how inference happens, and point out the structuring of proofs appearing in next chapters.

2.1 Mutual Exclusion

In this section, we first describe ME the classical concurrency problem and the safety property we wish to verify [25, pp. 261-262]. Secondarily, we recall the two algorithms we prove correct, Peterson-Lock and Filter-Lock. For each algorithm we explain how it guarantees ME and the connection to the other, we provide its pseudocode and comment it.

ME is a form of coordination in concurrent systems, its purpose is to protect a shared resource from unregulated access. It has been introduced and formalized for the first time by Edsger Dijkstra in 1956 [13]. Suppose there exists a shared resource which is able to support a bounded number of concurrent accesses k; suppose there exist a finite set of processes $\mathbb{P}$ that compete to access such resource; assume it is unknown when and which processes actually try to access; assume that all processes exhibit cyclic behaviour and communicate via shared variables. ME guarantees that the number of concurrent accesses never exceeds k.

Processes coordinating via this mechanism are characterized by four steps, each one representing a code fragment subject to some requirement:

1. The Non-critical Section (NCS) fragment represents the business logic of the software making use of the resource. It is not required to terminate.
2. The Entry fragment is responsible for regulating the access to the resource, it is always executed before the Critical Section (CS) fragment.
3. The CS fragment is responsible for making use of the resource. It is always executed before the Exit fragment and must be guaranteed to terminate.

4. The Exit fragment is responsible for abandoning the CS. It must be wait-free, namely, the process executing it must be able to terminate in a bounded number of *its own* steps.

One last requirement regards all variables that Entry and Exit fragments exploit in the regulation of the resource: aside from the program counter, none of them can be modified during the execution of NCS or CS.

Those algorithms that protect shared resources and comply to aforementioned requirements are said to satisfy mutual exclusion. In this context, we are interested in the instance of ME having $k = 1$; the first solution to this problem has been devised by Edsger Dijkstra [13]. When a new algorithm is devised, the challenge lies into designing Entry and Exit fragments in such a way that the property is satisfied. By ME we also refer to the safety property such algorithms satisfy.

Definition 2.1.1 (Mutual Exclusion)**.** At most one process is in CS at a time. Formally:

$$\forall p, q \in \mathbb{P} : \neg(pc_p = CS \wedge pc_q = CS \wedge p \neq q)$$

We now turn to the mutual exclusion algorithms we prove correct, they are due to Gary Peterson that formalized them in 1981 [30]. The first one is known as Peterson-Lock and solves the problem of ME for two processes, the second one is known as Filter-Lock and solves ME for arbitrarily many processes. Filter-Lock generalizes Peterson-Lock, consequently they are based on the same principles; we face the specific solution first and its generalization only later.

2.1.1 Peterson-Lock

Peterson-Lock is a mutual exclusion algorithm solving ME for two processes, we assume each process is associated to a unique ID drawn from $\{0, 1\}$.

Processes running Peterson's algorithm compete to enter in CS. When one process tries to gain access alone it always succeeds. When both processes compete, only one enters while the other is forced to wait until the former has not finished to use the resource and leaves CS. This way the algorithm guarantees ME.

Shared variables *level* and *victim* are used to compete for CS: each process engaging the competition first sets its *level* variable to *true* and then marks itself as *victim*. Variable *level* is used to communicate to the other process the interest to participate in the competition. Variable *victim* is used by one process to postpone the entry in CS and to allow the other to step into it.

Algorithm 1 Peterson-Lock

```
1: global natural victim                                  ▷ Initially arbitrary.
2: global boolean level[2]                                ▷ ∀p ∈ {0,1} : level[p] = false.
3: procedure PETERSON-LOCK(p ∈ [0,1])
4:     level[p] = true                                    ▷ Entry
5:     victim = p                                         ▷ Entry
6:     while level[1 − p] ∧ victim = p do {}              ▷ Entry
7:                                                        ▷ CS
8:     level[p] = false                                   ▷ Exit
```

Let us identify processes by p and $1 - p$, when p executes line 6 it reads variables $level[1 - p]$ and $victim$, hence it can witness three scenarios:

1. $1 - p$ has not engaged the competition, then p enters CS,
2. $1 - p$ has engaged the competition and is not the victim, then p must wait,
3. $1 - p$ has engaged the competition and is the victim, then p enters CS.

When both processes execute line 6, the process marked as victim loses the competition and must wait, while the other wins and enters CS. Once the winner of the competition is finished with the resource, it resets the *level* variable to $false$ and leaves CS; hence, the loser process can in turn enter.

2.1.2 Filter-Lock

Filter-Lock generalizes Peterson-Lock, but it is sufficiently different to represent a formalization challenge. It solves ME for arbitrarily many processes, we assume each process is associated to a unique ID draw from $\{0, \ldots, |\mathbb{P}| - 1\}$.

Filter-Lock exploits the same principle employed by Peterson-Lock: when two or more processes compete to enter in CS, the one playing the role of victim loses and waits, the rest wins the competition.

The intuition supporting Filter-Lock is to replicate $|\mathbb{P}| - 1$ times the mechanism employed by Peterson-Lock, generating $|\mathbb{P}| - 1$ levels. Levels are basically waiting rooms, inside which the loser of each competition must wait. The loser process is said to be filtered out, consequently each level is a filter. Processes that win the competition at level j qualify for the competition at level $j + 1$, where filtering is repeated. All processes competing for the CS must traverse all levels starting from level 0. Since each level hosts at most one loser process, the competition on level j is engaged by at most $|\mathbb{P}| - j$ competitors. Consequently, level $|\mathbb{P}| - 1$ hosts at most one winner: that level corresponds to CS. This way the algorithm guarantees ME.

Shared variables $level[p]$ and $victim[j]$ are used to compete for CS: each process engaging a competition on some specific level first sets variable $level$ and then marks itself $victim$ of that level. Variable $level[p] = j$ is employed by process p for competing in all competitions up to j; in fact, levels are cumulative: each process competing in level $j+1$ competes in level j as well. Variable $victim[j] = p$ is exploited by process p for communicating to competitors the intention to postpone its access to level $j + 1$.

Algorithm 2 Filter-Lock

```
1:  global natural victim[|ℙ|]                                   ▷ Initially arbitrary.
2:  global natural level[|ℙ|]                                    ▷ ∀j ∈ ℕ : (0 ≤ p ≤ |ℙ| − 1) ⇒ level[p] = 0.
3:  procedure FILTER-LOCK(p ∈ ℕ : p < |ℙ|)
4:      i = 1                                                    ▷ Entry
5:      while i < |ℙ| do                                         ▷ Entry
6:          level[p] = i                                         ▷ Entry
7:          victim[i] = p                                        ▷ Entry
8:          while (∃q ≠ p : level[q] ≥ i) ∧ victim[i] = p do {}  ▷ Entry
9:          i = i + 1                                            ▷ Entry
10:                                                              ▷ CS
11:     level[p] = 0                                             ▷ Exit
```

Analogously to what happens to Peterson-Lock, when some process p executes line 8 it reads $victim$ and scans competitors' $level$. Each process scans all competitors one at a time; let q be the k-th process scanned by p, let i be the level inside which p competes, then p can witness three scenarios:

1. No process has engaged the competition on p's level, then p enters the next level.
2. p and q have engaged the same competition and p is the victim, then it waits.
3. p and q have engaged the same competition and p is not the victim, then it enters the next level.

Alternatively:

1. $\forall q \neq p : level[q] < i$, then p enters the next level.
2. $level[q] \geq i$, $victim[i] = p$, then p waits.
3. $level[q] \geq i$, $victim[i] \neq p$, then p enters the next level.

When process p abandons CS it lowers its competition level to 0, allowing some other process to access the shared resource. It might not be apparent but the behaviour of processes is very aggressive. In fact, they try to enter in CS overcoming each other.

2.2 Theorem Proving

In this section we discuss the importance of TP by comparing its advantages and disadvantages with respect to MC. We discuss the additional difficulties the usage of TP involves and how it can cooperate with MC to allow fast algorithmic verification.

The purpose of formal methods is to verify the correctness of models at stake; however, manually verifying real-world models is generally impractical. This is why computer-aided formal methods have been introduced, they serve the purpose of mechanizing repetitive and error-prone activities leaving the human expert in charge to prove correctness. There exist two major methods, MC and TP.

MC is often regarded as a *light* heavy-weight formal method due to its capability to automatically check properties and its relatively light requirements over the user. Provided a *finite* model and a property are given, MC is concerned with the identification of a Counter Example (CE) (if any). This has been possible thanks to the adoption of decidable First Order Logic (FOL) fragments, like Temporal Logics (TLs) that are sufficient for safety and liveness properties checking [16][24, pp. 7,17]. Once the user is acquainted with the formal syntax, semantics and the proof system of the model checker, fast prototyping becomes possible. MC suffers two major limitations: finiteness of the model and limited scaling. The latter is due to the state space explosion problem [8, pp. 10-11].

TP is a heavy-weight formal method and a *structured synthesis* process, whose purpose is proving some formula valid. During formalizations, properties are identified and proved in isolation with the only possibility to employ previously proved theorems or axioms. Consequently, the prover is responsible for synthesizing relevant properties one by one hierarchically. TP is known to be more demanding than MC but overcomes its shortcomings. Model checkers can check an algorithm is correct, provided at most n processes operate concurrently; on the other hand, theorem provers can prove an algorithm correct for any n. It is undeniable that TP is crucial for understanding why software exhibits some property. In general, the user that reads or provides the proof has a clear explanation of the facts that force the software to behave correctly. Consequently, allowing the user to analyse and generate new algorithms based on sound ideas.

Initial research in the computer-aided proof generation field has been devoted to Automated Theorem Proving (ATP), that is, TP oriented to fully automatic solutions. In this context, mechanical theorem provers were employed either to decide the validity of theorems or for checking the correctness of their proofs. The FOL Resolution Principle, developed by Robinson, is an example of ATP procedure [32]. Automath is an example of proof consistency verifier, the human prover described the proof using some formal syntax and the computer was responsible for checking it [12][18, p. 139].

Parallel researches focused on Interactive Theorem Proving (ITP), that is, forfeiting automation and putting the human prover in charge to drive the construction of the proof. This choice has been motivated by two relevant limitative results in logic and computability theory, and practical difficulties that TP poses. Gödel's Incompleteness Theorem and Turing's undecidability of the halting problem discouraged the employment of ATP [6],[18, pp. 135-170]. Not only theoretical limitations were present but also practical: algorithms deciding the validity of Propositional Logic (PL) formulae incur into exponential blow ups in the amount of required resources; consequently, validity checking under more powerful logics can not be any easier.

Modern ITPs are more informally referred to as Proof Assistants (PAs), they are heterogeneous tools [34, p. 11], designed to help the human prover to mechanically construct and check the proof. To further support the user, additional features are often included and encompass: proof details management, supporting lemmata dependency tracking and automatic generation of handy logic patterns. Some PAs provide additional functionalities like access to model checkers and binary decision diagrams [29, 33]. Major PAs are equipped with a common core functionality: the support of user-defined strategies, that allow to repeat mechanical steps conditionally via some suitable scripting language. By custom strategies the PA can discard automatically tedious or repetitive proof steps.

Employing a PA is more difficult than using a MC due to many additional details the user must be aware of. First of all, PAs are based on proof systems that in turn are based on some logic. Consequently, proof systems require the user to be acquainted with their inference rules and their logic. Proof systems in question can be SC or Natural Deduction (ND), while the logic is often Higher Order Logic (HOL) or High Order Intuitionistic Logic (IL) (HOL without the *tertium non datur* axiom

[28, Section 1]). Secondly, feedbacks provided by PAs are not as rich as those provided by MCs. In fact, a failed proof attempt could be due to a missing lemma or a wrong strategy. Even in case a CE to the property exists, it could take some time to reach the branch where it is expressed, but the real hardship stems from making it apparent through the generation of an unprovable goal. Additional difficulties come from libraries shipped with the PA. Users can employ theorems declared in the local knowledge base of the tool; despite the collection is ordered it is weakly structured. The user can do very little to find handy theorems aside from searching them by name inside named theories. Finally, being an expert prover acquainted with the use of some specific PA does not guarantee the ability to be as proficient with another one.

We do not state that MC is superior to TP or the converse. We point out that each one is a suitable solution to a suitable problem. In fact, MC can relatively quickly convince the user that some software does not exhibit perceivable defects, but the user will not know exactly why this is the case. The user employing the MC could deduce the cause of the lack of errors but no apparent reason is ever provided, neither it is obviously tested. Conversely, PAs can not convince the human prover that some given module is correct, informally speaking, it is duty of the user to convince the mechanical prover that this is the case. MC and TP are different but can proficiently cooperate to allow relatively fast algorithm verification. Provided an algorithm is deemed correct, the prover can mix MC and TP; in [9, 10, 15] a methodology for mixing them is sketched. Firstly, the prover should be aware of the motivations that render the algorithm correct and high-level invariants should be prepared a priori and broken down into a set $\mathcal{I}$ of low-level invariants. Secondly, the algorithm should be modelled into the model checker's description language and each invariant $i \in \mathcal{I}$ should be checked for significant bounds (if bounded variables play some role). In such a way the user will save time and focus on correct properties. Failed proof attempts of invariants satisfied by the model are due to wrong strategies or missing provable lemmata. Consequently, if the user is aware that some invariant is CE free, he/she is able to direct the effort in discovering dependent lemmata or correcting the strategy. Once the prover is confident about the correctness of invariants, the TP direction could be regarded as a matter of taste. The topic is discussed further in chapter 5.

2.3 The Input Output Automaton Model

In this section we describe the IOA model, the formalism we employed in our formalization effort. We first discuss its role, its purpose and capabilities, secondly, we provide definitions and an example automaton for clarifying the description language and the qualities of the formalism.

The IOA model has been introduced by Nancy Lynch and Mark Tuttle in 1989, it emerged as a convenient formalism for timed dynamic systems modelling [26], and it has been widely employed to reason about distributed systems and concurrent algorithms [2, 9, 10, 14, 15, 20].

The IOA model is bare but general, it can encode: deterministic, non-deterministic, timed, untimed, concurrent and sequential systems. In fact, the lack of structure is not a limitation to its expressive power; the model is syntactically compact and extensions can be described in terms of basic structures [25, p. 199]. IOA can model different types of software ranging from simple concurrent algorithms to complex distributed systems. This is possible by virtue of composition: like in real-world scenarios, large complex systems are modelled through composition of submodules. Each component is singularly designed and proved correct, therefore correct complex systems are achieved by composition. This design choice fosters the reuse of submodules and renders the formalism suitable to reason about actual systems.

IOA distinguish from classic automata in many aspects. First of all, they do not simply compute a function and halt, they are intended to execute and never stop. Secondly, unlike classic automata that receive the input and compute a function over it, IOA are *input-enabled*, namely they can receive input from the environment at any time. Unlike classic automata, when considering IOA we are not interested in their language, rather in their executions. Finally, their state set is possibly infinite.

Input enabled-ness is satisfied when the automaton is unable to reject any input. This property requires the designer to define the automaton in such a way that any possible input is managed.

An IOA is basically a Labelled Transition System (LTS), equipped with state variables, edges

labelled by actions and an interface for communicating with the environment. Labelling actions are typed and range over *input*, *internal* and *output*. Input actions are responsible for instantaneously fetching the input from the environment and pass it to internal actions. Internal actions are intermediate and their output is not observable by external observers. Finally, output actions are responsible for instantaneously transmitting the output to the environment which, through them, becomes visible externally. IOAs are described in terms of their interaction with the environment, and as such they require an interface for doing so. The interface is called "signature", it specifies all action sets the IOA is equipped with, and which actions are responsible for fetching or transmitting data from and to the environment.

We provide definitions and conventions we will employ in next chapters. Since, we are interested in proving single modules correct, we do not involve details regarding composition or executions: additional information about these topics can be found in [26][25, pp. 199-255]. An LTS is a 4-tuple $T = (Q, I, \Sigma, \delta)$ whose elements represent, respectively, the set of states, the set of initial states, a finite alphabet and the transition function. $I \subseteq Q$ is defined non-empty; Σ is the labelling alphabet, in the IOA's context it is the action set; $\delta \subseteq (Q \times \Sigma \times Q)$ is a total transition function. Transition steps $(s_0, \pi, s_1) \in \delta$ are denoted by $s_0 \xrightarrow{\pi_p} s_1$, we refer to s_0 and s_1 as "pre-state" and "post-state", respectively; the subscript associated to action π identifies the process executing it, in all proofs, we identify the generic executor by $proc(\pi)$. For convenience, we denote by $input(\Sigma)$, $internal(\Sigma)$ and $output(\Sigma)$ respectively, Σ's pairwise disjoint subsets whose elements are either input, internal or output actions. The signature is a 3-tuple denoted $\mathcal{S} = \big(input(\Sigma), internal(\Sigma), output(\Sigma)\big)$. An IOA is therefore a pair $\mathcal{A} = (\mathcal{S}, T)$ where $\mathcal{S}$ is a suitable signature and T is an LTS.

IOA are described by standard notation introduced in [26] to which we comply; they are described symbolically in "precondition-effect" style. Transition steps are executed only when their precondition is satisfied. The process that takes a step instantaneously evaluates the precondition and in case it evaluates to true, atomically and instantaneously executes its effect; consequently all actions are instantaneous and atomic. Moreover, effects specify only which variables are subject to modifications and how they are modified.

We provide a demonstrative IOA implementing a mutual exclusion algorithm; similar examples are provided in [25, p. 310] and [14, p. 34]. We assume the existence of a set of concurrent processes $\mathbb{P}$, a set of program counter values $PCs = \{pcIdle, pcLock\_do, pcUnlock\_do, pcLock\_ret\}$, and define

$$Lock = \Big(Signature(Lock), \big(Q(Lock), I(Lock), \Sigma(Lock), \delta(Lock)\big)\Big)$$

Lock is a simple mutual exclusion algorithm; the regulation of the shared resource happens by atomic read and write of a single shared variable that specifies which process is currently owner of the resource. We proceed to describe an IOA modelling it:

- $Signature(Lock)$:
  - $input(Lock) = \{lock\_invoke_p : p \in \mathbb{P}\}$
  - $internal(Lock) = \{lock\_do_p, unlock\_do : p \in \mathbb{P}\}$
  - $output(Lock) = \{lock\_response_p : p \in \mathbb{P}\}$

The signature points out that the concurrent system contains $|\mathbb{P}|$ processes, each one responsible for executing some action. Processes are identified by unique natural IDs, the usage of the subscript clarifies which process executes $\pi \in \Sigma(Lock)$. States of an IOA are denoted as ordered tuple and correspond to the Cartesian product of state variables:

- $Q(Lock) = lock\text{-}holder \times pc$

Variables are typed as follows: $lock\text{-}holder \in \mathbb{N} : lock\text{-}holder \leq |\mathbb{P}|$ and $pc : \mathbb{P} \to PCs$. States of the *Lock* automaton involve one natural variable and program counter values for all processes in the system. The natural variables records the ID of the owner of the CS, the program counter is responsible for enforcing the program order. Finally, we define initial states of the system and the set of actions:

- $I(Lock) = \big\{q \in Q(Lock) : \big(lock\text{-}holder = |\mathbb{P}| \wedge (\forall p \in \mathbb{P} : pc_p = pcIdle)\big)\big\}$

- $\Sigma(Lock) = \{lock\_invoke, lock\_do, unlock\_do, lock\_response\}$

In table 2.1, we present *Lock*'s transition function.

$\delta(Lock)$	Comment:
lock_invoke$_p$:	Invocation.
pre: $pc_p = pcIdle$	
eff: $pc_p = pcLock\_do$	
lock_do$_p$:	Performs lock.
pre: $lock\text{-}holder = \lvert\mathbb{P}\rvert \wedge pc_p = pcLock\_do$	
eff: $lock\text{-}holder = p \wedge pc_p = pcUnlock\_do$	
unlock_do$_p$:	Performs unlock.
pre: $pc_p = pcUnlock\_do$	
eff: $lock\text{-}holder = \lvert\mathbb{P}\rvert \wedge pc_p = pcLock\_ret$	
lock_response$_p$:	Response.
pre: $pc_p = pcLock\_ret$	
eff: $pc_p = pcIdle$	

Table 2.1: Transition function, $\delta(Lock)$.

The behaviour of the automaton is extremely simple, when variable *lock-holder* is set to $|\mathbb{P}|$, the critical section is free. As soon as some process witnesses $lock\text{-}holder = |\mathbb{P}|$, it atomically overwrites the variable with its ID, consequently locking other processes out. The process performing the unlock operation simply resets the shared variable to $|\mathbb{P}|$.

2.4 Sequent Calculus

In this section, we illustrate the proof system employed by PVS, we provide an intuitive explanation to its inference rules, we clarify the relations between such rules and basic PVS commands. We provide one demonstrative proof to clarify three facts: how inference happens, how proofs are structured in SC, and how they relate to their mechanical counterparts. Eventually, we expose the style we employ in proofs appearing in next chapters.

SC is a logical argumentation style and a proof system developed by Gerhard Gentzen in 1934 [19]. Proof systems are based on three characterizing sets:

- Language: well-formed formulae admitted by the system.
- Inference rules: rewrite rules employed to prove theorems from axioms and existing theorems.
- Axioms: formulae assumed to be valid from which all theorems depend.

SC' language admits formulae of the shape:

$$A_1, A_2, \ldots, A_n \vdash B_1, B_2, \ldots, B_m \tag{2.1}$$

The turnstile symbol ($\vdash$) separates premises (assumptions) from conclusions (propositions), it is read "yields" and its role is the one of implication. Formulae A_i, B_i are in turn HOL formulae. Premises are in conjunction, while conclusions are in disjunction:

$$\begin{gathered} A_1, A_2, \ldots, A_n \vdash B_1, B_2, \ldots, B_m \\ \equiv \\ \vdash (A_1 \wedge A_2 \wedge \cdots \wedge A_n) \Rightarrow (B_1 \vee B_2 \cdots \vee B_m) \end{gathered}$$

Formulae in SC are called "sequents" and are characterized by having $n \geq 0$ premises and $m \geq 0$ conclusions. The sequent encoded by the turnstile alone, $\vdash$, is a syntactically valid sequent formula but it evaluates to trivially false and is consequently unprovable. By "antecedent" we refer to some formula on the left of the turnstile symbol (A_i in eq. (2.1)), conversely "consequent" identifies some formula on its right (B_i in eq. (2.1)). We denote by Γ and Δ respectively, finite sequences (or multi-sets) of antecedents and consequents.

SC is characterized by relatively few inference rules, reported in fig. 2.1. They are partitioned into two categories, *logical* and *structural*. In turn, logical rules are sub-partitioned into *propositional* and *quantifier*. We choose to further sub-partition quantifier rules into *introductory* and *instantiating* rules. Inference happens in a bottom-up fashion, the sequent to prove is the initial goal and is depicted under the inference line, the result obtained represents one or more secondary goals, depicted above. Single rules are sufficient to justify simple logical developments, however understanding how they relate to each other and how they can actually prove a theorem is not obvious. Therefore, we provide an intuitive explanation for each rule.

$\dfrac{}{\Gamma, A \vdash B, \Delta}\ (Ax),\ \text{if } A \equiv B$	$\dfrac{\Gamma \vdash \Delta, A \quad A, \Gamma \vdash \Delta}{\Gamma \vdash \Delta}\ (Cut)$	propositional, logical
$\dfrac{\neg A, \Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}\ (\neg L)$	$\dfrac{\Gamma \vdash \neg A, \Delta}{A, \Gamma \vdash \Delta}\ (\neg R)$	propositional, logical
$\dfrac{A \wedge B, \Gamma \vdash \Delta}{A, B, \Gamma \vdash \Delta}\ (\wedge L)$	$\dfrac{\Gamma \vdash \Delta, A \quad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \wedge B}\ (\wedge R)$	propositional, logical
$\dfrac{A, \Gamma \vdash \Delta \quad B, \Gamma \vdash \Delta}{A \vee B, \Gamma \vdash \Delta}\ (\vee L)$	$\dfrac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A \vee B}\ (\vee R)$	propositional, logical
$\dfrac{\Gamma \vdash \Delta, A \quad B, \Gamma \vdash \Delta}{A \Rightarrow B, \Gamma \vdash \Delta}\ (\Rightarrow L)$	$\dfrac{A, \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \Rightarrow B}\ (\Rightarrow R)$	propositional, logical
$\dfrac{A[x \leftarrow x'], \Gamma \vdash \Delta}{\exists x.A, \Gamma \vdash \Delta}\ (\exists L)$	$\dfrac{\Gamma \vdash \Delta, A[x \leftarrow x']}{\Gamma \vdash \forall x.A, \Delta}\ (\forall R)$	intr., quantifier, logical
$\dfrac{A[x \leftarrow \dot{x}], \Gamma \vdash \Delta}{\forall x.A, \Gamma \vdash \Delta}\ (\forall L)$	$\dfrac{\Gamma \vdash \Delta, A[x \leftarrow \dot{x}]}{\Gamma \vdash \exists x.A, \Delta}\ (\exists R)$	inst., quantifier, logical
$\dfrac{\Gamma_1, B, A, \Gamma_2 \vdash \Delta}{\Gamma_1, A, B, \Gamma_2 \vdash \Delta}\ (ExcL)$	$\dfrac{\Gamma \vdash \Delta_1, A, B, \Delta_2}{\Gamma \vdash \Delta_1, B, A, \Delta_2}\ (ExcR)$	structural
$\dfrac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\ (WeakL)$	$\dfrac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, A}\ (WeakR)$	structural
$\dfrac{A, A, \Gamma \vdash \Delta}{A, \Gamma \vdash \Delta}\ (CtrL)$	$\dfrac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta}\ (CtrR)$	structural

Figure 2.1: SC inference rules.

Propositional rules are possibly the simplest SC admits, they are responsible for managing conjuncts, disjuncts, negations and implications. Axiom and Cut rules are regarded as propositional.

1. Propositional axiom (Ax) states that the sequent having some syntactically equivalent antecedent and consequent is a tautology.
2. Cut rule (Cut) is a convenient mechanism to introduce case splitting in the current proof because it generates sub-goals "$\Gamma \vdash \Delta, A$" and "$\Gamma, A \vdash \Delta$". By the negation rule it is easy to see that A and $\neg A$ are antecedents of respective sub-goals, hence the application of Cut corresponds to introducing an assumptions and its converse.
3. Propositional rules ($\neg$, $\wedge$, $\vee$, $\Rightarrow$):
   (a) Negation rules ($\neg L$, $\neg R$) state that formulae can commute with respect to the turnstile, provided their polarity is adjusted.
   (b) Conjunction rules ($\wedge L$, $\wedge R$) state that the conjunction of antecedents is immaterial and that the conjunction of consequents must be evaluated in isolated sub-goals.
   (c) Disjunction rules ($\vee L$, $\vee R$) state that the disjunction of consequents is immaterial and that the disjunction of antecedents must be evaluated in isolated sub-goals.
   (d) Implication rules ($\Rightarrow L$, $\Rightarrow R$) are perhaps the less intuitive.
      i. Rule ($\Rightarrow L$) states that a useful implication must have a true premise. Alternatively, one can assume the conclusion to be true once the premise has been proved to hold.
      ii. Rule ($\Rightarrow R$) states an implication can be proved provided either its premise is false or when the premise is true, the conclusions is true.

Quantification rules are divided into *introductory* rules, those introducing skolem constants, and *instantiating* rules, those making use of them. Instantiating rules are not limited to skolem constants only, they can substitute variables for ground values.

1. Introductory rules ($\exists L$, $\forall R$) permit to introduce new constants that represents generic elements of the domain, consequently allowing to reason about them without being able to choose them. Assume true the statements $\exists x \in X : P(x)$; intuitively, if there exists an element satisfying $P(\cdot)$ the introduction of a skolem constant x' does not give the power to state that x' is arbitrary. Assume $\forall x \in X : P(x)$ has to be proved; if $P(x)$ actually holds on all elements of X, the introduction of a skolem constant x', representing a generic element, is sufficient to support the proof. In both cases, unless the user is able to prove it, to the new constant can not be associated more properties than the predicate it is bounded to.
2. Instantiating rules ($\exists R$, $\forall L$) make use of existing constants or ground values, they permit to choose the element subject to the predicate. Suppose $\exists x \in X : P(x)$ has to be proved and suppose some constant or ground value $\dot{x}$ satisfies $P(\cdot)$, then $\dot{x}$ can be instantiated in $P(\cdot)$. Assume true the statement $\forall x \in X : P(x)$; intuitively, if the predicate is satisfied by all elements of the domain we can instantiate $\dot{x}$, a convenient element of our choice.

Structural rules are employed to relax the proof permitting the user to rearrange the sequent by three operations: reordering antecedents and consequents, deleting formulae no longer useful, and carrying useful duplicate formulae.

1. Exchange rules ($ExcL$, $ExcR$) state that the order of antecedents and consequents is immaterial.
2. Weakening rules ($WeakL$, $WeakR$) state that formulae not required in a proof can be discarded.
3. Contraction rules ($CtrL$, $CtrR$) state that duplicates of existing formulae are immaterial.

SC has one axiom, (Ax) inference rule. It is recognizable by the lack of sub-goals depicted above the inference line.

PVS employs an extended version of SC: extensions are in terms of expressiveness and automation. Expressiveness is extended by means of syntactic sugar obtained by composition of pre-existing basic constructs, like the lambda abstraction, the if branching operation and evolved conditional statements. Automation is achieved by many decision procedures that can deduce whether some antecedent is false or some consequent is true, therefore they are able to complete the proof.

Some rules are applied automatically without user's intervention, at every step PVS tries to apply ($\neg L$) and ($\neg R$). Consequently, PVS does not allow negated formulae to appear in intermediate steps of the proof and moves them to the opposite side of the turnstile automatically; forcing PVS to keep negated formulae in place can be difficult. The last inference rule PVS applies automatically is (Ax), hence whenever some sequent trivially evaluates to a tautology the PA marks the goal as proved and proceeds to the next one (if any).

PVS makes the usage of inference rules transparent. Different inference rules are gathered together and tied to dedicated commands, hence manipulation of antecedents or consequents is intuitive. For instance, rules ($\wedge L$) and ($\vee R$) are applied by the *flatten* command. Their joint application is intuitive, $flatten$ is designed to get rid of conjunctions/disjunctions to access sub-formulae. In fact the net result of flattening an antecedent conjunction ($\bigwedge_i \phi_i$) or a consequent disjunction ($\bigvee_i \phi_i$) is removing operators $\bigwedge(\cdot)$ and $\bigvee(\cdot)$ leaving single ϕ_i in their place. Conversely, *split* command is responsible for separating sub-proof branches by virtue of ($\vee L$) and ($\wedge R$). Consequently, *split* and $flatten$ are also responsible for operating on ($\Rightarrow L$) and ($\Rightarrow R$), because the implication is expressed in terms of conjunction or disjunction depending on the side of the turnstile it occupies. The same is true for ($\exists L$) and ($\forall R$), they can be transparently employed through the *skolem* command. Their counterparts ($\forall L$) and ($\exists R$) are accessed by *instantiate* command.

SC develops proofs as trees, unfolded in a bottom-up fashion by a recursive procedure. The purpose of the procedure is generating the proof and eventually terminate; this happens when terminal nodes are reached. For each unproved goal the application of inference rules allows to obtain one or more sub-goals or the conclusion of the proof. The statement of the theorem plays the role of the root; suppose we want to prove De Morgan's law, then the sequent encoding the proof goal is:

$$\overline{\vdash \forall(P, Q : bool) : \neg(P \vee Q) \equiv (\neg P) \wedge (\neg Q)} \tag{2.2}$$

An unproved sequent is called a leaf. Leaves are proved when the PA is able to determine that (Ax) applies, namely when one of the following conditions happen:

1. some antecedent formula is false: $A_1 \wedge \neg A_1, \ldots, A_n \vdash B_1, B_2 \ldots, B_n$,
2. some consequent formula is true: $A_1, A_2, \ldots, A_n \vdash B_1 \vee \neg B_1, \ldots, B_{n-1}$,
3. the same formula is among antecedents and consequents: $C, A_1, A_2, \ldots, A_n \vdash C, B_1, B_2, \ldots, B_n$.

Observe that the former two cases can be rewritten by $(\neg L)$ and $(\neg R)$ and expressed in terms of the third, making (Ax) applicable. When a leaf is proved, it is said to be discharged. A branch is said to be discharged when all its leaves are discharged. A theorem is regarded as proved when all branches of its proof tree are discharged.

Let us now show an example proof with the purpose to clarify proof development and the application of inference rules. As mentioned earlier, the statement of the theorem is the root of the tree, the proof starts there. The user must apply some suitable inference rule which is annotated on the right of the inference line with the parameters it requires (if any). This simple proof does not require to point out the formula to which the inference rule is applied, in general this information is required. Inference deterministically generates sub-goals to which the recursive procedure must be applied. Supposing the user is able to prove the theorem, the proof attempt continues until all branches are discharged.

Theorem 1 (Demonstrative Proof)**.** $\vdash \forall(A, B, C : bool) : \big(A \Rightarrow B \wedge (A \vee C)\big) \Rightarrow (B \vee C)$

Proof. By direct proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{a \vdash \mathbf{a}, b, c}\,(Ax)}{\neg\mathbf{a}, a \vdash b, c}\,(\neg L) \quad \dfrac{}{\mathbf{b}, a \vdash b, c}\,(Ax)}{\mathbf{a} \Rightarrow \mathbf{b}, a \vdash b, c}\,(\Rightarrow L) \quad \dfrac{}{a \Rightarrow b, c \vdash b, c}\,(Ax)}{a \Rightarrow b, (a \vee c) \vdash b, c}\,(\vee L)}{\big(a \Rightarrow b \wedge (a \vee c)\big) \vdash b, c}\,(\wedge L)}{\big(a \Rightarrow b \wedge (a \vee c)\big) \vdash (b \vee c)}\,(\vee R)}{\vdash \big(a \Rightarrow b \wedge (a \vee c)\big) \Rightarrow (b \vee c)}\,(\Rightarrow R)}{\vdash \forall(A, B, C : bool) : \big(A \Rightarrow B \wedge (A \vee C)\big) \Rightarrow (B \vee C)}\,(\forall R, [A \leftarrow a, B \leftarrow b, C \leftarrow c)]$$

□

The depiction employed in the proof of theorem 1 is standard and used in literature. It is convenient to understand how inference rules operate, however in PVS, proofs do not visually resemble this style.

$$\begin{array}{cc} -1 & A_1 \\ -2 & A_2 \\ \vdots & \vdots \\ -n & A_n \\ & \vdash\!\!\!\!\!\!\!\!- \\ 1 & B_1 \\ 2 & B_2 \\ \vdots & \vdots \\ m & B_m \end{array}$$

Figure 2.2: Leaves as shown in PVS.

PVS restricts proof inspection to the current leaf only, the *postpone* command postpones the current goal and re-routes the user to the next unproved leaf. This design choice is reasonable because developing proof trees like depicted in the proof of theorem 1 is impractical in terms of space consumption and clarity. PVS arranges sequents like shown in fig. 2.2, also each formula of the current sequent is numbered: antecedents are negative indexed, while consequents are positive indexed. This depiction is extremely precise in clarifying which formula is modified but also space consuming. We presented it for showing how the proofs we provide in next chapters relate to their mechanical counterparts. We avoid to employ the aforementioned styles and prefer the one used in eq. (2.3) because it is very compact and clear. For specifying which formulae are modified, we employ the same indexing used in fig. 2.2 but omit all indexes. Hence when considering the following sequent

$$A_1, \ldots, A_n \vdash B_1, \ldots, B_m \tag{2.3}$$

we regard formula A_i as formula $-i$ and formula B_i as formula i.

Chapter 3

An Introduction to the Verification

In this chapter we develop the whole formalization of Peterson-Lock, the next chapter is focused on proving that Filter-Lock satisfies Mutual Exclusion (ME). The former represents a special case of the mutual exclusion protocol implemented by the latter, therefore, we formalize and prove them correct in this order. Peterson-Lock's formalization serves the purpose to expose and clarify several decisions and conventions that will be used in Filter-Lock's formalization; the latter represents the actual challenge, hence, in its development we wish to employ specific explanations we assume the reader is familiar with.

Before we proceed to any formalization we must discuss several aspects like: the motivations that led us to prefer the Prototype Verification System (PVS) over the Coq Proof Assistant (PA); the type of properties we plan to prove, how they are proved and how the PA can support us in this process; finally, how to synthesize the Input Output Automaton (IOA) provided the algorithm's pseudocode.

Each formalization is structured as follows: for each algorithm, we provide the IOA modelling it, we explain its peculiarities, we state required lemmata and develop the whole proof.

The chapter is structured as follows:

- In section 3.1, we consider PVS and Coq, two PAs that have been employed in literature for verifying concurrent software. We compare them and explain why we have chosen PVS.
- In section 3.2, we provide additional details about our decisions and the structure of our discussion. We explain the kind of properties we plan to employ, how they are proved and how Sequent Calculus (SC)' axiom can help to quicken the analysis.
- In section 3.3, we develop Peterson-Lock's formalization.
- In section 3.4, we discuss and explain six mechanical strategies used in our mechanical formalizations.

3.1 The Proof Assistant

In this section we discuss the motivations that lead us to choose a specific PA for carrying out our formalization. We present the two assistants we considered, we provide explanations to characteristic qualities that PAs are expected to enjoy and eventually explain our choice.

Becoming acquainted with a PA requires a considerable effort, the user wanting to employ some PA must devote several months of study to understand the tool and the details revolving around it; these comprehend and are not limited to: the proof system, the logic, possibly additional formalisms, the library shipped with the assistant and the interaction with the prover. Consequently, the choice of a PA is subjective, time consuming and hardly scientific.

In the preliminary phase of this work we evaluated two candidate PAs: Coq and PVS; they have been considered because of their previous employment in scientific literature and the presence of an active community keeping them maintained and up-to-date. Aforementioned PAs are currently heavily used in proving concurrent algorithms correct employing the IOA formalism: in [2, 20], Coq

has been employed for reasoning about distributed software correctness; in [9, 10, 14, 15], PVS has been employed for proving concurrent data structures and algorithms correct.

Coq and PVS have been designed with different purposes in mind, consequently they employ suitable proof systems and logics:

- The Coq PA has been designed to formalize mathematics and today is a general purpose tool. Its proof system is Natural Deduction (ND), its logic is Intuitionistic Logic (IL) and is augmented with the Calculus of Inductive Constructions (CIC) [7].
- The PVS PA has been designed for verifying hardware and software systems with high degrees of automation. Its proof system is SC, its logic is Higher Order Logic (HOL) [33].

In [34], seventeen industrial-strength PAs are employed to prove that $\sqrt{2} \notin \mathbb{Q}$. Each PA has been employed by its designers to prove the theorem: the result is a collection of formalizations highlighting interesting features of each system. We initially compared Coq and PVS according to the classification provided in [34]; table 3.1 shows the entries for the assistants of interest. For the sake of this discussion we say a feature is discriminating if and only if it is common to both candidates. In the table, discriminating features appear in top rows, non-discriminating ones are separated from the former and appear in bottom rows. In fact, both assistants are based on HOL, are extensible by means of strategies and tactics, employ typed logics extended by dependent types, satisfy Poincaré's principle and are featured with a mathematical library. Poincaré's principle states that predicates involving computations only do not require a proof [3]; consequently, it is satisfied when the PA has the ability to automatically prove the correctness of formulae involving calculations, like $1 + 1 = 2$.

Discriminating features are those that should allow to choose a candidate over the other; we found out that while being accurate, this classification is not sufficient for preferring a specific tool because none of the candidates is obviously superior.

Proof assistant:	PVS	Coq
Powerful automation	+	−
Constructive logic supported	−	+
Decidable Types	−	+
De Bruijn criterion	−	+
Based on HOL	+	+
Extensible by the user	+	+
Typed	+	+
Dependant Types	+	+
Poincaré principle	+	+
Large mathematical library	+	+

Table 3.1: Comparison of PA's features.

From the perspective of a novice user, discriminating features regard qualities that are either extremely clear or extremely nebulous. Almost all non-common features are difficult to assess because the user does not know a priori whether their lack implies relevant modifications to proofs and whether the formalization effort actually requires them. For instance, the decidability of types in PVS does not affect the power of the PA but forces the user to approach the formalization differently. The same is true for the lack of *powerful automation* in Coq, and the lack of *constructive logic* in PVS; it is difficult to tell whether these are actual impediments.

The feature that influenced our initial orientation the most has been the De Bruijn criterion: it is satisfied when the correctness of the proof system is guaranteed by a compact checking algorithm [7, p. 9][4]. The PA is a software responsible for aiding the construction of an error-free formalization, consequently the correctness of proofs depend on the correctness of the tool that in turn is not guaranteed to be bug free. Therefore, it is fair to ask whether such tools are reliable or not. The idea to connect correctness to a short checking procedure is desirable because it is known that the

shorter the algorithm the lower the probability it contains errors. PVS does not enjoy such property, meaning that the correctness of the proof depends on the consistency of non-compact procedures. Consequently, the probability a proof checked by PVS is correct is less or equal to the probability that the same proof is correct when checked by Coq. This is why we initially desired to take advantage of the Coq PA; however, we recall that the National Areonautic and Space Administration (NASA) agency, currently employs PVS for verifying its systems. We conclude that the probability that PVS accepts and incorrect proof is extremely low.

The second part of the initial phase has been devoted to test each PA, hence, we resorted to the evaluation of the code shipped with the aforementioned works ([2] for the Coq PA, [10] for PVS) and publicly available material ([7] for the Coq PA, [5] for PVS). We spent equal amounts of time trying to become acquainted with Coq and PVS for a total of two months of trial. During the evaluation we observed that from an abstract perspective Coq and PVS present many similarities: both are equipped with a specification language and a language to define extensions, both admit the employment of user extensions, and are operated by the user through a modern interface. User extensions are achieved through the employment of what are called "tactics" in Coq and "strategies" in PVS. These are basically programmable commands that the user can define to manipulate proof terms in order to repeat mechanical operations iteratively, collapsing repetitive and tedious steps. Tactics and strategies amount to sequences of basic proof commands provided by the PA and possibly other strategies provided by the user. The difference between basic proof commands, tactics and strategies lies in the ability of the end user to interfere with their application: rules are atomically applied, they are assumed to be correct and can not be modified by the end user; tactics and strategies need not be atomic, they are designed to be written, extended and adapted by the user. Consequently, tactics and strategies allow the user to extend the PA without compromising its correctness. We map the features that characterize Coq and PVS via four points:

- Coq's specification language is Gallina, an implementation of CIC and a powerful dialect of ML featured with complex polymorphism and dependent types.
- Coq's proof terms are manipulated by tactics and rules. Rules correspond to sequences of CIC's inference rules.
- Coq's tactic language is Ltac, a set of commands used to manipulate proof terms and prove theorems.
- Interaction with Coq happens via CoqIDE [27, p. 43].

Moreover, Coq is equipped with an additional language called "Vernacular" whose purpose is to enrich the environment by adding definitions, declaring inductive predicates and theorems; also Coq's parser is extensible, allowing the user to define and employ his own notation.

- PVS' specification language is "PVS language". It is a convenient language based on simply typed HOL supporting definition of generic (dependent) types using base types, functions, records and tuple types [33].
- PVS' proof terms are manipulated by strategies and rules. Rules correspond to sequences of SC inference rules.
- PVS' strategy language is Lisp augmented with PVS' rules and syntactic sugar.
- Interaction with PVS happens via Emacs, or Visual Studio Code - PVS extension [33, 27].

Relevant differences lie in the actual operation of the tool and the quality of the material the user is expected to consult during the training phase. On one hand, from our perspective Coq is subject to too much overloading: the Coq community treats the tool both as a programming environment and a PA; the duality of programs and proofs often makes the discussion opaque and difficult to follow. On the other hand, PVS is treated exclusively as a PA: this focuses the attention of the community on verification. Secondarily, Coq's community is bigger than PVS' and available resources teaching Interactive Theorem Proving (ITP) in Coq are much more abundant than PVS' however, the requirements that the Coq user is supposed to satisfy are much higher than those the PVS user is expected to satisfy. Last but not least, the Coq user must be very well acquainted with functional programming, conversely PVS does not require the same experience.

In our opinion SC is simpler to understand and employ with respect to ND. We have found CIC to be particularly cumbersome and according to us, the lack of analogue formalisms in PVS lowers the requirements placed upon the user making PVS preferable. The lack of automation in Coq makes ITP considerably more time consuming, and increases the effort the user must deal with. We found out that this feature is sometimes desirable, especially in the very beginning of the training phase because it forces the user to identify important steps of the proof and carry them out thoughtfully. However, this feature backfires when the user is acquainted with the PA and does not want to delve into detailed proofs. In fact, many low-level details that PVS can discharge automatically, in Coq must be manually managed by the user. An important detail regarding correctness proofs is that they are mathematically shallow, that is, they involved few and simple argumentations. These observations suggest that powerful automation can save time and mental energies, consequently making Coq a less desirable candidate. PVS is equipped with powerful decision procedures that allow the user to prove simple theorems very soon: it is the case of sum commutativity, the classic theorem used in ITP tutorials. This property of addition requires one lemma and the use of induction in case it is proved without equality decision procedures, conversely their employment terminates the proof immediately. PVS is designed to employ many decision procedures in autonomy, that allow the user to focus on complex proofs.

We discovered that the choice for a PA is very personal, because different combinations of features can result extremely fitting to some and inconvenient to others. Finally, all that being said, we decided to accomplish our verification in PVS considering the effort required for exploiting Coq far greater than the one required for exploiting PVS.

3.2 Invariant Properties

Peterson's algorithm is a simple solution to ME and its generalization to arbitrarily many processes is implemented by Filter-Lock. The former is significantly simpler than the latter, hence, we introduce the reader to the verification effort by developing the formalization of the Peterson-Lock. This task will help to introduce four important aspects that will be employed in Filter-Lock's formalization:

- the approach to encode the pseudocode into an IOA,
- the convention to name actions,
- the strategy to prove invariants,
- and the relation linking invariants to supporting invariants.

Proofs here provided are focused on most relevant aspects of the formalization and abstract low-level details away. The details we wish to remove regard the mechanical nature of the proof scripts we provide: we decide to employ a proof style that spaces between formal and mechanical, preferring a formal approach when possible. All mechanical proofs are formal and enjoy great rigour (the converse is not true), however mechanical proofs alone tend to be of little value to the novice user. Therefore, we employ a formal perspective and drop low-level details unless they are strictly required. We adopt this approach due to the overhead mechanical proofs involve; despite our proof scripts exploit PVS' automation whenever possible, the proof cumulatively amounts to roughly seven thousand PVS commands. Often the majority of them is required to unfold intuitive intermediate steps that from a non-mechanical perspective are obvious. This is why the proofs we provide follow the structure of the mechanical proof, but tend to give intuitive explanations mapping on sequences of commands. For each theorem, we provide its statement and when it is convenient we anticipate the proof by an explanation regarding why it works. Finally, the proof is provided.

The properties we wish to prove and employ depend on the implementation of the algorithm, and consequently on its structure. The proof we provide is based on invariants, that is, properties valid in all (reachable) states of the model. Invariants are inductive properties, consequently they are proved by induction: the user is responsible for proving the *basis* case and the *inductive step* case. In the basis, we prove the property holds in some generic initial state s_0. In the inductive step, we prove via a case analysis that no transition $s_0 \xrightarrow{\pi} s_1$ invalidates the property: given some generic pre-state s_0 where the property holds by inductive hypothesis, we prove it holds in the post-state s_1 as well. Other

proof methods are possible, in our verification we prove many invariants, both by induction or by direct proof. The latter case happens less often, that is, when the invariant can be proved employing one or more supporting invariants and no case analysis.

Invariant properties come with three advantages: few actions are actually involved in the case analysis because their majority preserves the property trivially, consequently the user can focus on relevant cases and dismiss the others; the case analysis involved by the inductive step is relatively effective in identifying Counter Examples (CEs), if they exist; invariant properties can be effectively composed together to design new invariants easily. We first define the reachability predicate:

Definition 3.2.1 (Reachability)**.** Let $\mathcal{M}$ be an LTS and $s_1 \in Q(\mathcal{M})$, then s_1 is reachable in $\mathcal{M}$ if and only if:

$$reach(\mathcal{M}, s_1) = s_1 \in I(\mathcal{M}) \vee (\exists s_0 \in Q(\mathcal{M}), \pi \in \Sigma(\mathcal{M}) : reach(s_0, \mathcal{M}) \wedge s_0 \xrightarrow{\pi} s_1)$$

Consequently, we define invariant properties:

Definition 3.2.2 (Invariant Property)**.** Let $\mathcal{M}$ be an LTS, let P be a predicate, then P is invariant over $\mathcal{M}$ if and only if:

$$invariant(\mathcal{M}, P) = \forall s_0 \in Q(\mathcal{M}) : reach(\mathcal{M}, s_0) \Rightarrow P(s_0)$$

Following [25, pp. 291-292], we differentiate invariants from strengthened invariants. The formers are invariant properties that can be proved by induction alone, this is possible when the inductive hypothesis is sufficient to prove the conclusion. The latter are invariant properties that are proved by induction or direct proof but also require supporting lemmata.

Non-strengthened invariants are proved by preservation of the property, that it, we must analyse all actions of the automaton and develop a sub-proof for each transition. Such analysis is the core of most inductive proofs over reachable states, hence, we wish to focus only on relevant cases and efficiently discharge the others. To do so, we take advantage of the (Ax) rule that applies when:

1. some antecedent formula is unsatisfiable,
2. some consequent formula is satisfied,
3. some formula appears both as antecedent and consequent.

These three cases map to as many situations:

1. the transition invalidates some premise; this happens mainly when the property is incompatible with the precondition of the analysed action,
2. the transition enforces some conclusion; this happens mainly when the property holds in the post-state by virtue of the transition step,
3. the transition leaves the state unchanged, preserving the property; this happens when the process does not take a step or when the transition is not responsible for modifying the variables involved by the invariant.

We take advantage of the first and last case to dismiss irrelevant case analyses: we observe that either the inductive hypothesis is incompatible with the precondition of the analysed action or the action does not modify variables involved by the property. The second case can not be exploited greedily, and we must carefully analyse actions not covered by the first and last one.

3.3 A Correctness Proof for Peterson-Lock

In this section we prove Peterson's algorithm correct: we first explain how the algorithm is encoded into an IOA, then we explain one denotational convention for naming actions, and finally we show

and comment the model. After this we state supporting invariants, we provide a proof for each one and finally provide the correctness proof.

We encode the algorithm as an IOA, to do so we first specify suitable program counter values and actions encoding the program. We associate one program counter value to each instruction, plus one reserved value representing the idle state. We equip the automaton with two external actions to interface with the algorithm and several internal actions: two actions for each branching instruction and one action for each non-branching instruction. One external action is used to encode the invocation of the algorithm, the other encodes the response; external actions bring the process from the idle state to the operational state and vice versa. Branching instructions are encoded by two actions: the first one directs the execution to the branch where the condition evaluates to true, the other encodes the specular behaviour. In this thesis, we comply to the convention of encoding conditional branches by actions whose name are suffixed with letters T or F. Actions suffixed by T are responsible for delaying access to the Critical Section (CS), conversely actions suffixed by F move the process closer to CS. We also adopt the convention to prefix all actions responsible for executing the Entry fragment with *lock* and all actions responsible for executing Exit fragment with the prefix *unlock*. The correspondence between single instructions of the pseudocode and (pairs of) actions of the model render this strategy convenient; it generates IOAs that strongly resemble the algorithm and clarify its execution flow.

We apply the aforementioned procedure to the Peterson-Lock and develop PL the IOA modelling it. When algorithm 1 is invoked it executes and returns only after the executing process has gained access to the CS and has abandoned it. Hence, we introduce in PL one action responsible for invoking the algorithm and bring it from the idle state to the first instruction and one responsible for bringing the process back to the idle state. Algorithm 1 exhibits two branching operations, one tests the level of the competitor process, the other tests whether the process is the designated victim; both operations are wait-free and singularly atomic. PL must preserve these properties, hence, discusses instructions

Algorithm 1 Peterson-Lock

```
1: global natural victim                                   ▷ Initially arbitrary.
2: global boolean level[2]                                 ▷ ∀p ∈ {0,1} ⇒ level[p] = false.
3: procedure PETERSON-LOCK(p ∈ [0,1])
4:     level[p] = true                                     ▷ Entry
5:     victim = p                                          ▷ Entry
6:     while level[1 − p] ∧ victim = p do {}               ▷ Entry
7:                                                        ▷ CS
8:     level[p] = false                                    ▷ Exit
```

necessarily must occur in some order but they can not happen together simultaneously. We decide to test the *level* variable first and *victim* variable only later.

With these assumptions, the following PC values $PCs = \{pc1, pc2, pc3, pc4, pc5, pc6, pcIdle\}$, and employing the aforementioned approach we define PL:

$$PL = \Big(Signature(PL), \big(Q(PL), I(PL), \Sigma(PL), \delta(PL)\big)\Big)$$

- $Signature(PL)$:
  - $input(PL) = \{lock\_invoke_p : p \in \mathbb{P}\}$
  - $internal(PL) = \{lock\_1_p, lock\_2_p, lock\_3T_p, lock\_3F_p, lock\_4T_p, lock\_4F_p, unlock\_1_p : p \in \mathbb{P}\}$
  - $output(PL) = \{lock\_response_p : p \in \mathbb{P}\}$
- $Q(PL) = pc \times level \times victim$
- $I(PL) = \big\{q \in Q(PL) : \forall p \in \mathbb{P} : \big(q.pc_p = pcIdle \wedge q.level[p] = false\big)\big\}$
- $\Sigma(PL) = \{lock\_invoke, lock\_1, lock\_2, lock\_3T, lock\_3F, lock\_4T, lock\_4F, lock\_5, lock\_response\}$

State variable are typed as follows: $pc : \mathbb{P} \to PCs$, $level : \mathbb{P} \to \{\top, \bot\}$, $victim : \mathbb{P}$.

$\delta(PL)$	Comment:
lock_invoke$_p$:	Invocation.
pre: $pc_p = pcIdle$	
eff: $pc_p = pc1$	
lock_1$_p$:	Set $level$.
pre: $pc_p = pc1$	
eff: $level[p] = true \wedge pc_p = pc2$	
lock_2$_p$:	Set $victim$.
pre: $pc_p = pc2$	
eff: $victim = p \wedge pc_p = pc3$	
lock_3F$_p$:	The other does not compete: enter.
pre: $pc_p = pc3 \wedge \neg level[1-p]$	
eff: $pc_p = pc5$	
lock_3T$_p$:	The other competes: delay.
pre: $pc_p = pc3 \wedge level[1-p]$	
eff: $pc_p = pc4$	
lock_4F$_p$:	The other is victim: enter.
pre: $pc_p = pc4 \wedge victim \neq p$	
eff: $pc_p = pc5$	
lock_4T$_p$:	I am victim: delay.
pre: $pc_p = pc4 \wedge victim = p$	
eff: $pc_p = pc3$	
unlock_1$_p$:	Performs unlock.
pre: $pc_p = pc5$	
eff: $pc_p = pc6 \wedge level[p] = false$	
lock_response$_p$:	Response.
pre: $pc_p = pc6$	
eff: $pc_p = pcIdle$	

Table 3.2: Transition function, $\delta(PL)$.

Peterson-Lock is so simple and succinct that in [25, pp. 278-283] a very similar formalization is provided; our formalization presents differences in the model, in the properties and in the proof.

1. Our model differs in the lack of two external actions, executed respectively to conclude Entry and to start Exit fragment,
2. our properties differ in the statement of one invariant: our version lacks a clause,
3. our proof differs in the the strategy employed to prove the algorithm satisfies mutual exclusion, specifically the case analysis for action $\pi = lock\_4F$.

In this thesis we adopt the convention to suffix all property indexes with letters P or F depending on the algorithm that satisfies them: for instance, lemma 1P is satisfied by Peterson-Lock and lemma 1F is satisfied by Filter-Lock. Peterson's algorithm can be proved correct by two supporting invariants. The first one specifies that all processes have $level$ variable set to $true$ or $false$ depending on the PC value:

$$\forall s_0 \in Q(PL) : reach(PL, s_0) \Rightarrow \forall p \in \mathbb{P} : s_0.pc_p \in \{pc2, pc3, pc4, pc5\} \Leftrightarrow s_0.level[p] = true$$

The second invariant specifies that when process p wins the competition leaving the other behind, p is not the victim:

$$\forall s_0 \in Q(PL) : reach(PL, s_0) \Rightarrow \forall p, q \in \mathbb{P} : s_0.pc_p = pc5 \wedge s_0.pc_q \in \{pc3, pc4\} \wedge p \neq q \Rightarrow s_0.victim \neq p$$

The version of this property stated in [25, pp. 287-283] also describes a secondary scenario where both p and q are in CS and p can not be the victim. When we proved the Peterson-Lock correct,

we independently discovered such property lacking the clause regarding p being in CS; we prefer this version because we find it more intuitive and its employment in the mechanical setting is slightly simpler. By means of these two properties, we can prove Peterson-Lock satisfies mutual exclusion:

$$\forall s_0 \in Q(PL) : reach(PL, s_0) \Rightarrow \forall p, q \in \mathbb{P} : \neg(s_0.pc_p = pc5 \wedge s_0.pc_q = pc5 \wedge p \neq q)$$

Before we proceed we introduce a convenient lemma used to prove those branches where process p is suspended.

Lemma 1P (Suspended Process)**.** In all reachable states, suspended processes are not subject to updates.

$$\forall s_0, s_1 \in Q(PL), \pi \in \Sigma(PL), p \in \mathbb{P} :$$
$$\big(reach(PL, s_0) \wedge s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \wedge proc(\pi) \neq p\big) \Rightarrow \big(s_0.pc_p = s_1.pc_p \wedge s_0.level[p] = s_1.level[p]\big)$$

Proof. By construction. □

We proceed with the minor formalization, stating and proving each invariant.

Invariant 1P (Interest in Competition)**.** In all reachable states, level variable is *true* after it has been set to *true* and before it is reset to $false$.

$$\forall s_0 \in Q(PL) : reach(PL, s_0) \Rightarrow \Big(\forall p \in \mathbb{P} : s_0.pc_p \in \{pc2, pc3, pc4, pc5\} \Leftrightarrow s_0.level[p] = true\Big)$$

The proof of invariant 1P could be split into two sub-branches, the ($\Leftarrow$) case and ($\Rightarrow$) case. Since equivalence ($A \Leftrightarrow B$) is satisfied when left (A) and right (B) formulae are equally satisfied, we do not split the proof and evaluate left and right formulae to tell whether the equivalence holds. In this proof, the case analysis is extended to all actions of the automaton but the actual proof is provided for one scenarios only; in case the same proof applies to two or more actions we omit the duplicated proof and point out to the analogous case.

Proof. By induction on reachable states of the automaton.
Basis. Let s_0 be an initial state of the automaton.

$$s_0 \in I(PL) \vdash \forall p \in \mathbb{P} : s_0.pc_p \in \{pc2, pc3, pc4, pc5\} \Leftrightarrow s_0.level[p] = true$$

Let p be a generic process in state s_0. Observe that in initial states all processes have their PC value set to *pcIdle* and *level* variable set to $false$. Consequently, satisfying the consequent. □
Inductive step. Let s_0 and s_1 be states of the automaton, let s_0 be reachable; we assume the invariant holds in state s_0 by inductive hypothesis.

$$\forall p \in \mathbb{P} : s_0.pc_p \in \{pc2, pc3, pc4, pc5\} \Leftrightarrow s_0.level[p] = true, reach(PL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash$$
$$\forall p \in \mathbb{P} : s_1.pc_p \in \{pc2, pc3, pc4, pc5\} \Leftrightarrow s_1.level[p] = true$$

Let p be a generic process in state s_0. There are two cases:
Case 1. Suppose that process p is the generic executor: $proc(\pi) = p$. We analyse all the transitions of the automaton.

- $\pi = lock\_invoke$, this action invokes the algorithm and does not satisfy the PC based conjunct neither in the pre-state nor in the post-state. Observe that when A is false the equivalence ($A \Leftrightarrow B$) reduces to $\neg B$, hence, we have that $s_0.level[p] \neq true$ and that $s_1.level[p] \neq true$. The action does not modify *level*, trivially preserving the property. □
- $\pi = lock\_1$, this action sets $level[p] = true$. In the post-state $pc_p = pc2$ and $level[p] = true$, consequently enforcing the conclusion. □
- $\pi = lock\_2$, this action does not modify *level* variable but satisfies the PC based conjunct in the pre-state and the post-state. Analogously to case $\pi = lock\_invoke$, it preserves the property. □

- $\pi = lock\_3T$, this case is analogous to case $\pi = lock\_2$. □
- $\pi = lock\_3F$, this case is analogous to case $\pi = lock\_2$. □
- $\pi = lock\_4T$, this case is analogous to case $\pi = lock\_2$. □
- $\pi = lock\_4F$, this case is analogous to case $\pi = lock\_2$. □

- $\pi = unlock\_1$, this action sets $level[p] = false$. In the post-state $pc_p = pc6$, consequently left and right predicates are invalidated: $s_1.level[p] \neq true$ and $s_1.pc_p \notin \{pc2, pc3, pc4, pc5\}$. This satisfies the consequent. □

- $lock\_reponse$, this case is analogous to case $\pi = lock\_invoke$. □

Case 2. Conversely, suppose the generic executor is not p, $proc(\pi) \neq p$, then by lemma 1P we know that the transition function can not modify p's pc and $level$ variables. Consequently, the invariant is preserved by all actions. □

The proof we just gave is extremely simple, especially when carried out with the aid of PVS. In fact, invariant 1P does not require strengthening because the inductive hypothesis is always sufficient for concluding the theorem holds. The high automation degree allows to discharge all subcases automatically. We now state and prove invariant 2P, it is strengthened by invariant 1P.

Invariant 2P (Winner is not Victim)**.** In all reachable states, when p is winner and q is loser of the competition, p is not the victim.

$$\forall s_0 \in Q(PL) : reach(PL, s_0) \Rightarrow$$
$$\Big(\forall p, q \in \mathbb{P} : s_0.pc_p = pc5 \wedge s_0.pc_q \in \{pc3, pc4\} \wedge p \neq q \Rightarrow s_0.victim \neq p\Big)$$

Proof. By induction on reachable states of the automaton.
Basis. Let s_0 be an initial state of the automaton.

$$s_0 \in I(PL) \vdash \forall p, q \in \mathbb{P} : s_0.pc_p = pc5 \wedge s_0.pc_q \in \{pc3, pc4\} \wedge p \neq q \Rightarrow s_0.victim \neq p$$

Let p and q be generic processes in state s_0. Observe that in initial states all processes have their PC set to $pcIdle$, trivially satisfying the consequent. □
Inductive step. Let s_0 and s_1 be states of the automaton, let s_0 be reachable; we assume the invariant holds in state s_0 by inductive hypothesis.

$$\forall p, q \in \mathbb{P} : s_0.pc_p = pc5 \wedge s_0.pc_q \in \{pc3, pc4\} \wedge p \neq q \Rightarrow s_0.victim \neq p, reach(PL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash$$
$$\forall p, q \in \mathbb{P} : s_1.pc_p = pc5 \wedge s_1.pc_q \in \{pc3, pc4\} \wedge p \neq q \Rightarrow s_1.victim \neq p$$

Let p and q be generic processes in state s_0. There are three cases:
Case 1. Suppose that p or q take a step, $p = proc(\pi) \vee q = proc(\pi)$. There are two cases:
Case 1.1. Suppose that process p takes a step, $p = proc(\pi)$. Observe that the conjunct regarding p's PC is invalidated by all actions having in their effect the rewrite of pc_p to values different from $pc5$. Those actions trivially satisfy the consequent, therefore, they do not require additional analysis. Conversely, actions $\pi \in \{lock\_3F, lock\_4F\}$ must be analysed:

- $\pi = lock\_3F$, this action tests $1 - p$'s $level$ variable. Observe that the precondition is satisfied when $level[1 - p] = false$ and by invariant 1P, applied twice, we know that $level[p] = true$ and $level[1 - p] = true$. Independently of the value associated to p and q, the action can not be executed. This contradicts the hypothesis that p takes a step. □

- $\pi = lock\_4F$, this action tests $victim$ variable. Observe that the precondition is satisfied when $victim \neq p$, enforcing the property. □

Case 1.2. Conversely, suppose that q takes a step, $q = proc(\pi)$. Observe that the conjunct regarding q's PC value is invalidated by all actions that do not set PC to $pc3$ or $pc4$. Consequently, actions $\pi \in \{lock\_2, lock\_3T, lock\_4T\}$ must be analysed:

- $\pi = lock\_2$, this action sets *victim* variable. Observe it sets $victim = q$, consequently satisfying the conclusion. □
- $\pi = lock\_3T$, this action tests $level[1-q]$. Observe that the inductive hypothesis is satisfied and preserved by the transition, because variable *victim* is not overwritten. □
- $\pi = lock\_4T$, this action tests *victim* variable. Observe the precondition is satisfied when $victim = q$, consequently satisfying the conclusion. □

Case 2. Conversely, suppose the generic executor is neither p nor q, $p \neq proc(\pi) \wedge q \neq proc(\pi)$, then by lemma 1P applied twice, both processes are suspended and the transition function can not modify their variables. Consequently, the invariant is preserved by all actions. □

Eventually, we state and prove the safety property, it is strengthened by invariants 1P and 2P.

Invariant 3P (Mutual Exclusion)**.** In all reachable states, distinct processes can not occupy CS contemporarily.

$$\forall s_0 \in Q(PL) : reach(PL, s_0) \Rightarrow \forall p, q \in \mathbb{P} : \neg(s_0.pc_p = pc5 \wedge s_0.pc_q = pc5 \wedge p \neq q)$$

Proof. By induction on reachable states of the automaton.
Basis. Let s_0 be an initial state of the automaton.

$$s_0 \in I(PL) \vdash \forall p, q \in \mathbb{P} : \neg(s_0.pc_p = pc5 \wedge s_0.pc_q = pc5 \wedge p \neq q)$$

Let p and q be generic processes in state s_0. Observe that in initial states, no process has its PC value set to $pc5$; this trivially satisfies the consequent. □
Inductive step. Let s_0 and s_1 be states of the automaton, let s_0 be reachable; we assume the invariant holds in state s_0 by inductive hypothesis.

$$\forall p, q \in \mathbb{P} : \neg(s_0.pc_p = pc5 \wedge s_0.pc_q = pc5 \wedge p \neq q), reach(PL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash$$
$$\forall p, q \in \mathbb{P} : \neg(s_1.pc_p = pc5 \wedge s_1.pc_q = pc5 \wedge p \neq q)$$

Let p and q be generic processes in state s_0. There are two cases:
Case 1. Suppose p or p is the generic executor, $p = proc(\pi) \vee q = proc(\pi)$. There are two cases:
Case 1.1. Suppose that p is the generic executor, $p = proc(\pi)$, and takes a step. Consequently, by virtue of lemma 1P process q is suspended and all its variables are untouched. Observe that all actions but $\pi \in \{lock\_3F, lock\_4F\}$ trivially satisfy the consequent due to a PC based conflict in the post-state. We direct our attention to remaining actions:

- $\pi = lock\_3F$, this action is executed when $level[1-p] = false$. By invariant 1P, applied twice, we know that $level[p] = true$ and $level[1-p] = true$. Consequently, the action can not be executed, this is a contradiction. □
- $\pi = lock\_4F$, this action is executed when $victim \neq p$. Observe that since q is suspended $s_0.pc_q = s_1.pc_q$ and by hypothesis $s_1.pc_q = pc5$. Observe that $s_0.pc_p = pc4$. By invariant 2P we know that $s_0.victim \neq q$. Since $(s_0.victim, p, q) \in \{0, 1\}$, independently of the value associated to p and q we reach a contradiction. □

Case 1.2. This case is analogous to the case where p is the generic executor but here $q = proc(\pi)$. □
Case 2. Conversely, suppose the generic executor is neither p nor q, that is $p \neq proc(\pi)$ and $q \neq proc(\pi)$. Then by lemma 1P, applied twice, the transition function can not modify their variables and the invariant is preserved by all actions. □

As stated in the beginning of this section Peterson-Lock satisfies mutual exclusion. This formalization helped to introduce many useful elements upon which Filter-Lock's formalization is built. Filter-Lock represents a formalization challenge, the proof is considerably more sophisticated than the one we have just shown. In fact, it takes tens of lemmata and intermediate proofs to support the safety property.

3.4 Six Custom Strategies

In this section we discuss six strategies that are employed in the mechanical version of our formalizations: at this point the reader that wishes to compare the mechanical and formal setting for the Peterson-Lock is almost ready to do so. Mechanical proofs take advantage of several handy strategies whose purpose is to make the proof as accessible and intuitive as possible, however the user that is not familiar with them could take some time to understand their purpose. We postponed their discussion with respect to the formalization of the Peterson-Lock because in our perspective the user must first understand the proof from a formal perspective and inspect the mechanical proof only later. We anticipate their discussion with respect to the formalization of the Filter-Lock to provide to the user a complete vision of the minor formalization.

In fact, custom strategies are the most important feature the user can exploit to break down proof complexity. PVS strategies are programmed in Common Lisp [33] and are stored into a special file named "pvs-strategies" or in strategy packages [1]. Strategies serve many purposes and are in fact an adaptable tool to obtain heterogeneous results:

- they shorten proofs by carrying out repetitive work,
- they improve commands accessibility by means of aliases,
- they focus user's attention on relevant cases,
- they test hypothesis,
- they can prove whole theorems autonomously.

We report and explain six strategies we designed or adapted that are a product of the formalization. The third one is suggested in [10], the others are original and devised by us:

1/2. (*in-*)*fact*, states a fact and can be used to test hypothesis,
3. *inv-setup*, sets up induction on reachable states of the automaton,
4. *for-alpha*, develops case analyses for all actions and possibly proves the property is preserved,
5. *notice*, satisfies the premise of some invariant to obtain its conclusion,
6. *post-it*, instantiates invariants in the post-state.

The first and second strategies we discuss are *fact* and *in-fact*. They descend from an adaptable strategy: in this context the *think* command is simply a placeholder for some decision procedure; the granularity is adjusted rewriting it: we employed the *fact* strategy that exploits *assert* instead of *think* and the *in-fact* strategy that employs *grind*.

```
1 (defstep fact (arg &rest args)
2   (apply
3     (then
4       (branch (case arg)
5         ((postpone)
6          (think))
7       )
8       (if (null args)
9         (postpone)
10        (hide (args))
11      )
12    )
13  )
14  "fact ``...'' (fnum*)." ; Help
15  "Fact stated." ; Success msg.
16 )
```

Figure 3.1: *fact* strategy.

The two are able to introduce new formulae with different degrees of power. The idea supporting *fact* is to introduce consistent intermediate formulae that can be proved to hold in the light of the current context. The strategy is simple and builds on pre-existing commands: *apply* causes PVS to collapse the operation in a single atomic step; *then* is used to execute a list of commands; *branch* is a strategy that executes its parameter on the sequents obtained by its application, it applies the n-th command of the list to the n-th child sequent; *case* is responsible for splitting the proof into two branches, one where its parameter is supposed to be true, the other where it is supposed to be false; eventually, the *hide* command hides those formulae that the user does not want to employ any longer. Consequently, *fact* is responsible for splitting the proof into two branches: one where we suppose *arg* is true, the other where we suppose *arg* is false. By hypothesis, if *arg* can be proved true it is not false, hence the second proof branch is discharged automatically. In case PVS is unable to deduce that *arg* is true, the user can immediately notice because the proof is split in two unproved branches. Conversely,

the fact is introduced. This strategy allows to clarify the proof context because the user can introduce new formulae that represent the status of the reasoning and prove several properties that PVS is unable to observe on its own. The second point is not trivial, the possibility to suppose some formula true that afterwards is discovered to be true, often represents a leap toward the conclusion. On one hand, *fact* is for introducing formulae that are tautologies in the light of the current context but do not require additional operations. On the other hand, *in-fact* strategy is more effective at stating tautologies based on transitions, by virtue of the employment of the *grind* command that decomposes the transition and considers how it is defined. In fig. 3.2, we show a few steps of invariant 3P's proof, we employ *in-fact* to state several facts and hide the premises that justified them: the proof becomes compact and only relevant facts are retained. *lemma* command is used to employ some lemma in the current proof context.

$s_0.pc_p = pc3, s_1.pc_q = pc3 \vee s_1.pc_q = pc4 \vdash p = q, \ldots$	lemma $invariant\_1P$
$s_0.pc_p = pc3, s_1.pc_q = pc3 \vee s_1.pc_q = pc4, inv\_1P \vdash p = q, \ldots$	lemma $invariant\_1P$
$s_0.pc_p = pc3, s_1.pc_q = pc3 \vee s_1.pc_q = pc4, inv\_1P, inv\_1P \vdash p = q, \ldots$	expand inv_1P
$s_0.pc_p = pc3, s_1.pc_q = pc3 \vee s_1.pc_q = pc4, \ldots \vdash p = q, \ldots$	inst -4 p
$s_0.pc_p = pc3, s_1.pc_q = pc3 \vee s_1.pc_q = pc4, \ldots \vdash p = q, \ldots$	inst -5 q
$s_0.pc_p = pc3, s_1.pc_q = pc3 \vee s_1.pc_q = pc4, \ldots \vdash p = q, \ldots$	in-fact "$s_0.level(p)$"
$s_0.pc_p = pc3, s_1.pc_q = pc3 \vee s_1.pc_q = pc4, \ldots, s_0.level(p) \vdash p = q, \ldots$	in-fact "$s_0.level(q)$"(-1 -2 -3)
$s_0.level(p), s_0.level(q) \vdash p = q, \ldots$	in-fact "$s_0.victim = p$"
$s_0.level(p), s_0.level(q), s_0.victim = p \vdash p = q, \ldots$	in-fact "$proc(\pi) \in \{0, 1\}$"
$s_0.level(p), s_0.level(q), s_0.victim = p, proc(\pi) \in \{0, 1\} \vdash p = q, \ldots$	*grind*

Figure 3.2: *in-fact* strategy in action.

The third strategy we discuss is *inv-setup*. It is adapted from the one provided in [10], it gathers several steps that set up induction for proving invariants, it is the first or second step of invariant proofs. The strategy first instructs PVS to expand the definitions provided in "invariant" theory, allowing to remove explicit expansions of each definition. It sets up the basis and inductive step case. The former is left untouched, the latter is organized as the inductive step employed in previous proofs: s_0 is defined reachable while the inductive hypothesis, the transition step and the inductive step are labelled.

```
(defstep inv-setup ()
  (then
    (auto-rewrite-theory-with-importings "invariant" :importchain? nil)
    (assert)
    (rule-induct reachable? + reachable?_induction)
    (with-labels (skolem 1 "s1") (("ind-step")))
    (flatten)
    (branch (split)
      ((postpone)     ; Leave basis untouched.
       (with-labels   ; Set up ``reach'', ``ind-hyp'' and ``tran'' predicates.
         (then(skolem -1 (s0 alpha))(flatten))
        (("reach" "ind-hyp" "tran" "ind-step"))
       )
      )
    )
  )
  "Sets up the induction over reachability." ; Help
  "Induction is set up."                     ; Success msg.
)
```

Figure 3.3: *inv-setup* strategy.

The fourth strategy we discuss serves the purpose to quicken invariant proofs; we have clearly shown that invariants are often preserved by many actions of the model and only a small fraction needs to be analysed to explicitly show the property is preserved. We provide a strategy for developing automatically all transitions and possibly to discharge those branches where the invariant is preserved

trivially. *for-alpha*'s purpose is to mechanize the time consuming task of developing the invariant

```
1 (defstep for-alpha (&key lazy?)
2   (then
3     (with-labels(typepred alpha)((alpha)))
4     (install-rewrites :defs T :theories (``actions'' ``Peterson''))
5     (assert alpha)
6     (branch (split alpha)
7       ((then
8         (eta-alpha) ; Applies eta axiom to alpha: obtains alpha = ...
9         (replace alpha tran :dir rl) ; Rewrites alpha in tran.
10        (assert tran) ; Develops the definition w.r.t. alpha.
11        (with-labels(flatten tran)(("pre" "eff")))
12        (hide alpha)
13        (if (null lazy?) (think) (postpone))
14      ))
15    )
16  )
17  "for-alpha :lazy? (T|nil)"          ; Help
18  "Alpha values are now considered." ; Success msg.
19 )
```

Figure 3.4: *for-alpha* strategy.

action-based case analysis. In our proofs we assume that label "alpha" is reserved to actions and label "tran" is reserved to transitions; hence, in line 3 of fig. 3.4 we introduce the disjunction of actions and label it "alpha". In line 4, PVS installs all definitions regarding actions: it is a convenient way to automatically expand the definition from top to bottom, also it allows to do so using a single *assert*. In line 4, we employ the same principle for expanding the definition of transition. Eventually, we instruct PVS to split the disjunction and for each child sub-proof to apply the same list of commands: rewrite the actual action in the definition of the transition step and expand the transition as much as possible. This generates a context inside which the action is hidden, the precondition of the action is labelled "pre" and its effect is labelled "eff". The lazy version stops here, but the user can invoke it for trying to discharge the sub-proof automatically adjusting the *think* placeholder. Those invariants that discuss properties of a single processes can be proved by two calls to $for\text{-}alpha : lazy?\ nil$. It is sufficient to split the proof in two subcases, one where process p executes, the other where p is suspended: the strategy must be run in each one. As a demonstration, we provide the proof for invariant 1P (page 26). The employment of rewrite rules allows to focus on relevant steps of the

```
1 (then
2  (install-rewrites :defs T :theories (``Lemmata'' ``Peterson''))
3  (branch (inv-setup)
4   ((then (assert) (skeep) (grind)) ;-Initial state
5    (then ;-Inductive step
6     (assert (ind-step ind-hyp)) (skeep) (inst?)
7     (branch (case "proc(alpha)=p")
8      ((for-alpha :lazy? nil)
9       (for-alpha :lazy? nil))
10    )
11  )))
12 )
```

Figure 3.5: Proof object of invariant 1P.

proof, that is, introduce skolem constants and observe the invariant is preserved. In case *for-alpha* is unable to complete the proof alone, the user is rerouted to any unproved branch and PVS asks for a command.

The fifth strategy we wish to discuss is *notice*. This strategy is useful for getting rid of implications ($A \Rightarrow B$) by taking advantage of the inference rules ($\Rightarrow L$). Antecedent implications are proved splitting the proof into two sub-branches: in the first one the conclusion (B) of the implication is assumed to hold, in the second one the premise of the implication (A) must be proved to hold.

Supposing to have some formula that proves that the premise is true, *notice* replaces the implication with the conclusion.

```
(defstep notice (index)
  (apply (branch (split index) ((postpone)(think)) ) )
  "notice (fnum)"       ; Help
  "Premise discharged." ; Success msg.
)
```

Figure 3.6: *notice* strategy.

Since many invariants are implications and many conclusions are premises for other invariants, having a simple and compact way to extract conclusions is extremely important.

The last strategy we want to discuss is *post-it*, its purpose is to instantiate invariants in post-states. The only requirement the user must satisfy is to provide a proof that the post-state is reachable; to do so it suffices to develop the definition of reachability from the post-state backwards. The *post-it* strategy does exactly this, it sets up the invariant in the post-state and splits the proof into two branches: in the first one we get the property we desire, in the second one the reachability proof is carried out. The intuition regarding the proof is that the post-state is reachable via the transition function starting from the pre-state; consequently the strategy recovers the reachability and transition definitions and plugs them in.

```
(defstep post-it (name)
 (apply(then(with-labels (lemma name) (("temp-lemma")))
            (expand invariant temp-lemma)
            (inst temp-lemma s1)
            (branch (with-labels (split index) (("conseq" "target")) )
             ((postpone) ; The branch where we want to work.
              (then(expand reachable? conseq)
                   (with-labels (flatten conseq) (("init-state" "ind-step")))
                   (inst ind-step s0 alpha)
                   (reveal reach) (reveal tran) (assert))
             ))))
 "post-it ``...''"    ; Help
 "Post-state reached."; Success msg.
)
```

Figure 3.7: *post-it* strategy.

The strategies we have provided are simple and intuitive, their employment collapses many repetitive steps together shortening proofs, giving them structure and providing useful tools for easing proof attempts. In the mechanical ME proof for Peterson-Lock and Filter-Lock we often takes advantage of aforementioned strategies: they aid us on focusing on relevant proof steps, compact the proof and abstract low-level details away.

Chapter 4

The Verification

This chapter is focused on proving that Filter-Lock satisfies Mutual Exclusion (ME). We assume the reader is acquainted with the methodology we exposed in the previous chapter, that is, the conventions used to: translate the pseudocode into an Input Output Automaton (IOA), label actions of the automaton, name, state and prove invariants.

The chapter is structured as follows:

- In section 4.1, we introduce the reader to Filter-Lock's formalization; here we obtain and discuss the model encoding the algorithm and provide an intuitive ME proof. Before we actually carry it out, we provide a depiction of the results dependency graph: it serves the purpose to highlight the role of each invariant and show how assertions support each other.
- In section 4.2, we state and prove all properties required to claim that Filter-Lock satisfies mutual exclusion.

4.1 A Correctness Proof for Filter-Lock: Preliminaries

In this section we introduce the reader to the formalization of the Filter-Lock: we first discuss how to obtain the IOA model encoding it, then we show and discuss the model in question, highlighting the role of ghost variables. We compare our formalization to the one found in literature in [25]: we explain where ours differs and why. Eventually, we sketch a strategy for proving mutual exclusion: the actual formal proof is provided in section 4.2.

Algorithm 1 Filter-Lock

```
1:  global natural victim[|ℙ|]                                   ▷ Initially arbitrary.
2:  global natural level[|ℙ|]                                    ▷ ∀j ∈ ℕ : (0 ≤ p ≤ |ℙ| − 1) ⇒ level[p] = 0.
3:  procedure FILTER-LOCK(p ∈ ℕ : p < |ℙ|)
4:      i = 1                                                    ▷ Entry
5:      while i < |ℙ| do                                         ▷ Entry
6:          level[p] = i                                         ▷ Entry
7:          victim[i] = p                                        ▷ Entry
8:          while (∃q ≠ p : level[q] ≥ i) ∧ victim[i] = p do {}  ▷ Entry
9:          i = i + 1                                            ▷ Entry
10:                                                              ▷ CS
11:     level[p] = 0                                             ▷ Exit
```

The IOA encoding Filter-Lock is analogous to the one encoding Peterson-Lock but with slight sophistications. When structuring the signature, the labelling alphabet and the transition function, we employ the same approach and conventions we employed to structure the IOA encoding Peterson-Lock; to define the states and the behaviour of the automaton we need to do additional work and solve the ambiguity regarding the evaluation of the innermost while-loop's boolean guard. We enforce the following evaluation order: each process p scans all competitors, one by one in arbitrary order

but without duplicate checks. In case some competitor q is found to have *level* greater or equal to i, *victim* variable is tested; in case p is marked as victim it restarts the scanning process, otherwise, it enters level $i+1$. Conversely, if all competitors are found to have *level* variable less than i, no further check is done and the process enters level $i+1$. The scanning process, carried out in arbitrary order without duplicates, is achieved by taking advantage of a finite set of processes S: it is initially empty and gathers the IDs of scanned competitors. When some yet to be scanned competitor, qq, is chosen, it is evaluated and added to the set; when S is full, all competitors have been scanned and the process can enter the next competition level. Every time the scanning is (re)initiated set S is cleared. In [22], an analogous approach is used to encode the scanning process and to prove the Bakery algorithm correct. The evaluation of the boolean guard involves four checks:

1. the identification of some yet to be scanned competitor qq,
2. the comparison of qq's *level* against scanner's i variable,
3. the testing for the presence of more processes to scan,
4. and the testing of *victim* variable.

We map these checks on as many pairs of actions indexed from 6 to 9 that are responsible for encoding the evaluation of the boolean guard:

1. $lock\_6T$ and $lock\_6F$ check the presence of the next competitor and possibly choose it,
2. $lock\_7T$ and $lock\_7F$ check the level of the k-th competitor out of $|\mathbb{P}|$,
3. $lock\_8T$ and $lock\_8F$ check whether all processes have been scanned,
4. $lock\_9T$ and $lock\_9F$ check *victim* variable.

Remaining actions are responsible for setting local and shared variables and iterating the outer while-loop. To render the formalization feasible, from a mechanical perspective, we add two state variables: a secondary finite set C equal to S's complement and a tertiary finite set W_j whose purpose is to collect the IDs of all winners of the competition on level j. S, C and W_j are defined as finite sets of natural numbers below $|\mathbb{P}|$. These are "ghost variables", they are not involved by the internal mechanics of the algorithm, but renderer the proof possible. By $\top$ we denote the full set, namely the set containing all items of some type. With these assumptions, employing the following PC values $PCs = \{pc1, pc2, pc3, pc4, pc5, pc6, pc7, pc7, pc9, pc8, pc9, pc10, pc11, pc12, pcIdle\}$ we declare:

$$FL = \Big(Signature(FL), \big(Q(FL), I(FL), \Sigma(FL), \delta(FL)\big)\Big)$$

- $Signature(FL)$:
  - $input(FL) = \{lock\_invoke_p : p \in \mathbb{P}\}$
  - $internal(FL) = \left\{ \begin{array}{l} lock\_1_p, lock\_2T_p, lock\_2F_p, lock\_3_p, lock\_4_p, lock\_5_p, \\ lock\_6T_p, lock\_6F_p, lock\_7T_p, lock\_7F_p, \\ lock\_8T_p, lock\_8F_p, lock\_9T_p, lock\_9F_p, \\ lock\_10_p, unlock\_1_p \end{array} : p \in \mathbb{P} \right\}$
  - $output(FL) = \{lock\_response : p \in \mathbb{P}\}$
- $Q(FL) = pc \times level \times i \times C \times S \times W \times qq \times victim$
- $I(FL) = \left\{ \begin{array}{c} q \in Q(FL) : \\ \forall p \in \mathbb{P} : \big(pc_p = pcIdle \wedge level_p = 0 \wedge i_p = 1 \wedge C_p = \top \wedge S_p = \emptyset\big) \\ \wedge \\ \forall j \in \mathbb{N} : \big((j = 0 \Rightarrow W_j = \top) \wedge (j \neq 0 \Rightarrow W_j = \emptyset)\big) \end{array} \right\}$

State variable are typed as follows: $pc : \mathbb{P} \to PCs$, $(level, i : \mathbb{P} \to \mathbb{N})$, $(C, S : \mathbb{P} \to 2^{\mathbb{P}})$, $(W : \mathbb{N} \to 2^{\mathbb{P}})$, $(qq : \mathbb{P} \to \mathbb{P})$, $(victim : \mathbb{N} \to \mathbb{P})$.

$\delta(FL)$	Comment:
lock_invoke$_p$: **pre:** $pc_p = pcIdle$ **eff:** $pc_p = pc1$	Invocation.
lock_1$_p$: **pre:** $pc_p = pc1$ **eff:** $pc_p = pc2 \wedge i_p = 1$	Sets i.
lock_2T$_p$: **pre:** $pc_p = pc2 \wedge i_p < \|\mathbb{P}\|$ **eff:** $pc_p = pc3$	Test i.
lock_2F$_p$: **pre:** $pc_p = pc2 \wedge i_p = \|\mathbb{P}\|$ **eff:** $pc_p = pc11$	Test i and enters CS.
lock_3$_p$: **pre:** $pc_p = pc3$ **eff:** $pc_p = pc4 \wedge level_p = i_p$	Set level.
lock_4$_p$: **pre:** $pc_p = pc4$ **eff:** $pc_p = pc5 \wedge victim[i_p] = p \wedge S = \{p\} \wedge C = S^c$	Set $victim[i]$.
lock_5$_p$: **pre:** $pc_p = pc5$ **eff:** $pc_p = pc6 \wedge S = \{p\} \wedge C = S^c$	Set up C_p and S_p.
lock_6T$_p$: **pre:** $pc_p = pc6 \wedge qq_p \in C_p \wedge qq_p \notin S_p$ **eff:** $pc_p = pc7$	Choose new qq.
lock_6F$_p$: **pre:** $pc_p = pc6 \wedge C_p = \emptyset$ **eff:** $pc_p = pc10 \wedge W_{i_p} = W_{i_p} \cup \{p\}$	Enter next level.
lock_7T$_p$: **pre:** $pc_p = pc7 \wedge level[qq_p] < i_p$ **eff:** $pc_p = pc8 \wedge S_p = S_p \cup \{qq_p\} \wedge C_p = C_p \setminus \{qq_p\}$	Remove qq from candidate set.
lock_7F$_p$: **pre:** $pc_p = pc7 \wedge level[qq_p] \geq i_p$ **eff:** $pc_p = pc9 \wedge S_p = \emptyset \wedge C_p = \top$	Go to *victim* check.
lock_8T$_p$: **pre:** $pc_p = pc8 \wedge \|S_p\| < \|\mathbb{P}\|$ **eff:** $pc_p = pc6$	Go to scanning.
lock_8F$_p$: **pre:** $pc_p = pc8 \wedge \|S_p\| = \|\mathbb{P}\|$ **eff:** $pc_p = pc10 \wedge S_p = \emptyset \wedge C_p = \top \wedge W_{i_p} = W_{i_p} \cup \{p\}$	Enter next level.
lock_9T$_p$: **pre:** $pc_p = pc9 \wedge victim[i_p] = p$ **eff:** $pc_p = pc5$	I am victim: to scanning
lock_9F$_p$: **pre:** $pc_p = pc9 \wedge victim[i_p] \neq p$ **eff:** $pc_p = pc10 \wedge W_{i_p} = W_{i_p} \cup \{p\}$	I am not victim: enter.
lock_10$_p$: **pre:** $pc_p = pc10$ **eff:** $pc_p = pc2 \wedge i_p = i_p + 1$	Increments i.

unlock_1$_p$:			Perform unlock.
pre:	$pc_p = pc11$		
eff:	$pc_p = pc12$	$\wedge$	
	$level_p = 0$	$\wedge$	
	$i_p = 1$	$\wedge$	
	$\forall j \in \mathbb{N}:$ $\Big(0 < j \leq i_p \Rightarrow (W_j = W_j \setminus \{p\})$	$\wedge$	
	$\neg(0 < j \leq i_p) \Rightarrow (W_j = W_j)\Big)$		
lock_response$_p$:			Response.
pre:	$pc_p = pc12$		
eff:	$pc_p = pcIdle$		

Table 4.1: Transition function, $\delta(FL)$.

The model we just described is subject to several design decisions that are not self apparent and require an explanation: action *lock_1* redundantly sets variable $i_p = 1$, this does not invalidate the formalization and its presence does not represent an impediment. We keep it to highlight an important point: executions that start in initial states having variable $i_p = 1$ and that reset it through action *unlock_1* are very convenient because they allow writing simple and elegant properties. Having $i_p \neq 1$ in initial states would force us to rewrite the properties involving this variable: resulting invariants would present disjunctions of PC based equalities in their premises. This situation is extremely inconvenient because when proving such properties the proof tree becomes very deep and branched out, consequently Prototype Verification System (PVS) requires tens of minutes to rerun the case analysis, resulting in a waste of time and electricity.

Actions *lock_4* and *lock_5* have very similar effects, this is due to the purpose they serve: action *lock_4* is used to set the executing process as victim of level j but it must set $S = \{p\}$ and $C = S^c$ to make invariant 20F (page 62) provable; action *lock_5* is responsible for resetting S and C before a new scan of all competitors is initiated.

Set C is required to prove that during the last iteration of the scanning process qq is exactly the last competitor, this make invariant 1F (page 41) provable. Initial states are characterized by having variables $C = \top$ and $S = \emptyset$, this makes invariant 20F (page 62) provable.

Set W_j is required to record all processes that win the competition on level j, this ghost variable is useful for simplifying the proof of invariants 4F to 6F and 13F (pages 48, 50, 51, and 57).

We now consider a similar IOA modelling Filter-Lock that is available in literature: we explain how it differs from ours and what elements we have adopted. In [21, 31], correctness proofs are provided but they failed to convince us: these resources are subject to the lack of formal details and the employment of several lemmata left to the imagination of the reader that are definitely an impediment. Effectively, the core of the proof could be missed: during our first attempt we planned to show that the competing victim could not perform the entry step in Critical Section (CS) because the actions stepping into it can not be executed. Despite this fact is correct, we could not prove it.

In [25], we found a sketched but very convincing ME proof: along with it come the IOA modelling Filter-Lock (call it FL'), a convenient definition of winner and competitor processes, and the statements of two supporting invariants required for proving correctness (invariants 1F and 3F, stated and proved in pages 41 and 46). We adapt to our model both definitions and the two invariants:

Definition 4.1.1 (Winner Process on Level j)**.** In all states of the automaton, any process is winner on level j if and only if its i variable is greater than j or i equals j and the process is about to increment i.

$$\forall s_0 \in Q(FL), p \in \mathbb{P}, j \in \mathbb{N} : winner(s_0, p, j) = s_0.i_p > j \vee (s_0.i_p = j \wedge s_0.pc_p = pc10)$$

Definition 4.1.2 (Competitor Process on Level j)**.** In all state of the automaton, any process is a competitor on level j if and only if it is a winner on level j or its i equals j and it is about to scan some competitor or check *victim* variable.

$$\forall s_0 \in Q(FL), p \in \mathbb{P}, j \in \mathbb{N} :$$
$$comp(s_0, p, j) = winner(s_0, p, j) \vee \big(s_0.i_p = j \wedge s_0.pc_p \in \{pc5, pc6, pc7, pc8, pc9\}\big)$$

Employing the aforementioned definitions, invariants 1F and 3F, we could completed the formalization. FL differs from FL', in fact the FL IOA is an extended version designed to stick to the pseudocode of the algorithm as much as possible and it is not subject to several atomic steps. In fact, FL' encodes the evaluation of the innermost boolean guard via two atomic actions that correspond to our action sets $\{lock\_2T, lock\_2F, lock\_6T, lock\_6F, lock\_7T, lock\_7F, lock\_8T, lock\_8F, lock\_10\}$ and $\{lock\_2T, lock\_2F, lock\_9T, lock\_9F, lock\_10\}$. For the sake of comparison we report FL', let $PCs = \{pc1, pc2, pc3, pc4, pc5, pc6, pcIdle\}$ be the set of legal program counter values, then we define:

$$FL' = \Big(Signature(FL'), \big(Q(FL'), I(FL'), \Sigma(FL'), \delta(FL')\big)\Big)$$

- $Signature(FL')$:
  - $input(FL') = \{lock\_invoke_p\}$
  - $internal(FL') = \{lock\_1_p, lock\_2_p, lock\_3_p, lock\_4_p, unlock\_1_p\}$
  - $output(FL') = \{lock\_response_p\}$
- $Q(FL) = pc \times level \times i \times S \times victim$
- $I(FL') = \{\ q \in Q(FL') : \forall p \in \mathbb{P} : \big(pc_p = pcIdle \wedge i_p = 1 \wedge level_p = 0 \wedge S_p = \emptyset\big)\ \}$

State variable are typed as follows:

- $pc : \{1, \ldots, |\mathbb{P}|\} \to PCs$,
- $i : \{1, \ldots, |\mathbb{P}|\} \to \{1, \ldots, |\mathbb{P}| - 1\}$,
- $level : \{1, \ldots, |\mathbb{P}|\} \to \{0, \ldots, |\mathbb{P}| - 1\}$,
- $S : \{1, \ldots, |\mathbb{P}|\} \to 2^{\mathbb{P}}$,
- $victim : \{1, \ldots, |\mathbb{P}| - 1\} \to \{1, \ldots, |\mathbb{P}|\}$.

$\delta(FL')$	Comment:
lock_invoke$_p$:	Invocation.
pre: $pc_p = pcIdle$	
eff: $pc_p = pc1$	
lock_1$_p$:	Sets $level_p$.
pre: $pc_p = pc1$	
eff: $level_i = i_p \wedge pc_p = pc2$	
lock_2$_p$:	Sets $victim[i_p]$.
pre: $pc_p = pc2$	
eff: $victim[i_p] = p \wedge S_p = \{p\} \wedge pc_p = pc3$	
lock_3$_p$:	Scans j.
pre: $pc_p = pc3 \wedge j \notin S_p$	
eff: **if** $level_j < i_p$ **then**	
$S_p = S \cup \{j\}$	
if $\lvert S\rvert = \lvert\mathbb{P}\rvert$ **then**	
$S_p = \emptyset$	
if $i_p < \lvert\mathbb{P}\rvert - 1$ **then**	
$i_p = i_p + 1 \wedge pc = pc1$	
else	
$pc_p = pc5$	
else	
$pc_p = pc3$	
else	
$S_p = \emptyset \wedge pc_p = pc4$	

lock_4$_p$:	Checks $victim[i_p]$.
pre: $pc_p = pc4$	
eff: **if** $victim[i_p] \neq p$ **then**	
if $i_p < \lvert\mathbb{P}\rvert - 1$ **then**	
$i_p = i_p + 1 \wedge pc_p = pc1$	
else	
$pc_p = pc5$	
else	
$S_p = \{p\} \wedge pc_p = pc3$	
unlock_1$_p$:	Performs unlock.
pre: $pc_p = pc5$	
eff: $pc_p = pc6 \wedge level_p = 0 \wedge i_p = 1$	
lock_response$_p$:	Response.
pre: $pc_p = pc6$	
eff: $pc_p = pcIdle$	

Table 4.2: Transition function, $\delta(FL')$.

FL' misses a few state variables that FL exploits and that according to us are strictly required to prove Filter-Lock correct. Hence, we suspect the author decided to provide FL' to show the point of the proof rather than to actually point out all details. In our opinion FL' is designed to ease proof construction and is modelled to make apparent several useful facts during the unfolding of proof steps. However, the presence of atomicities in $lock\_3$ and $lock\_4$ is apparent an unconvincing: when the automaton identifies j to have a level strictly lower than i it instantaneously records j as scanned and depending on its level it either reroutes the control to the next cycle or to CS. The same is true for the latter action, the process first checks whether it is the designated victim and in case it is not, enters the next level or CS.

All lemmata and invariant we state in the next section are proved on the FL IOA, they are also provable on FL'. However, FL' must be adjusted adding missing variables and the majority of proof steps need to be reorganized: proofs shorten and simplify, but they do not change in principle. In fact, due to the structure of FL' guaranteeing that the competitor is evaluated atomically and that matches the entry in CS with the entry in the last level, three invariants become useless: invariants 6F, 7F and 19F (pages 51, 52 and 61).

That said we proceed with the major formalization that is carried out on model FL.

4.1.1 An Intuitive Mutual Exclusion Proof

In this section we expose the intuition supporting the proof employing a top-down approach, then we discuss and expose the four main intermediate results that support ME.

Filter-Lock provably satisfies mutual exclusion: from an intuitive perspective, the proof is simple, but from a formal perspective, in our opinion, it is not. The difficulties that this proof poses stem from the wide difference between the intuitive and the formal setting. In fact, Filter-Lock's inventor states that correctness is apparent [30], but three intricate invariants (invariants 1F to 3F, pages 41 and 46) are required to formally claim Filter-Lock correct. That said, we consider the proof from an intuitive perspective, this helps to sketch the main idea and to prepare the reader to the formal proof. To prove ME we must show that at most one process is in CS at a time, this task involves four major proofs:

- for each competition level j at most $\lvert\mathbb{P}\rvert - j$ processes can win the competition,
- the victim process is filtered out,
- the victim process is a competitor itself,
- the process that wins some competition is not a victim.

We map this four steps on as many intermediate invariants that support the correctness proof: these represent the main formalization challenge; more supporting invariants and lemmata are required but

perspectively they require a minor effort, they are stated and proved in the next section. Hence, we focus on the four invariants that support ME in a top-down fashion.

Invariant 5F is the main intermediate result: it states that the number of processes that win the competition on level j can not be more than $|\mathbb{P}|-j$. This predicate is strengthened by many invariants and lemmata, however this discussion is concerned with the role of invariants 1F, 3F and 4F that are the most relevant.

Invariant 5F (Cardinality is Filtered)**.** In all reachable states, for any competition level j below $|\mathbb{P}|$, the number of winners on level j is $|\mathbb{P}|-j$ at most.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow \forall j \in \mathbb{N} : 0 \leq j \leq |\mathbb{P}| - 1 \Rightarrow |s_0.W_j| \leq |\mathbb{P}| - j$$

Invariant 5F is proved by two nested inductions and a proof by contradiction: the first induction is on reachable states of the automaton, the second is on j. The first induction serves the purpose to make the basis of the second induction provable, that is to show that $|W_0| \leq |\mathbb{P}|$; the second induction is required to reach the conclusion. The basis of the induction on j reduces to a trivial evaluation of the cardinality, hence, we must focus on the inductive step: there we prove by contradiction that the cardinality of the winner set decreases, hence, we split the proof into two cases.

In the first case we suppose that the generic next level, $j+1$, contains more competitors than the allowed quantity ($|\mathbb{P}|-j$ instead of $|\mathbb{P}|-j-1$): this must turn out to be a contradictory supposition, that is, victim is a winner process. We can conclude so by joining the supposition with additional observations regarding winner sets W_j: it is clear that W_{j+1} is a subset of W_j, because all winners of level $j+1$ have won competition j as well. Therefore, we conclude that sets W_j and W_{j+1} share the same cardinality and if one is a subset of the other, then they must be the same set. By invariant 3F, victim process is as a competitor on level $j+1$, and consequently a winner on level j. Therefore, the victim process belongs to set W_j, and by previous inference it must belong to set W_{j+1}. By invariant 4F we observe that the victim can not be a winner, and we reach a contradiction. So, assuming there are more processes than allowed is absurd.

In the second case, we suppose that the next level $j+1$ contains the appropriate number of processes and this proves the initial statement. Also observe that this assumption does not support the same conclusion, namely we can not conclude that $W_j = W_{j+1}$. The formal proof is in page 50.

Invariant 4F is the second invariant we want to discuss, it is an interesting intermediate result that supports invariant 5F, it captures the intuition of the user regarding victim not being a winner. One major point supporting the intuition about the proof lies in showing that for each competition at level j, the process whose role is the one of *victim* on level j can not be part of the winner set on that level. This invariant states that if a competition is won by at least two processes, then victim in not one of them. Inspecting the pseudocode of algorithm 1, we notice that when two or more processes concurrently execute the innermost while-loop, the one designated as victim remains stuck on the cyclic evaluation of competitor's *level* and *victim* variables. Consequently, the designated *victim* process loses the competition and waits in busy-waiting while its competitor(s) become winner(s) and break the while-loop.

Invariant 4F (Set Theoretic Peterson's Principle)**.** In all reachable states, for all positive indexed levels, if there are at least two winners then victim is not one of them.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow \forall j, n \in \mathbb{N} : \big(j > 0 \wedge |s_0.W_j| = 2 + n\big) \Rightarrow s_0.victim[j] \notin s_0.W_j$$

Invariant 4F is strengthened by invariant 1F (page 40) and is proved in page 48 by two nested inductions, the first one on reachable states of the automaton, the second one on n. Induction over reachable states serves the purpose to generate an inductive hypothesis that is preserved, the second induction instead is useful for generalizing the application of invariant 1F arbitrarily many times.

Invariant 3F states that if there is a competitor on level j, the victim of that level is a competitor on that level. It is useful for proving the contradiction case of invariant 5F. Invariant 3F is defined by words in [25], the formal statement here provided is our interpretation:

Invariant 3F (Victim is a Competitor)**.** In all reachable states, if there is a competitor process on level j, variable $victim[j]$ records the ID of a competitor.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow \forall j \in \mathbb{N} : \big(\exists p \in \mathbb{P} : comp(s_0, p, j)\big) \Rightarrow comp(s_0, s_0.victim[j], j)$$

Invariant 3F is strengthened by invariant 1F and is proved in page 46 by induction over reachable state of the automaton.

Invariant 1F is a the last main result we want to employ, it is a very sophisticated predicate: the assertion is formed by two sub-predicates that together can be proved by induction. The proof of this property also requires many supporting invariants. For the sake of this discussion, we call the conjunct that defines processes p and q competitors, the "left predicate". It serves the purpose to make the whole assertion provable, while the right predicate is the result we are actually interested in; this is why, in due time, we forget the statement of invariant 1F and employ the right predicate alone: it corresponds to invariant 2F. This invariant represents the whole formalization challenge, its proof is the longest and most intricate. Invariant 1F is defined by words in [25], the formal statement here provided is our interpretation; we could not prove the version as given, hence, we adapted it and proved the resulting invariant: our version rules out the case $j = 0$, however this is not an impediment.

Invariant 1F. In all reachable states, for all pairs of distinct processes p and q, for all positive indexed levels j the following are both true: if p and q are defined competitors on level j, p is busy testing the level of some competitor and q is recorded having *level* variable lesser than p's, then p is not victim on level j; if p and q are defined respectively winner and competitor on level j, then p is not victim on level j.

$$\begin{gathered}\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow \\ \left(\begin{array}{c}\forall p, q \in \mathbb{P}, j \in \mathbb{N} : \\ j > 0 \wedge comp(s_0, p, j) \wedge s_0.pc_p \in \{pc5, pc6, pc7, pc8\} \wedge \\ comp(s_0, q, j) \wedge q \in s_0.S_p \wedge p \neq q \Rightarrow s_0.victim[j] \neq p\end{array}\right) \\ \wedge \\ \Big(\forall p, q \in \mathbb{P}, j \in \mathbb{N} : j > 0 \wedge winner(s_0, p, j) \wedge comp(s_0, q, j) \wedge p \neq q \Rightarrow s_0.victim[j] \neq p\Big)\end{gathered}$$

In the next section all required lemmata and invariants are stated and proved.

4.2 A Correctness Proof for Filter-Lock

In this section we provide a proof for each invariant or lemma we employ. The section is split into four parts, each one gathering assertions by their role. We start with most interesting and sophisticate invariants and carry on with increasingly simpler properties:

- Section 4.2.1 (page 41) gathers the four properties we mentioned in the preliminaries and two additional invariants that prove Filter-Lock satisfies mutual exclusion. Invariants 1F to 7F are proved in this section.
- Section 4.2.2 (page 52) gathers evolved behaviours of processes and variables. These are used frequently to support high-level invariants. Invariants 8F to 13F are proved in this section.
- Section 4.2.3 (page 58) gathers many invariants specifying how variables of the model vary. These are the simplest properties and are of relative interest. Often the property they encode can be deduced by inspection of the model and the majority is used just a few times to prove higher results. Invariants 14F to 23F are proved in this section.
- Section 4.2.4 (page 66) gathers three observations that from a non-mechanical perspective are trivial, however the Proof Assistant (PA) does not recognize them as such.

The reader has two major options: to start with the inspection of high-level invariants' proofs and proceed with increasingly simpler ones or to start from the last category and proceed backward. The first approach focuses on understanding central invariants, the second on clarifying each step. In each

subsection, assertions are provided and proved in topological order, that is, properties required to prove others are declared and proved first.

To ease the inspection of the formalization, we provide the dependency graph depicting the support of some invariant with respect to another; we depict each assertion differently depending on the role it plays. Non-strengthened invariants are depicted by squares, conversely strengthened invariants are represented by circles and ingoing edges relate them to their supporting invariants. The relation here shown is $Supports(a, b) = a \to b$: it is read "invariant a supports invariant b". We restrict the relation here depicted to invariants only: lemmata are not reported due to the alternation of ubiquity to sparseness that makes the dependency graph difficult to read.

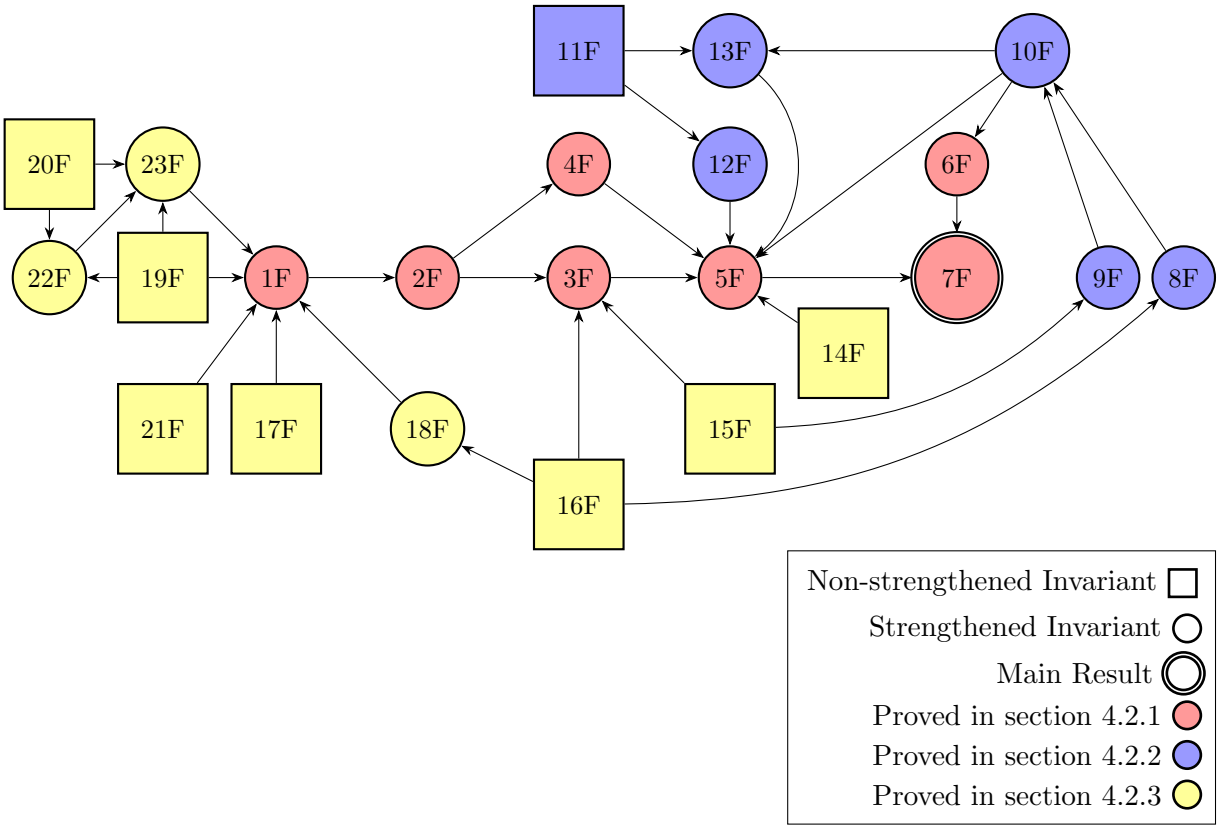

Figure 4.1: Results support dependency graph

4.2.1 High-Level invariants

We now state and prove high-level invariants, those that directly support the correctness proof: the last property is the main result itself.

Invariant 1F. In all reachable states, for all pairs of distinct processes p and q, for all positive indexed levels j the following are both true: if p and q are defined competitors on level j, p is busy testing the level of some competitor and q is recorded having *level* variable lesser than p's, then p is not victim on level j; if p and q are defined respectively winner and competitor on level j, then p is not victim on level j.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow$$
$$\left(\begin{array}{c} \forall p, q \in \mathbb{P}, j \in \mathbb{N} : \\ j > 0 \wedge comp(s_0, p, j) \wedge s_0.pc_p \in \{pc5, pc6, pc7, pc8\} \wedge \\ comp(s_0, q, j) \wedge q \in s_0.S_p \wedge p \neq q \Rightarrow s_0.victim[j] \neq p \end{array}\right)$$
$$\wedge$$
$$\Big(\forall p, q \in \mathbb{P}, j \in \mathbb{N} : j > 0 \wedge winner(s_0, p, j) \wedge comp(s_0, q, j) \wedge p \neq q \Rightarrow s_0.victim[j] \neq p\Big)$$

For convenience we recall the definition of winner and competitor:

$$winner(s_0, p, j) = s_0.i_p > j \vee (s_0.i_p = j \wedge s_0.pc_p = pc10)$$
$$comp(s_0, p, j) = s_0.i_p > j \vee (s_0.i_p = j \wedge s_0.pc_p = pc10) \vee (s_0.i_p = j \wedge s_0.pc_p \in \{pc5, pc6, pc7, pc8, pc9\})$$

Proof. By induction on reachable states of the automaton.
Basis. Let s_0 be an initial state of the automaton. For convenience, we split the proof immediately; there are two cases:
Case 1. We prove the left predicate: $L(s_0)$.

$$s_0 \in I(FL) \vdash \left(\begin{array}{c} \forall p, q \in \mathbb{P}, j \in \mathbb{N} : \\ j > 0 \wedge comp(s_0, p, j) \wedge s_0.pc_p \in \{pc5, pc6, pc7, pc8\} \wedge \\ comp(s_0, q, j) \wedge q \in s_0.S_p \wedge p \neq q \Rightarrow s_0.victim[j] \neq p \end{array} \right)$$

Let p and q be generic processes in state s_0. Observe that in initial states all processes have set $S = \emptyset$, this makes the consequent vacuously true. □
Case 2. We prove the right predicate: $R(s_0)$.

$$s_0 \in I(FL) \vdash \forall p, q \in \mathbb{P}, j \in \mathbb{N} : j > 0 \wedge winner(s_0, p, j) \wedge comp(s_0, q, j) \wedge p \neq q \Rightarrow s_0.victim[j] \neq p$$

Let p and q be generic processes in state s_0. We assume true that $j > 0$, p is a winner, q is a competitor and $p \neq q$. Observe that initial states are characterized by having variable $i = 1$ for all processes. Consequently, if p is a winner, then $j = 0$. This is so because all PC based assumptions are invalidated by the PC value set in initial states. This deduction contradicts the assumption that $j > 0$. □
Inductive step. Let s_0 and s_1 be states of the automaton, let s_0 be reachable; we assume the invariant holds in state s_0 by inductive hypothesis.

$$L(s_0), R(s_0), reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash$$
$$L(s_1) \wedge R(s_1)$$

We split the proof immediately:
Case 1. We prove the left predicate: $L(s_1)$.

$$L(s_0), R(s_0), reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash$$
$$L(s_1)$$

Let p and q be generic processes in state s_0. We assume true the premise of the consequent, there are two cases:
Case 1.1. Suppose that p or q take a step: $proc(\pi) = p \vee proc(\pi) = q$. There are two cases:
Case 1.1.1. Suppose that process p takes a step: $proc(\pi) = p$. Process q is suspended, consequently, by lemma 2F (page 57) we know that it is a competitor in the pre-state as well. Observe that the following actions trivially invalidate the assumption that p is a competitor due to a PC based conflict:

$$\pi \in \left\{ \begin{array}{c} lock\_invoke, lock\_response, lock\_1, lock\_2F, lock\_2T, lock\_3, \\ lock\_6F, lock\_7F, lock\_8F, lock\_9F, lock\_10, unlock\_1 \end{array} \right\}$$

Consequently, we focus on:

$$\pi \in \left\{ \; lock\_4, lock\_5, lock\_6T, lock\_7T, lock\_8T, lock\_9T \; \right\}$$

- $\pi = lock\_4$, this action sets $victim = p$, $S = \{p\}$ and $C = S^c$. Observe that by hypothesis $p \neq q$, the transition invalidates the assumption that $q \in s_1.S_p$. □
- $\pi = lock\_5$, this case is analogous to case $\pi = lock\_4$. □
- $\pi = lock\_6T$, this action identifies and choose a new competitor to scan: qq. Observe that the inductive hypothesis is preserved because no involved variable is modified. □

- $\pi = lock\_7T$, this action tests qq's level with respect to i_p, finds it lesser and records qq in set S. There are two cases:

  Case 1.1.1.1. Suppose that $q = s_0.qq_p$. Observe that by assumption p is a competitor in the post-state, hence:

  $$comp(s_1, p, j) = s_1.i_p > j \vee (s_1.i_p = j \wedge s_1.pc_p = pc10) \vee (s_1.i_p = j \wedge s_1.pc_p \in \{pc5, pc6, pc7, pc8, pc9\})$$

  Observe that this action sets $pc_p = pc8$, invalidating the middle clause. Hence, we focus on outer ones, there are two cases:

  Case 1.1.1.1.1. $s_1.i_p > j$. Observe that this action does not modify i_p, consequently p is a winner in the pre-state as well. By assumption $q = qq_p$ and we can not prove that $q \in s_0.S_p$: by invariant 19F (page 61) we can prove the exact converse, therefore, we want to reach a contradiction. By $R(s_0)$, we can prove that $s_0.victim[j] \neq p$. $R(s_0)$'s premises are satisfied: by assumption $j > 0$, q is a competitor, $p \neq q$ and we deduced that p is a winner. The transition does not modify $victim[j]$ and the conclusion comes. □

  Case 1.1.1.1.2. $s_1.i_p = j \wedge s_1.pc_p \in \{pc5, pc6, pc7, pc8, pc9\}$. The action does not modify i_p, hence, $s_1.i_p = s_0.i_p$. Focus on the fact that q is a competitor: we are interested in reaching a contradiction, that is, q can not be a competitor because its *level* variable conflicts with i_p. We expand the definition of competitor and analyse three cases:

  Case 1.1.1.1.2.1. $s_0.i_q > j$. We assumed that $q = s_0.qq_p$ and by construction the current action is executed when $s_0.level_{qq_p} < s_0.i_p$. By invariant 18F (page 61) we know that $s_0.i_q = s_0.level_q \vee s_0.i_q = s_0.level_q - 1$. It is sufficient to observe that rewriting, we contradict the precondition of the action: $j < s_0.level_q$. □

  Case 1.1.1.1.2.2. $s_0.i_q = j \wedge s_0.pc_q = pc10$. Observe that by invariant 17F (page 60) we know that $s_0.i_q = s_0.level_q$. Consequently, $s_0.level_q = j$, this contradicts the precondition of the current action. □

  Case 1.1.1.1.2.3. $s_0.i_q = j \wedge s_0.pc_q \in \{pc5, pc6, pc7, pc8, pc9\}$. This case is analogous to the previous one. □

  Case 1.1.1.2. Conversely, suppose that $q \neq s_0.qq_p$. Then the transition did not record q in S and by assumption we have that $q \in s_1.S_p$; therefore, q was in set S before the transition executes: $q \in s_0.S_p$. Observe that p is a competitor is the post-state and since the transition does not change variable i it is a competitor in the pre-state as well. These observations satisfy the inductive hypothesis that is preserved by the transition. □

- $\pi = lock\_8T$, this case is analogous to case $\pi = lock\_6T$. □
- $\pi = lock\_9T$, this case is analogous to case $\pi = lock\_4$. □

This concludes the case where p is the generic executor. □

Case 1.1.2. Conversely, suppose that process q takes a step: $proc(\pi) = q$. Observe that p is suspended and by lemma 2F (page 57) we know it is a competitor in the pre-state as well. We plan to satisfy the inductive hypothesis and preserve it, hence, we show its premises are true. We assumed that $j > 0$, $p \neq q$, $s_0.pc_p = s_1.pc_p$ and deduced that p is a competitor in the pre-state. We still must show that q is a competitor in the pre-state and that $q \in s_0.S_p$. We assumed $q \in s_1.S_p$, observe that only p can modify S_p and since it is suspended $q \in s_0.S_p$. The last thing we must prove is $comp(s_0, q, j)$. We do so expanding the definition of competitor in the post-state, there are three cases:

Case 1.1.2.1. $s_1.i_q > j$. We assumed that $j > 0$, joining the two assumptions we get that $s_1.i_q \geq 2$. Observe that actions setting $s_1.i_q = 1$, trivially invalidate the assumption:

$$\pi \in \{\ lock\_1, unlock\_1\ \}$$

Remaining actions either increase or leave i_q unchanged, we analyse them.

- $\pi = lock\_invoke$, this action invokes the algorithm leaving i_q untouched. Hence, $s_1.i_q > j$ and $s_0.i_q > j$. This qualifies q as a winner in the pre-state and consequently as a competitor. This deduction satisfies the inductive hypothesis that is preserved because the transition does not rewrite $victim[j]$. □

- $\pi = lock\_2T$, this case is analogous to case $\pi = lock\_invoke$. □
- $\pi = lock\_2F$, this case is analogous to case $\pi = lock\_invoke$. □
- $\pi = lock\_3$, this case is analogous to case $\pi = lock\_invoke$. □

- $\pi = lock\_4$, this actions sets $victim[i_q] = q$. Observe that by assumption $s_1.i_q > j$ and $s_0.i_q > j$. Hence, the status of winner is preserved backward to the pre-state, it satisfies the inductive hypothesis that is in turn preserved because $victim[j]$ is not rewritten. □

- $\pi = lock\_5$, this case is analogous to case $\pi = lock\_invoke$. □
- $\pi = lock\_6T$, this case is analogous to case $\pi = lock\_invoke$. □
- $\pi = lock\_6F$, this case is analogous to case $\pi = lock\_invoke$. □
- $\pi = lock\_7T$, this case is analogous to case $\pi = lock\_invoke$. □
- $\pi = lock\_7T$, this case is analogous to case $\pi = lock\_invoke$. □
- $\pi = lock\_8T$, this case is analogous to case $\pi = lock\_invoke$. □
- $\pi = lock\_8F$, this case is analogous to case $\pi = lock\_invoke$. □

- $\pi = lock\_9T$, this action sets $S = \{q\}$ and $C = S^c$. Observe that by assumption $s_1.i_q > j$ and $s_0.i_q > j$. Hence, the status of winner is preserve to the pre-state, it satisfies the inductive hypothesis that is in turn preserved. □

- $\pi = lock\_9F$, this case is analogous to case $\pi = lock\_9T$. □

- $\pi = lock\_10$, this action increments i_q. Then $s_0.i_q \geq j$. There are two cases:

  Case 1.1.2.1.1. $s_0.i_q > j$. Hence, the status of winner is preserved to the pre-state, it satisfies the inductive hypothesis that is in turn preserved. □

  Case 1.1.2.1.2. $s_0.i_q = j$. Observe that the second clause of the winner predicate is satisfied, because $s_0.i_q = j$ and $s_0.pc_q = pc10$. This satisfied the inductive hypothesis that is in turn preserved. □

- $\pi = lock\_response$, this case is analogous to case $\pi = lock\_invoke$. □

Case 1.1.2.2. $s_1.i_q = j \wedge s_1.pc_q = pc10$. Observe that the only actions that set $s_1.pc_q = pc10$ are:

$$\pi \in \{\ lock\_6F, lock\_8F, lock\_9F\ \}$$

No action in this set modifies i_q, hence, $s_0.i_q = j$. Also observe that $pc6, pc8, pc9$ satisfy the definition of competitor. Consequently, $comp(s_0, q, j)$ is satisfied and with it the inductive hypothesis that is preserved. □

Case 1.1.2.3. $s_1.i_q = j \wedge s_1.pc_q \in \{pc5, pc6, pc7, pc8, pc9\}$. Observe that the only actions that satisfy the PC based assumption are:

$$\pi \in \{\ lock\_4, lock\_5, lock\_6T, lock\_7T, lock\_7F, lock\_8T, lock\_9T\ \}$$

Consequently, we analyse them:

- $\pi = lock\_4$, this actions sets $victim[j] = q$. Observe that the conclusion comes by the effect of the action, because it is obviously true that $victim[j] \neq p$. □

- $\pi = lock\_5$, this action sets $S = \{q\}$ and $C = S^c$. Observe this action does not modify i_q, consequently $s_0.i_q = s_1.i_q$, this preserves the status of q as a competitor that in turn satisfies the inductive hypothesis that is preserved. □

- $\pi = lock\_6T$, this case is analogous to case $\pi = lock\_5$. □
- $\pi = lock\_7T$, this case is analogous to case $\pi = lock\_5$. □
- $\pi = lock\_7F$, this case is analogous to case $\pi = lock\_5$. □
- $\pi = lock\_8T$, this case is analogous to case $\pi = lock\_5$. □

- $\pi = lock\_9T$, this action tests $victim[j]$. The precondition specifies that this action is executed when $victim[j] = q$. Observe this action does not modify i_q, consequently $s_0.i_q = s_1.i_q$; this preserves the status of q as a competitor that in turn satisfies the inductive hypothesis that is preserved. □

This concludes the case where q is the generic executor. □
Case 1.2. Conversely, suppose that neither p nor q take a step: $proc(\pi) \neq p \wedge proc(\pi) \neq q$. We assumed that $j > 0$, p and q are competitors in the post-state. Then by lemma 2F (page 57), applied twice, we know that p and q are competitors in the pre-state as well. By their inaction we know that their PC values and set S do not change. These observations satisfy the inductive hypothesis that is preserved by all actions. □
This concludes the proof of the left predicate. □
Case 2. We prove the right predicate: $R(s_1)$.

$$L(s_0), R(s_0), reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash R(s_1)$$

Let p and q be generic processes in state s_0. We assume true the premise of the consequent, there are two cases:
Case 2.1. Suppose that p or q take a step: $proc(\pi) = p \vee proc(\pi) = q$. There are two cases:
Case 2.1.1. Suppose that p takes a step: $proc(\pi) = p$. Observe that q is suspended and by lemma 2F (page 57) its status of competitor is extended to the pre-state. We plan to exploit the inductive hypothesis: we have assumed $j > 0$, $p \neq q$, $winner(s_1, p, j)$ and we have deduced $comp(s_0, q, j)$. We must show that p is a winner in the pre-state. There are two cases:
Case 2.1.1.1. $s_1.i_p > j$. This case is analogous to **case 1.1.2.1** (page 43). □
Case 2.1.1.2. $s_1.i_p = j \wedge s_1.pc_p = pc10$. Observe that the only actions that do not invalidate the PC based assumption are:

$$\pi \in \{\ lock\_6F, lock\_8F, lock\_9F\ \}$$

- $\pi = lock\_6F$, this action adds p to W_{i_p}. Observe that this action does not modify i_p, hence, $s_0.i_p = j$; also observe that this action is executed when $pc_p = pc6$: these two observations are sufficient to qualify p as a competitor. Observe also that the inductive hypothesis is of no use here because the definition of winner does not apply to p in the pre-state. Hence, we employ the "left predicate", $L(s_0)$. Recall we assumed that $j > 0$, $p \neq q$ and deduced $comp(s_0, p, j)$ and $comp(s_0, q, j)$. The last thing we must show is that $q \in s_0.S_p$. There are two cases:

  Case 2.1.1.2.1. Suppose that $q \in s_0.S_p$. This assumption satisfied the "left predicate" and proves the conclusion. □

  Case 2.1.1.2.2. Conversely, suppose that $q \notin s_0.S_p$. Observe that this action is executed only when $s_0.C = \emptyset$. By invariant 21F (page 63) we know that $s_0.S_p = \top$: hence $q \notin s_0.S_p$ is a contradiction. □

- $\pi = lock\_8F$, this action sets $S = \{p\}$ and $C = S^c$. Observe that this action does not modify i_p, consequently $s_0.i_p = j$; also observe that this action is executed when $pc_p = pc8$: these two observations are sufficient to qualify p as a competitor. Observe that the inductive hypothesis is of no use here because the definition of winner does not apply to p in the pre-state. Hence, we employ the "left predicate", $L(s_0)$. Recall we assumed that $j > 0$, $p \neq q$ and deduced $comp(s_0, p, j)$ and $comp(s_0, q, j)$. The last thing we must show is that $q \in s_0.S_p$. There are two cases:

  Case 2.1.1.2.1. Suppose that $q \in s_0.S_p$. This assumption satisfied the "left predicates" and proves the conclusion. □

Case 2.1.1.2.2. Conversely, suppose that $q \notin s_0.S_p$. Observe that this action is executed only when $|s_0.S_p| = |\mathbb{P}|$. By invariant 23F (page 64) we know that $|s_0.C_p| = 0$, therefore, $s_0.C_p = \emptyset$. By invariant 21F (page 63) we know that $s_0.S_p = \top$: hence $q \notin s_0.S_p$ is a contradiction. □

- $\pi = lock\_9F$, this action adds p's ID to W_{i_p}. Observe this actions is executed only when $s_0.victim[j] \neq p$, consequently enforcing the property. □

This concludes the case where p is the generic executor. □

Case 2.1.2. Conversely, suppose that q takes a step: $proc(\pi) = q$. Observe that p is suspended and by lemma 1F (page 57) the status of winner is extended to the pre-state. We plan to exploit the inductive hypothesis: we have assumed that $j > 0$, $p \neq q$, $comp(s_1, q, j)$ and we have deduced that $winner(s_0, p, j)$. We must show that q is a competitor in the pre-state. Observe this case is analogous to **cases 1.1.2.1, 1.1.2.2 and 1.1.2.3** (page 43) of the left predicate. □

Case 2.2. Conversely, suppose that neither p nor q take a step: $proc(\pi) \neq p \wedge proc(\pi) \neq p$. By lemmata 1F and 2F (page 57) we know that p and q preserve their status of *winner* and *competitor* in the pre-state. Consequently, the inductive hypothesis is satisfied and no transition invalidates it. □

Invariant 2F (Peterson's Principle)**.** In all reachable states, for all positive indexed levels j, if p and q are defined respectively winner and competitor on level j, then p is not victim on level j.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow$$
$$\forall p, q \in \mathbb{P}, j \in \mathbb{N} : j > 0 \wedge winner(s_0, p, j) \wedge comp(s_0, q, j) \wedge p \neq q \Rightarrow s_0.victim[j] \neq p$$

Proof. By direct proof. Let p and q be generic process in state s_0, let j be a generic natural. Apply invariant 1F to prove the conclusion. □

Invariant 3F (Victim is Competitor)**.** In all reachable states, if there is a competitor process on level j, variable $victim[j]$ records the ID of a competitor.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow \forall j \in \mathbb{N} : \big(\exists p \in \mathbb{P} : comp(s_0, p, j)\big) \Rightarrow comp(s_0, s_0.victim[j], j)$$

For convenience we recall the definition of competitor:

$$comp(s_0, p, j) = winner(s_0, p, j) \vee (s_0.i_p = j \wedge s_0.pc_p \in \{pc5, pc6, pc7, pc8, pc9\})$$
$$\Leftrightarrow$$
$$comp(s_0, p, j) = s_0.i_p > j \vee (s_0.i_p = j \wedge s_0.pc_p = pc10) \vee (s_0.i_p = j \wedge s_0.pc_p \in \{pc5, pc6, pc7, pc8, pc9\})$$

Proof. By induction on reachable states of the automaton.
Basis. Let s_0 be an initial state of the automaton.

$$s_0 \in I(FL) \vdash \forall j \in \mathbb{N} : \big(\exists p \in \mathbb{P} : comp(s_0, p, j)\big) \Rightarrow comp(s_0, s_0.victim[j], j)$$

Let j be a generic natural. We assume true the existence of a generic competitor p in state s_0. Observe that all processes in initial states are characterized by having $pc = pcIdle$ and $i = 1$. Consequently, the middle and rightmost clauses of the competitor definition are unsatisfed and the only case we need to analyse is: $i_p > j$. By this assumption we deduce that $j = 0$, consequently p is a winner because $i_p > 0$. Consequently, $victim[j]$ is a competitor because it is a winner in the initial state. □
Inductive step. Let s_0 and s_1 be states of the automaton, let s_0 be reachable; we assume the invariant holds in state s_0 by inductive hypothesis.

$$\forall j \in \mathbb{N} : \big(\exists p \in \mathbb{P} : comp(s_0, p, j)\big) \Rightarrow comp(s_0, s_0.victim[j], j), reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash$$
$$\forall j \in \mathbb{N} : \big(\exists p \in \mathbb{P} : comp(s_1, p, j)\big) \Rightarrow comp(s_1, s_1.victim[j], j)$$

Let j be a generic natural. We assume true the existence of a generic competitor p in the post-state. There are two cases:

Case 1. Suppose that p or $s_0.victim[j]$ take a step: $proc(\pi) = p \vee proc(\pi) = s_0.victim[j]$. There are two cases:
Case 1.1. Suppose that p and $s_0.victim[j]$ are the same process, then $s_0.victim[j]$ is a competitor. There are two cases:
Case 1.1.1. Suppose that $s_0.victim[j] = s_1.victim[j]$. Observe that all actions not modifying *victim* trivially preserve the status of the process and the only action we need to analyse is $\pi = lock\_4$.

- $\pi = lock\_4$, this action sets *victim* variable. Observe this action sets $s_1.victim[j] = p$ and $p = s_0.victim[j]$, consequently satisfying the consequent. □

Case 1.1.2. Conversely, suppose that $s_0.victim[j] \neq s_1.victim[j]$. Observe that all actions that do not modify the *victim* variable invalidate the assumption, consequently we focus on $\pi = lock\_4$.

- $\pi = lock\_4$, this action sets *victim* variable. Observe this action sets $s_1.victim[j] = p$ and $p = s_0.victim[j]$, this is a contradiction. □

Case 1.2. Suppose that p and $s_0.victim[j]$ are the different processes. There are two cases:
Case 1.2.1. Suppose that p is the process taking a step: $proc(\pi) = p$. In this case we must show the inductive hypothesis is helping, to do so we assume its premise is satisfied and consequently prove this fact. This generates two cases:
Case 1.2.1.1. Suppose the inductive hypothesis is satisfied, then $s_0.victim[j]$ is a competitor in the pre-state. Observe that only the action rewriting *victim* needs to be analysed, because the others trivially preserve the status of $s_0.victim[j]$ that is suspended. Consequently we focus on:

- $\pi = lock\_4$, this action sets $s_1.victim[j] = p$. Process p is a competitor in the pre-state and since the transition does not modify i_p in the post-state as well. □

Case 1.2.1.2. We now must prove the inductive hypothesis is satisfied: we do so by contradiction. There are two cases:
Case 1.2.1.2.1. Suppose that p is not a competitor on level j in the pre-state.

- $\pi = lock\_1$, this action sets $i_p = 1$. We assumed p to be a competitor on level j in the post-state. This action invalidates the two rightmost clauses of the competitor definition, we focus on the leftmost: $s_1.i > j$. By invariant 16F we know that $s_0.i_p = 1$, we infer that $j = 0$, therefore, p is a winner and a competitor of level 0; this contradicts the assumption that p is not a competitor. □

- $\pi = unlock\_1$, this actions sets $i_p = 1$ and lowers the competition level to 0. Observe that the action invalidates the two rightmost clauses of predicate competitor, we focus on the leftmost: $s_1.i_p > j$. We infer that $j = 0$, therefore, by invariant 15F we know that $s_0.i_p = |\mathbb{P}|$. Hence p is a winner and a competitor of level 0; this contradicts the assumption that p is not a competitor. □

Case 1.2.1.2.2. Conversely, suppose that p is a competitor in level j, then there exists a competitor in that level and that proves the premise of the inductive hypothesis. □
Case 1.2.2. Conversely, suppose that $s_0.victim[j]$ is the process taking a step: $proc(\pi) = s_0.victim[j]$, then process p is suspended. By lemma 2F (page 57) we know that process p is a competitor in the pre-state. This observation satisfies the inductive hypothesis, that is, $s_0.victim[j]$ is a competitor: $comp(s_0, p, j)$. We must develop the transition for all possible actions. Observe that the status of competitor is preserved by all actions that do not modify i, consequently we focus on:

$$\pi \in \{lock\_1, lock\_10, unlock\_1\}$$

- $\pi = lock\_1$, this actions sets $i_{s_0.victim[j]} = 1$. Observe that this action trivially invalidates the two rightmost clauses of competitor definition, consequently we focus on the remaining clause: $s_0.i_{s_0.victim[j]} > j$. By invariant 16F (page 59) we know that $s_0.victim[j]$'s i variable equals 1, therefore, $j = 0$. The definition of competitor is satisfied in the consequent. □

- $\pi = lock\_10$, this action increases $i_{s_0.victim[j]}$. We analyse the definition of competitor in the pre-state, there three cases:

  Case 1.2.2.1. $s_0.i_{s_0.victim[j]} > j$. Observe that the increment enforces $s_1.i_{s_0.victim[j]} > j$ qualifying *victim* as winner and consequently as a competitor. □

  Case 1.2.2.2. $s_0.i_{s_0.victim[j]} = j \wedge s_0.pc_p = pc10$. This case is analogous to the previous. □

  Case 1.2.2.3. $s_0.i_{s_0.victim[j]} = j \wedge s_0.pc_p \in \{pc5, pc6, pc7, pc8, pc9\}$. Observe that the assumption invalidate the precondition of the action due to a PC based conflict. □

- $\pi = unlock\_1$, this action sets $i_{s_0.victim[j]} = 1$. Observe that this action trivially invalidates the rightmost clause of the competitor definition, consequently, we focus on the remaining clause: $winner(s_0, s_0.victim[j], j)$. We plan to employ invariant 2F (page 46), to do so we must satisfy its premise: we deduced that process q is a competitor on level j and that $p \neq s_0.victim[j]$. We must show that $j > 0$; to do so we observe that $comp(s_1, s_0.victim[j], j)$ is a consequent. We also observe that this action does not satisfy the rightmost clauses of the definition, and the only satisfiable clause is: $s_1.i_{s_0.victim[j]} > j$. By the transition the inequality becomes $1 > j$. By rule $(\neg R)$, this corresponds to the antecedent $\neg(1 > j)$, that is $1 \leq j$. Hence, $j > 0$, invariant 2F can be applied and by it we know that $s_0.victim[j] \neq s_0.victim[j]$: this is a contradiction. □

Case 2. Conversely, suppose the generic executor is neither p nor $s_0.victim[j]$. By lemma 2F (page 57) we know that p is a competitor in the pre-state as well. This satisfies the inductive hypothesis that allows us to conclude that $s_0.victim[j]$ is a competitor in the pre-state. Since neither p nor $s_0.victim[j]$ take a step, the invariant is preserved by all actions. □

Invariant 4F (Set Theoretic Peterson's Principle)**.** In all reachable states, for all positive indexed levels, if there are at least two winners, then victim is not one of them.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow \forall j, n \in \mathbb{N} : j > 0 \wedge |s_0.W_j| = 2 + n \Rightarrow s_0.victim[j] \notin s_0.W_j$$

This proof involves contemporarily two nested inductions, one on reachable states of the automaton, the other on the cardinality of the set n.

Proof. By induction on reachable states of the automaton.
Basis. Let s_0 be an initial state of the automaton.

$$s_0 \in I(FL) \vdash \forall j, n \in \mathbb{N} : j > 0 \wedge |s_0.W_j| = 2 + n \Rightarrow s_0.victim[j] \notin s_0.W_j$$

Let j and n be generic naturals. Assume that $j > 0$ and $|W_j| = 2 + n$; observe that initial states are characterized by having $W_0 = \top$ and $W_{j \geq 0} = \emptyset$. Then it is false that $|\emptyset| = 2 + n$. Vacuously satisfying the consequent. □
Inductive step. Let s_0 and s_1 be states of the automaton, let s_0 be reachable; we assume the invariant holds in state s_0 by inductive hypothesis.

$$\forall j, n \in \mathbb{N} : j > 0 \wedge |s_0.W_j| = 2 + n \Rightarrow s_0.victim[j] \notin s_0.W_j, reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash$$
$$\forall j, n \in \mathbb{N} : j > 0 \wedge |s_1.W_j| = 2 + n \Rightarrow s_1.victim[j] \notin s_1.W_j$$

Let j be a generic natural. We induct on n.
Basis. We assume that $|s_1.W_j| = 2$. By lemma 5F (page 67) we know that $s_1.W_j$ contains two distinct elements, let them be x and y; by the same lemma we also know that if $x \in W_j$, $y \in W_j$ and $r \in W_j$ then x is y, or y is r, or r is x. Instantiate r to $s_1.victim[j]$; by rule $(\neg R)$ we know that $s_1.victim[j] \in W_j$, consequently one of the elements that belong to W_j must be an alias. Recall that up to now we assumed that $x \in W_j$, $y \in W_j$ and $s_1.victim[j] \in W_j$; by invariant 10F (page 55) we know they are winners. By invariant 2F (page 46), applied twice, we know that since x and y are winners, $s_1.victim[j] \neq x$ and $s_1.victim \neq y$. This contradicts the hypothesis that some element is an alias. □

Inductive step. The proof context is:

$$\begin{array}{l} reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1, \\ \forall j, n \in \mathbb{N} : j > 0 \wedge |s_0.W_j| = 2 + n \Rightarrow s_0.victim[j] \notin s_0.W_j \vdash \\ \forall J \in \mathbb{N} : (j > 0 \wedge |W_j| = 2 + J \Rightarrow s_1.victim[j] \notin W_j) \Rightarrow \\ \qquad \big(j > 0 \wedge |W_j| = 2 + (J+1) \Rightarrow s_1.victim[j] \notin W_j\big) \end{array}$$

Let J be the a generic natural. Observe that the only actions that need analysis are those that modify the cardinality of W_j or rewrite *victim* variable:

$$\pi \in \{lock\_4, lock\_6F, lock\_8F, lock\_9F, unlock\_1\}$$

We focus our attention on them:

- $\pi = lock\_4$, this action sets $victim[i_{proc(\pi)}] = proc(\pi)$, leaving W_j untouched. Observe that the cardinality of W_j is consequently untouched as well. There are two cases:

  Case 1. Suppose that $j \neq s_0.i_{proc(\pi)}$, then the victim of another level is set and the invariant is trivially preserved. □

  Case 2. Conversely, suppose that $j = s_0.i_{proc(\pi)}$. In this case the inductive hypothesis descending from the induction on reachable states is of no use because the transition overwrites $s_0.victim[j]$. Since we want to prove that $s_1.victim[j] \notin s_1.W_j$, by rule $(\neg R)$ we infer that $s_1.victim[j] \in s_1.W_j$. Observe that if $proc(\pi)$ sets itself as victim, it belongs to $s_1.W_j$ and by invariant 10F (page 55) it satisfies the definition of winner. Expand the definition of winner and observe there are two cases:

  Case 2.1. $s_1.i_{proc(\pi)} > j$. This is a contradiction because by hypothesis $j = s_0.i_{proc(\pi)}$. □

  Case 2.2. $s_1.i_{proc(\pi)} = j \wedge s_1.pc_{proc(\pi)} = pc10$. This is a contradiction because the effect of the action sets $pc_{proc(\pi)} = pc5$. □

- $\pi = lock\_6F$, this action enters the next competition level and adds the current executor $proc(\pi)$ to $W_{i_{proc(\pi)}}$. There are two cases:

  Case 1. Suppose that $j \neq s_0.i_{proc(\pi)}$, then W_j is untouched, then the property holds by inductive hypothesis and is preserved. □

  Case 2. Conversely, suppose that $j = s_0.i_{proc(\pi)}$. There are two cases:

  Case 2.1. Suppose that $proc(\pi) \in s_0.W_j$, then its addition does not change the cardinality of W_j. We assumed that $|s_1.W_j| = 2 + (J+1)$ and by these assumptions $|s_0.W_j| = 2 + (J+1)$. Consequently, the inductive hypothesis holds is preserved and the conclusion comes. □

  Case 2.2. Conversely, suppose that $proc(\pi) \notin s_0.W_j$. By specular reasoning we deduce that $|s_0.W_j| = 2 + J$ and that $proc(\pi)$ was added to W_j. There are two cases:

  Case 2.2.1. Suppose that $s_0.victim[j] \neq proc(\pi)$. By inductive hypothesis on reachable state we known that $s_0.victim[j] \notin W_j$, then the conclusion comes. □

  Case 2.2.2. Conversely, suppose that $s_0.victim[j] = proc(\pi)$. We necessarily must reach a contradiction. Recall that we are assuming that $proc(\pi) \notin s_0.W_j$ and $proc(\pi) \in s_1.W_j$. Observe that the cardinality of $s_1.W_j$ is greater equal two, therefore, there exist at least two distinct elements belonging to it, let them be x and y. By invariant 10F (page 55) we know that x, y and $proc(\pi)$ are winners. By invariant 2F (page 46) applied to x and $proc(\pi)$ we conclude that $s_1.victim[j] \neq proc(\pi)$ and since this action does not rewrite *victim* we deduce $s_0.victim[j] \neq proc(\pi)$. This is the contradiction. □

- $\pi = lock\_8F$, this case is analogous to case $\pi = lock\_6F$. □

- $\pi = lock\_9F$, this case is analogous to case $\pi = lock\_6F$. □

- $\pi = unlock\_1$, this action removes the current executor $proc(\pi)$ from $W_{0<j\leq i_{proc(\pi)}}$, leaving all other sets untouched. We assume that $|s_1.W_j| = 2 + J \Rightarrow s_1.victim[j] \notin s_1.W_j$, $j > 0$ and $|s_1.W_j| = 2 + (J + 1)$. We prove that $s_1.victim[j] \notin s_1.W_j$. There are two cases:

  Case 1. Suppose that $0 < j \leq i_{proc(\pi)}$. The generic executor removes itself from W_j. There are two cases:

  Case 1.1. Suppose that $proc(\pi)$ is not winner. By invariant 10F (page 55), $proc(\pi) \notin W_j$, consequently its removal does not modify the cardinality. By hypothesis $|s_1.W_j| = 2 + (J + 1)$ and by deduction $|s_0.W_j| = 2 + (J + 1)$. We employ the inductive hypothesis generated by the induction on reachable states to show that $s_0.victim[j] \notin s_0.W_j$, preserving the property. □

  Case 1.2. Conversely, suppose $proc(\pi)$ is a winner. By invariant 10F (page 55), $proc(\pi) \in W_j$, consequently its removal does modify the cardinality. By hypothesis $|s_1.W_j| = 2+(J+1)$ and by deduction $|s_0.W_j| = 2+(J+2)$. We employ the inductive hypothesis generated by the induction on reachable states to show that $s_0.victim[j] \notin s_0.W_j$, preserving the property. □

  Case 2. Conversely, suppose that $j > i_{proc(\pi)}$, then set W_j is untouched, and $s_0.W_j = s_1.W_j$. Consequently, W_j's cardinality is untouched as well and by hypothesis $|W_j| = 2 + (J + 1)$. We employ the inductive hypothesis generated by induction over reachable states: by its use we prove that $s_0.victim[j] \notin W_j$ and since the set is unchanged, $s_1.victim[j] \notin W_j$. □

Invariant 5F (Filtered Cardinality)**.** In all reachable states, for any competition level j below $|\mathbb{P}|$, the number of winners on level j is $|\mathbb{P}| - j$ at most.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow \forall j \in \mathbb{N} : 0 \leq j \leq |\mathbb{P}| - 1 \Rightarrow |s_0.W_j| \leq |\mathbb{P}| - j$$

Proof. By induction on reachable states of the automaton.
Basis. Let s_0 be an initial state of the automaton.

$$s_0 \in I(FL) \vdash \forall j \in \mathbb{N} : 0 \leq j \leq |\mathbb{P}| - 1 \Rightarrow |s_0.W_j| \leq |\mathbb{P}| - j$$

Let j be a generic process. Observe that in initial states all processes have $W_0 = \top$ and $W_{j>0} = \emptyset$. There are two cases:
Case 1. Suppose $j = 0$. By lemma 6F (page 67) we know that the cardinality of the full set is $|\mathbb{P}|$, this proves the conclusion. □
Case 2. Conversely, suppose that $j > 0$. By definition, the cardinality of the empty set is 0, this proves the conclusion. □
Inductive step. Let s_0 and s_1 be states of the automaton, let s_0 be reachable; we assume the invariant holds in state s_0 by inductive hypothesis.

$$\forall j \in \mathbb{N} : 0 \leq j \leq |\mathbb{P}| - 1 \Rightarrow |s_0.W_j| \leq |\mathbb{P}| - j, reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash$$
$$\forall j \in \mathbb{N} : 0 \leq j \leq |\mathbb{P}| - 1 \Rightarrow |s_1.W_j| \leq |\mathbb{P}| - j$$

Let j be a generic natural. Observe that all actions not involved in the modification of W_j trivially preserve the property. Hence, we focus on:

$$\pi \in \{lock\_6F, lock\_8F, lock\_9F, unlock\_1\}$$

- $\pi = lock\_6F$, this action adds the executor to $W_{i_{proc(\pi)}}$. We induct on j.

  Basis. $0 \leq |\mathbb{P}| - 1$, then the cardinality of set $W_0 \leq |\mathbb{P}|$. By inductive hypothesis we have that $|W_0| \leq |\mathbb{P}|$, but this action modifies $W_{i_{proc(\pi)}}$. By invariant 14F (page 58) we know that $i_p > 0$, hence, the action does not modify set W_0 preserving the property. □

  Inductive step. The proof context is:

$$\forall j \in \mathbb{N} : j \leq |\mathbb{P}| - 1 \Rightarrow |s_0.W_j| \leq |\mathbb{P}| - j, \pi = lock\_6F, reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash$$
$$\forall J \in \mathbb{N} : (J \leq |\mathbb{P}| - 1 \Rightarrow |s_1.W_J| \leq |\mathbb{P}| - J) \Rightarrow$$
$$(J + 1 \leq |\mathbb{P}| - 1 \Rightarrow |s_1.W_{J+1}| \leq |\mathbb{P}| - (J + 1))$$

Assume by inductive hypothesis that $J \leq |\mathbb{P}| - 1 \Rightarrow |s_0.W_J| \leq |\mathbb{P}| - J$ and $J + 1 \leq |\mathbb{P}| - 1$. We must prove that in the post-state the cardinality of W_{J+1} is lesser or equal $|\mathbb{P}| - (J + 1)$. Let J be a generic natural. We prove the conclusion by contradiction, there are two cases:

Case 1. Suppose that $|s_1.W_{J+1}| > |\mathbb{P}| - (J + 1)$, then $|s_1.W_{J+1}| \geq |\mathbb{P}| - J$. By invariant 13F (page 57) we know that W_{J+1} is a subset of W_J, consequently $|s_1.W_{J+1}| \leq |s_1.W_J|$. By inductive hypothesis we know that $|s_1.W_J| \leq |\mathbb{P}| - J$ and by transitivity of $\leq (\cdot, \cdot)$ we get $|s_1.W_{J+1}| \leq |\mathbb{P}| - J$; by joining this observation with the initial supposition of the current case we infer that $|s_1.W_{J+1}| = |\mathbb{P}| - J$. By the observation that W_{J+1} and W_J are one the subset of the other we deduce that $|s_1.W_J| = |\mathbb{P}| - J$. By the assumption that W_{J+1} and W_J share the same cardinality and elements, we deduce they are the same set. By the premise of the conclusion consequent we have that $|\mathbb{P}| - J \geq 2$, consequently $|s_1.W_{J+1}| \geq 2$. This implies there exist two distinct winner processes on level $J + 1$, let them be x and y. Remember that being a winner on level j implies being a competitor on the same level. By invariant 3F (page 46) and by the existence of x we know that the victim process is a competitor on level $J + 1$. By invariant 12F (page 56) we know that the victim process of level $J + 1$ is a winner on level J, consequently by invariant 10F (page 55) it belongs to W_J. By invariant 4F (page 48) we know that the victim process on level $J + 1$ is not a winner on level $J + 1$, consequently it does not belong to W_{J+1}. We previously deduced that $W_J = W_{J+1}$, hence, the belonging of the victim to the winner set is contradictory. □

Case 2. Conversely, suppose that $|s_1.W_{J+1}| \leq |\mathbb{P}| - (J + 1)$. That is what we wanted to prove. □

- $\pi = lock\_8F$, this case is analogous to case $\pi = lock\_6F$. □
- $\pi = lock\_9F$, this case is analogous to case $\pi = lock\_6F$. □
- $\pi = unlock\_1$, this case is analogous to case $\pi = lock\_6F$. □

Invariant 6F (Unique Loop-breaker)**.** In all reachable states, processes in CS or that break the outer while-loop are winners of the level $|\mathbb{P}| - 1$.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow$$
$$\forall p \in \mathbb{P} : \big(s_0.pc_p = pc11 \vee (s_0.pc_p = pc2 \wedge s_0.i_p = |\mathbb{P}|)\big) \Rightarrow p \in s_0.W_{|\mathbb{P}|-1}$$

Proof. By induction on reachable states of the automaton.
Basis. Let s_0 be an initial state of the automaton.

$$s_0 \in I(FL) \vdash \forall p \in \mathbb{P} : \big(s_0.pc_p = pc11 \vee (s_0.pc_p = pc2 \wedge s_0.i_p = |\mathbb{P}|)\big) \Rightarrow p \in s_0.W_{|\mathbb{P}|-1}$$

Let p be a generic process in state s_0. Observe that in initial states all processes have their PC values set to $pcIdle$, satisfying the consequent vacuously. □
Inductive step. Let s_0 and s_1 be states of the automaton, let s_0 be reachable; we assume the invariant holds in state s_0 by inductive hypothesis.

$$\forall p \in \mathbb{P} : \big(s_0.pc_p = pc11 \vee (s_0.pc_p = pc2 \wedge s_0.i_p = |\mathbb{P}|)\big) \Rightarrow p \in s_0.W_{|\mathbb{P}|-1}, reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash$$
$$\forall p \in \mathbb{P} : \big(s_1.pc_p = pc11 \vee (s_1.pc_p = pc2 \wedge s_1.i_p = |\mathbb{P}|)\big) \Rightarrow p \in s_1.W_{|\mathbb{P}|-1}$$

Let p be a generic process in state s_0. There are two cases:
Case 1. Suppose that p is the generic executor: $proc(\pi) = p$. Observe that all actions not having $pc_p \in \{pc2, pc11\}$ as destination trivially invalidate the premise of the consequent. Hence, we concentrate on:

$$\pi \in \{lock\_1, lock\_2F, lock\_10\}$$

- $\pi = lock\_1$, this action sets $i_p = 1$. We assume that in the post-state $pc_p = pc11$ or $pc_p = pc2$ and $i_p = |\mathbb{P}|$. Observe that this action invalidates both assumptions, making the consequent hold vacuously. □

- $\pi = lock\_2F$, this action enters CS. Observe the this action can execute only when $s_0.i_p = |\mathbb{P}|$ and $s_0.pc_p = pc2$, this satisfies the inductive hypothesis that is preserved. □
- $\pi = lock\_10$, this action increments i_p. We assume that in the post-state $pc_p = pc11$ or $pc_p = pc2$ and $i_p = |\mathbb{P}|$. The first case is absurd and is dismissed in a separate branch. The other leads to conclude that in the pre-state $i_p = |\mathbb{P}| - 1$ invalidating the inductive hypothesis. By inspection of variables pc and i we observe that p is $winner$ in level $|\mathbb{P}| - 1$. By invariant 10F (page 55) we deduce then that $p \in W_{|\mathbb{P}|-1}$. Proving the conclusion. □

Case 2. Conversely, suppose that p is not the generic executor, $proc(\pi) \neq p$, then it is suspended, its local and private variables do not change, and the property is preserved trivially. □

Invariant 7F (Mutual Exclusion)**.** In all reachable states, Filter-Lock satisfies mutual exclusion.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow \forall p, q \in \mathbb{P} : \neg(s_0.pc_p = pc11 \wedge s_0.pc_q = pc11 \wedge p \neq q)$$

Proof. By direct proof.

$$reach(FL, s_0) \vdash \forall p, q \in \mathbb{P} : \neg(s_0.pc_p = pc11 \wedge s_0.pc_q = pc11 \wedge p \neq q)$$

Let p and q be generic processes in state s_0. By invariant 6F (page 51), applied twice, we know that $p \in W_{|\mathbb{P}|-1}$ and $q \in W_{|\mathbb{P}|-1}$. By invariant 5F (page 50) we know that $|s_0.W_{|\mathbb{P}|-1}| \leq 1$. There are two cases:
Case 1. Suppose $|W_{|\mathbb{P}|-1}| = 0$, this hypothesis contradicts the observation that p and q are in $W_{|\mathbb{P}|-1}$. □
Case 2. Conversely, suppose $|W_{|\mathbb{P}|-1}| = 1$. By lemma 4F (page 66) we know that the set whose cardinality is 1 contains a single element. Therefore, it is not true that p and q are in CS and are different. □

As stated Filter-Lock satisfies mutual exclusion. The formalization did not only prove the property but also provided a detailed explanation of the motivations that force ME to be guaranteed.

4.2.2 Middle-Level Invariants

In this subsection we state and prove middle-level invariants.

Invariant 8F (Processes in Winner Sets are Winners)**.** In all reachable states, processes belonging to the winner set on level j are winners on level j.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow \forall p \in \mathbb{P}, j \in \mathbb{N} : p \in s_0.W_j \Rightarrow winner(s_0, p, j)$$

For convenience we recall the definition of winner:

$$winner(s_0, p, j) = s_0.i_p > j \vee (s_0.i_p = j \wedge s_0.pc_p = pc10)$$

Proof. By induction on reachable states of the automaton.
Basis. Let s_0 be an initial state of the automaton.

$$s_0 \in I(FL) \vdash \forall p \in \mathbb{P}, j \in \mathbb{N} : p \in s_0.W_j \Rightarrow winner(s_0, p, j)$$

Let p be a generic process in s_0 and j be a generic natural. We prove the left clause of the definition is satisfied and conclude the basis; observe that by definition of initial states, two cases are possible:
Case 1. When $j = 0$, the premise of the consequent is satisfied and so is the definition of $winner$: process p has $i = 1$ and $i > j$. □
Case 2. When $j > 0$, then $W_j = \emptyset$: this vacuously satisfies the consequent. □

Inductive step. Let s_0 and s_1 be states of the automaton, let s_0 be reachable; we assume the invariant holds in state s_0 by inductive hypothesis.

$$\forall p \in \mathbb{P}, j \in \mathbb{N} : p \in s_0.W_j \Rightarrow winner(s_0, p, j), reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash$$
$$\forall p \in \mathbb{P}, j \in \mathbb{N} : p \in s_1.W_j \Rightarrow winner(s_1, p, j)$$

Let p be a generic process in state s_0, let j be a generic natural. We assume the premise of the consequent to be true, hence, $p \in s_1.W_j$. There are two cases:
Case 1. Suppose p is the generic executor: $proc(\pi) = p$. Observe that all actions that do not modify W_j or i, trivially preserve the property. Consequently, we focus on these:

$$\pi \in \{\ lock\_1, lock\_6F, lock\_8F, lock\_9F, lock\_10, unlock\_1\ \}$$

- $\pi = lock\_1$, this actions sets $i_p = 1$. Observe that this action does not modify set W_j, consequently $p \in s_0.W_j$. By this observation we know the inductive hypothesis is satisfied, that is, p is a *winner* in the pre-state. By invariant 16F (page 59) we know that in the pre-state $i_p = 1$, consequently if p is a winner in the pre-state then j must be 0. Therefore, the definition of winner is satisfied in the post-state as well. □

- $\pi = lock\_6F$, this action adds p's ID to W_{i_p} while leaving i_p untouched. There are three cases:

  Case 1.1. Suppose $s_1.i_p = j$. By the transition $s_1.pc_p = pc10$ and $s_1.i_p = s_0.i_p$. Consequently p is a winner in the post-state because the right clause of the definition is satisfied and with it the consequent. □

  Case 1.2. Conversely, suppose that $s_1.i_p \neq j$. The transition leaves i_p unchanged, consequently W_j is unchanged as well. Therefore, $p \in s_0.W_j$ and $p \in s_1.W_j$. There are two cases:

  - **Case 1.2.1.** Suppose $s_0.i_p > j$, then p is a winner and the property is preserved by the transition. □
  - **Case 1.2.2.** Suppose $s_0.i_p < j$, the inductive hypothesis is invalidated because $p \in s_0.W_j$ but p is not a winner. That is a contradiction. □

- $\pi = lock\_8F$, this case is analogous to case $\pi = lock\_6F$. □

- $\pi = lock\_9F$, this case is analogous to case $\pi = lock\_6F$. □

- $\pi = lock\_10$, this action increments i_p. Observe that this actions does not modify W_j for any j and recall we assumed the premise of the consequent to be true: $p \in s_1.W_j$. Consequently, it must be true that $p \in s_0.W_j$, hence, the inductive hypothesis holds and by means of it we know that p is winner in the pre-state. There are two cases:

  Case 1.1. Suppose p is winner because $s_0.i_p > j$, by the transition $s_1.i_p > j$, because the transition increments i. Enforcing the conclusion. □

  Case 1.2. Suppose p is winner because $s_0.i_p = j \wedge s_0.pc_p = pc10$, by the transition $s_1.i_p > j$, satisfying the definition of *winner* in the post-state. □

- $\pi = unlock\_1$, this action removes p from all winner sets W_j for $0 < j \leq i_p$. For $j > 0$ the premise of the consequent is invalidated, making it hold vacuously. Conversely, for $j = 0$, the inductive hypothesis holds because the W_0 is never abandoned, hence, the property holds in the pre-state and is preserved. □

Case 2. Conversely, suppose the generic executor is not p, $proc(\pi) \neq p$, then the transition function can not modify i_p or W_j for any j. Consequently, the invariant is preserved by all actions. □

Invariant 9F (Winners are Processes in Winner Sets)**.** In all reachable states, winner processes on level j belong to the winner set on level j.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow \forall p \in \mathbb{P}, j \in \mathbb{N} : winner(s_0, p, j) \Rightarrow s_0.W_j$$

Proof. By induction on reachable states of the automaton.
Basis. Let s_0 be an initial state of the automaton.

$$s_0 \in I(FL) \vdash \forall p \in \mathbb{P}, j \in \mathbb{N} : winner(s_0, p, j) \Rightarrow s_0.W_j$$

Let p be a generic process in state s_0, let j be a generic natural. Observe that by definition of initial state all processes start with $pc = pcIdle$, $i = 1$, $W_0 = \top$ and $W_{j>0} = \emptyset$. There are two cases:
Case 1. p is winner because $i_p > j$, then $j = 0$ and by construction $p \in W_0$. □
Case 2. p is winner because $i_p = j$ and $pc_p = pc10$, then we reach a contradiction and the consequent is vacuously satisfied. □
Inductive step. Let s_0 and s_1 be states of the automaton, let s_0 be reachable; we assume the invariant holds in state s_0 by inductive hypothesis.

$$\begin{array}{l} \forall p \in \mathbb{P}, j \in \mathbb{N} : winner(s_0, p, j) \Rightarrow s_0.W_j, reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash \\ \forall p \in \mathbb{P}, j \in \mathbb{N} : winner(s_1, p, j) \Rightarrow s_1.W_j \end{array} \tag{4.1}$$

Let p be a generic process in state s_0, let j be a generic natural. We assume the premise of the consequent to be true, hence, $winner(s_1, p, j)$. There are two cases:
Case 1. Suppose p is the generic executor: $proc(\pi) = p$. Observe that all actions that do not modify W_j or variable i trivially preserve the property. Consequently, we focus on these:

$$\pi \in \{\ lock\_1, lock\_6F, lock\_8F, lock\_9F, lock\_10, unlock\_1\ \}$$

- $\pi = lock\_1$, this actions sets $i_p = 1$. We assumed p to be a winner in the post-state, hence, it is because $s_1.i_p > j$ (the right clause is unsatisfiable during this action), consequently $j = 0$. This deduction satisfied the definition of winner in the pre-state, that in turn satisfies the inductive hypothesis. The inductive hypothesis is preserved by the transition and the conclusion comes. □

- $\pi = lock\_6F$, this action adds p to W_{i_p}. Observe that this action does not modify i_p, we expand the definition of $winner(s_1, p, j) = \big(s_1.i_p > j \vee (s_1.i_p = j \wedge s_1.pc_p = pc10)\big)$ and split the proof into two cases:

  Case 1. $s_1.i_p > j$. By the transition $s_0.i_p > j$, hence, the inductive hypothesis is satisfied and preserved. □

  Case 2. $s_1.i_p = j \wedge s_1.pc_p = pc10$. The inductive hypothesis is of no use here, however $j = i_p$, hence, the transition adds p to W_j, satisfying the consequent. □

- $\pi = lock\_8F$, this case is analogous to case $\pi = lock\_6F$. □

- $\pi = lock\_9F$, this case is analogous to case $\pi = lock\_6F$. □

- $\pi = lock\_10$, this action increments i_p. We expand the definition of *winner* and split the proof into two cases:

  Case 1. $s_1.i_p > j$. The increment implies the existence of two subcases:

  Case 1.1. $s_1.i_p > j$ and $s_0.i_p > j$, then the inductive hypothesis is satisfied and preserved. □

  Case 1.2. $s_1.i_p > j$ and $s_0.i_p = j$, then the definition of winner is satisfied in the pre-state; the inductive hypothesis is in turn satisfied and preserved because $s_0.pc_p = pc10$. □

  Case 2. $s_1.i_p = j \wedge s_1.pc_p = pc10$. The transition sets $s_1.pc_p = pc2$, invalidating the assumption that p is a winner. □

- $\pi = unlock\_1$, this action sets $i_p = 1$ and removes p from all sets $W_{j>0}$. We expand the definition of *winner* and split the proof into two cases:

  Case 1. $s_1.i_p > j$, consequently $j = 0$. By invariant 15F we know that in the pre-state $i_p = |\mathbb{P}|$. This satisfies the definition of $winner(s_0, p, 0)$, that in turn satisfies the inductive hypothesis that is preserved. □

  Case 2. $s_1.i_p = j \wedge s_1.pc_p = pc10$, observe this action sets $s_1.pc_p = pc12$, invalidating the assumption that p is a winner. □

Case 2. Conversely, suppose the generic executor is not p, $proc(\pi) \neq p$, then the transition function can not modify i_p or W_j for any j. Consequently, the invariant is preserved by all actions. □

Invariant 10F (Winners are Winners)**.** In all reachable states, winner processes belong to the winner set and vice versa.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow \forall p \in \mathbb{P}, j \in \mathbb{N} : winner(s_0, p, j) \Leftrightarrow p \in s_0.W_j$$

Proof. By direct proof.

$$reach(FL, s_0) \vdash \forall p \in \mathbb{P}, j \in \mathbb{N} : winner(s_0, p, j) \Leftrightarrow p \in s_0.W_j$$

Let p be a generic process in state s_0, let j be a generic natural; applying invariants 8F and 9F (pages 52 and 53) we prove the conclusion. □

Invariant 10F is useful for rewriting the definition of winner processes in set theoretic terms.

Invariant 11F (Cumulative Levels)**.** In all reachable states, any process that wins competition $j+1$ also wins competition j.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow \forall p \in \mathbb{P}, j \in \mathbb{N} : winner(s_0, p, j+1) \Rightarrow winner(s_0, p, j)$$

Proof. By induction on reachable states of the automaton.
Basis. Let s_0 be an initial state of the automaton.

$$s_0 \in I(FL) \vdash \forall p \in \mathbb{P}, j \in \mathbb{N} : winner(s_0, p, j+1) \Rightarrow winner(s_0, p, j)$$

Let p be a generic process in the initial state s_0, let j be a generic natural. We assume the premise of the consequent to be true, hence, $winner(s_0, p, j+1)$. We expand the definition of winner and split the proof into two cases:
Case 1. $i_p > j+1$, observe that by definition of initial states $i_p = 1$. This invalidates the assumption that p is a winner. □
Case 2. $i_p = j+1 \wedge pc_p = pc10$, observe that by definition all processes in initial state have $pc = pcIdle$, invalidating the assumption. □
Inductive step. Let s_0 and s_1 be states of the automaton, let s_0 be reachable; we assume the invariant holds in state s_0 by inductive hypothesis.

$$\forall p \in \mathbb{P}, j \in \mathbb{N} : winner(s_0, p, j+1) \Rightarrow winner(s_0, p, j), reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash$$
$$\forall p \in \mathbb{P}, j \in \mathbb{N} : winner(s_1, p, j+1) \Rightarrow winner(s_1, p, j)$$

Let p be a generic process in state s_0.
Case 1. Suppose p is the generic executor: $proc(\pi) = p$. Observe that all actions that do not modify i trivially preserve the property. Consequently, we focus on these actions:

$$\pi \in \{lock\_1, lock\_10, unlock\_1\}$$

We assume the premise of the consequent to be true, hence, $winner(s_1, p, j+1)$.

- $\pi = lock\_1$, this action sets $i_p = 1$. Setting variable i_p invalidates the assumption that p is a winner because: $(s_1.i_p \not> 1 + j)$ and $(s_1.i_p = 1 + j \wedge s_1.pc_p \neq pc10)$. □
- $\pi = lock\_10$, this action increments i_p. We expand the definition of $winner(p, j+1)$ and split the proof into two cases:

  Case 1.1. $s_1.i_p > j+1$, this observation trivially satisfies the conclusion that $winner(s_1.p, j)$, because $s_0.i_p > j$. □

  Case 1.2. $s_1.i_p = j+1 \wedge s_1.pc_p = pc10$, the transition sets $s_1.pc_p = pc2$ invalidating the assumption that p is a winner. □

- $\pi = unlock\_1$, this case is analogous to case $\pi = lock\_1$. □

Case 2. Conversely, suppose the generic executor is not p, $proc(\pi) \neq p$, then the transition function can not modify i_p or W_j for any j. Consequently, the invariant is preserved by all actions. □

Invariant 12F (Competitors are Lower Levels' Winners)**.** In all reachable states, competitors on level $j+1$ are winners on level j.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow \forall p \in \mathbb{P}, j \in \mathbb{N} : comp(s_0, p, j+1) \Rightarrow winner(s_0, p, j)$$

For convenience we recall the definition of competitor:

$$comp(s_0, p, j) = winner(s_0, p, j) \vee (s_0.i_p = j \wedge s_0.pc_p \in \{pc5, pc6, pc7, pc8, pc9\})$$
$$\Leftrightarrow$$
$$comp(s_0, p, j) = s_0.i_p > j \vee (s_0.i_p = j \wedge s_0.pc_p = pc10) \vee (s_0.i_p = j \wedge s_0.pc_p \in \{pc5, pc6, pc7, pc8, pc9\})$$

Proof. By induction on reachable states of the automaton.
Basis. Let s_0 be an initial state of the automaton.

$$s_0 \in I(FL) \vdash \forall p \in \mathbb{P}, j \in \mathbb{N} : comp(s_0, p, j+1) \Rightarrow winner(s_0, p, j)$$

Let p be a generic process in the initial state and j a generic natural. We expand the definition of $comp(s_0, p, j+1)$ and split the proof into two cases:
Case 1. $winner(s_0, p, j+1)$. By invariant 11F (page 55) we know that $winner(s_0, p, j+1)$ implies $winner(s_0, p, j)$. That is what we wanted to prove. □
Case 2. $s_0.i_p = j+1 \wedge s_0.pc_p \in \{pc5, pc6, pc7, pc8, pc9\}$. Observe that in initial states no process has its PC value set to any value involved by the predicate. This contradicts the assumption that p is winner at level $j+1$. Consequently, the consequent holds vacuously. □
Inductive step. Let s_0 and s_1 be states of the automaton, let s_0 be reachable; we assume the invariant holds in state s_0 by inductive hypothesis.

$$s_0 \in I(FL) \vdash \forall p \in \mathbb{P}, j \in \mathbb{N} : comp(s_0, p, j+1) \Rightarrow winner(s_0, p, j), reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash$$
$$s_0 \in I(FL) \vdash \forall p \in \mathbb{P}, j \in \mathbb{N} : comp(s_1, p, j+1) \Rightarrow winner(s_1, p, j)$$

Let p be a generic process in state s_0. There are two cases:
Case 1. Suppose p is the generic executor: $proc(\pi) = p$. We assume the premise of the consequent to be true, hence, $comp(s_1, p, j+1)$. Expand the definition of competitor and split the proof into two cases:
Case 1.1. $winner(s_1, p, j+1)$. By invariant 11F (page 55) we deduce $winner(s_1, p, j)$, this trivially proves the conclusion. □
Case 1.2. $s_1.i_p = j+1 \wedge s_1.pc_p \in \{pc5, pc6, pc7, pc8, pc9\}$. Observe that the following actions invalidate the assumption that p is a competitor, due to a PC based conflict.

$$\pi \in \left\{ \begin{array}{l} lock\_invoke, lock\_1, lock\_2T, lock\_2F, lock\_3, \\ lock\_6F, lock\_8F, lock\_9F, lock\_10, unlock\_1, lock\_response \end{array} \right\}$$

Observe that remaining actions do not modify i_p consequently: $s_0.i_p = j+1$.

$$\pi \in \{\ lock\_4, lock\_5, lock\_6T, lock\_7T, lock\_7F, lock\_8T, lock\_9T\ \}$$

Then $s_0.i_p > j$, this qualify p as a winner in the pre-state on level j. All transitions preserve the status of winner to the post-state, proving the property. □
Case 2. Conversely, suppose the generic executor is not p, $proc(\pi) \neq p$, then the transition function can not modify i_p or pc_p. Consequently, the invariant is preserved by all actions. □

Invariant 13F (Winner Sets form a Hierarchy)**.** In all reachable states, for any j, the set of winners on level $j+1$ is an improper subset of the set of winners on level j.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow \forall j \in \mathbb{N} : s_0.W_{j+1} \subseteq s_0.W_j$$

We recall the definition of improper subset:

$$\forall (A, B : finiteset) : A \subseteq B \Leftrightarrow \forall x \in A \Rightarrow x \in B$$

Proof. By induction on reachable states of the automaton.
Basis. Let s_0 be an initial state of the automaton.

$$s_0 \in I(FL) \vdash \forall j \in \mathbb{N} : s_0.W_{j+1} \subseteq s_0.W_j$$

Let j be a generic natural. Observe that in all initial states set $W_0 = \top$ and $W_{j>0} = \emptyset$, consequently there are two cases:
Case 1. Suppose that $j = 0$. Then $W_1 \subseteq W_0$, because W_1 is empty. □
Case 2. Conversely, suppose that $j > 0$. Then $W_{j+1} \subseteq W_j$, because $\emptyset \subseteq \emptyset$. □
Inductive step. Let s_0 and s_1 be states of the automaton, let s_0 be reachable; we assume the invariant holds in state s_0 by inductive hypothesis.

$$\forall j \in \mathbb{N} : s_0.W_{j+1} \subseteq s_0.W_j, reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash$$
$$\forall j \in \mathbb{N} : s_1.W_{j+1} \subseteq s_1.W_j$$

Let x be the generic element of W_{j+1}, we assume by inductive hypothesis that $x \in W_{j+1}$. Observe that only the following actions modify W_j, the others trivially preserve the property:

$$\pi \in \{lock\_6F, lock\_8F, lock\_9F, unlock\_1\}$$

Hence, we focus on them.

- $\pi = lock\_6F$, this actions adds the executor to $W_{i_{proc(\pi)}}$. Observe that by inductive hypothesis and by invariant 10F (page 55), x is defined winner on level $j+1$. By invariant 11F (page 55) we know that x is defined winner on level j and by invariant 10F (page 55) we know that x is a member of W_j. This proves the conclusion. □
- $\pi = lock\_8F$, this case is analogous to case $act = lock\_6F$. □
- $\pi = lock\_9F$, this case is analogous to case $act = lock\_6F$. □
- $\pi = unlock\_1$, this case is analogous to case $act = lock\_6F$. □

Lemma 1F (Suspended Winner)**.** In all reachable states, suspended winners in the post-state are winners in the pre-state.

$$\forall s_0, s_1 \in Q(FL), \pi \in \Sigma(FL), p \in \mathbb{P}, j \in \mathbb{N} :$$
$$\big(reach(FL, s_0) \wedge s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \wedge proc(\pi) \neq p \wedge winner(s_1, p, j)\big)$$
$$\Rightarrow winner(s_0, p, j)$$

Proof. By construction. □

Lemma 2F (Suspended Competitor)**.** In all reachable states, suspended competitors in the post-state are competitors in the pre-state.

$$\forall s_0, s_1 \in Q(FL), \pi \in \Sigma(FL), p \in \mathbb{P}, j \in \mathbb{N} :$$
$$\big(reach(FL, s_0) \wedge s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \wedge proc(\pi) \neq p \wedge comp(s_1, p, j)\big)$$
$$\Rightarrow comp(s_0, p, j)$$

Proof. By construction. □

We could avoid the introduction of lemmata 1F and 2F; in the mechanical setting they serve the purpose to quickly and efficiently discharge those cases where the property of being a winner or competitor is required to be preserved backward, from the post-state to the pre-state.

4.2.3 Low-Level Invariants

In this section we state and prove the invariants requires to specify the behaviour of variables during the execution of the algorithm.

Lemma 3F (Suspended Process)**.** In all reachable states, suspended processes are not subject to program counters updates.

$$\begin{gathered}\forall s_0, s_1 \in Q(FL), \pi \in \Sigma(FL), p \in \mathbb{P} : \\ \big(reach(FL, s_0) \wedge s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \wedge proc(\pi) \neq p\big) \\ \Rightarrow s_0.pc_p = s_1.pc_p\end{gathered}$$

Proof. By definition. □

Whenever we need to analyse a case where some process is not executing we implicitly refer to this lemma. This case happens in most proofs by induction on reachable states, hence, pointing out the application every time would result into useless wordiness.

Invariant 14F (i is Positive)**.** In all reachable states, for all processes, variable i is positive.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow \forall p \in \mathbb{P} : s_0.i_p > 0$$

Proof. By induction on reachable states of the automaton.
Basis. Let s_0 be an initial state of the automaton.

$$s_0 \in I(FL) \vdash \forall p \in \mathbb{P} : s_0.i_p > 0$$

Let p be the generic process in state s_0, by definition of initial states $i_p = 1$, the conclusion comes. □
Inductive step. Let s_0 and s_1 be states of the automaton, let s_0 be reachable; we assume the invariant holds in state s_0 by inductive hypothesis.

$$\begin{gathered}\forall p \in \mathbb{P} : s_0.i_p > 0, reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash \\ \forall p \in \mathbb{P} : s_1.i_p > 0\end{gathered}$$

Let p be the generic process in state s_0. We assumed by inductive hypothesis that $s_0.i_p > 0$. There are two cases:
Case 1. Suppose the generic executor is p: $proc(\pi) = p$. We must analyse all actions of the automaton. Observe the only actions modifying i are:

$$\pi \in \{lock\_1, lock\_10, unlock\_1\}$$

The others leave the variable untouched preserving the property, consequently we focus on these them:

- $\pi = lock\_1$, this action sets $i_p = 1$, enforcing the conclusion. □
- $\pi = lock\_10$, this action sets $i_p = i_p + 1$, enforcing the conclusion. □
- $\pi = unlock\_1$, this action sets $i_p = 1$, enforcing the conclusion. □

Case 2. Conversely, suppose the generic executor is not p, $proc(\pi) \neq p$, then the transition function can not modify p's private variable i. Consequently, the invariant is preserved by all actions. □

Invariant 15F (i in CS equals $|\mathbb{P}|$)**.** In all reachable states, processes in CS have variable i equal to $|\mathbb{P}|$.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow \forall p \in \mathbb{P} : s_0.pc_p = pc11 \Rightarrow s_0.i_p = |\mathbb{P}|$$

Proof. By induction on reachable states of the automaton.
Basis. Let s_0 be an initial state of the automaton.

$$s_0 \in I(FL) \vdash \forall p \in \mathbb{P} : s_0.pc_p = pc11 \Rightarrow s_0.i_p = |\mathbb{P}|$$

Let p be the generic process in the initial state s_0. Observe that in initial states no process is in CS, making the consequent vacuously satisfied. □
Inductive step. Let s_0 and s_1 be states of the automaton, let s_0 be reachable; we assume the invariant holds in state s_0 by inductive hypothesis.

$$\begin{array}{l}\forall p \in \mathbb{P} : s_0.pc_p = pc11 \Rightarrow s_0.i_p = |\mathbb{P}|, reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash \\ \forall p \in \mathbb{P} : s_1.pc_p = pc11 \Rightarrow s_1.i_p = |\mathbb{P}|\end{array}$$

Let p be the generic process in state s_0. There are two cases:
Case 1. Suppose the generic executor is p: $proc(\pi) = p$. We must analyse all actions of the automaton. Observe that all actions but the following trivially invalidate the premise of the consequent formula, making it hold vacuously.

$$\pi \in \left\{ \ \pi = lock\_2F \ \right\}$$

Consequently, we focus on $\pi = lock\_2F$.

- $\pi = lock\_2F$, this action enters CS. Observe that the transition is executed only when $s_0.i_p = |\mathbb{P}|$ and it does not modify i_p, therefore, the conclusion comes. □

Case 2. Conversely, suppose the generic executor is not p, $proc(\pi) \neq p$, then the transition function can not modify p's program counter value. Consequently, the invariant is preserved by all actions. □

Invariant 16F (i is 1)**.** In all reachable states, for suitable PC values, local variable $i = 1$.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow \forall p \in \mathbb{P} : s_0.pc_p \in \{pcIdle, pc1, pc12\} \Rightarrow s_0.i_p = 1$$

Proof. By induction on reachable states of the automaton.
Basis. Let s_0 be an initial state of the automaton.

$$s_0 \in I(FL) \vdash \forall p \in \mathbb{P} : s_0.pc_p \in \{pcIdle, pc1, pc12\} \Rightarrow s_0.i_p = 1$$

Let p be the generic process in s_0. Observe that no process in initial states has pc set to $pc1$ or $pc12$ making the consequent vacuously satisfied. Also observe that those processes having pc set to $pcIdle$ also have $i = 1$, hence, the conclusion comes. □
Inductive step. Let s_0 and s_1 be states of the automaton, let s_0 be reachable; we assume the invariant holds in state s_0 by inductive hypothesis.

$$\begin{array}{l}\forall p \in \mathbb{P} : s_0.pc_p \in \{pcIdle, pc1, pc12\} \Rightarrow s_0.i_p = 1, reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash \\ \forall p \in \mathbb{P} : s_1.pc_p \in \{pcIdle, pc1, pc12\} \Rightarrow s_1.i_p = 1\end{array}$$

Let p be the generic process in state s_0. There are two cases:
Case 1. Suppose the generic executor is p: $proc(\pi) = p$. We must analyse all actions of the automaton. Observe that the following actions

$$\pi \in \left\{ \begin{array}{l} lock\_1, lock\_2T, lock\_2F, lock\_3, lock\_4, lock\_5, \\ lock\_6F, lock\_6T, lock\_7F, lock\_7T, \\ lock\_8T, lock\_8F, lock\_9T, lock\_9F, lock\_10 \end{array} \right\}$$

invalidate the premise of the consequent formula due to a PC based conflict, making it hold vacuously.

Consequently, we focus on actions:

$$\pi \in \{lock\_invoke, unlock\_1, lock\_response\}$$

- $\pi = lock\_invoke$, observe that the inductive hypothesis holds and the transition preserves it, because variable i_p is untouched. □
- $\pi = unlock\_1$, observe that the transition sets $i_p = 1$, enforcing the conclusion. □
- $\pi = lock\_response$, this case is analogous to case $\pi = lock\_invoke$. □

Case 2. Conversely, suppose the generic executor is not p, $proc(\pi) \neq p$, then the transition function can not modify p's private variables and program counter value. Consequently, the invariant is preserved by all actions. □

Invariant 17F (i is $level$)**.** In all reachable states, for suitable PC values, i and $level$ variables are equal.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow$$
$$\forall p \in \mathbb{P} : s_0.pc_p \in \{pc4, pc5, pc6, pc7, pc8, pc9, pc10\} \Rightarrow s_0.i_p = s_0.level_p$$

Proof. By induction on reachable states of the automaton.
Basis. Let s_0 be an initial state of the automaton.

$$s_0 \in I(FL) \vdash \forall p \in \mathbb{P} : s_0.pc_p \in \{pc4, pc5, pc6, pc7, pc8, pc9, pc10\} \Rightarrow s_0.i_p = s_0.level_p$$

Let p be the generic process in the initial states; observe that in initial states no process has pc set to any value admitted by the property, making it hold vacuously. □
Inductive step. Let s_0 and s_1 be states of the automaton, let s_0 be reachable; we assume the invariant holds in state s_0 by inductive hypothesis.

$$\forall p \in \mathbb{P} : s_0.pc_p \in \{pc4, pc5, pc6, pc7, pc8, pc9, pc10\} \Rightarrow s_0.i_p = s_0.level_p, reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash$$
$$\forall p \in \mathbb{P} : s_1.pc_p \in \{pc4, pc5, pc6, pc7, pc8, pc9, pc10\} \Rightarrow s_1.i_p = s_1.level_p$$

Let p be the generic process state s_0. There are two cases:
Case 1. Suppose the generic executor is p: $proc(\pi) = p$. We must analyse all actions of the automaton. Observe that the following actions invalidate the premise of the consequent formula, making it hold vacuously:

$$\pi \in \{\ lock\_invoke, lock\_1, lock\_2T, lock\_2F, lock\_10, unlock\_1, lock\_response\ \}$$

Consequently, we focus on:

$$\pi \in \{\ lock\_3, lock\_4, lock\_5, lock\_6T, lock\_6F, lock\_7F, lock\_7T, lock\_8T, lock\_8F, lock\_9T, lock\_9F\ \}$$

- $\pi = lock\_3$, this action sets $level_p = i_p$. Observe that the effect enforces the conclusion. □
- $\pi = lock\_4$, this action sets $victim[j] = p$. Observe that the inductive hypothesis is satisfied and preserved because the transition does not rewrite $level_p$ or i_p. □
- $\pi = lock\_5$, this case is analogous to case $\pi = lock\_4$. □
- $\pi = lock\_6T$, this case is analogous to case $\pi = lock\_4$. □
- $\pi = lock\_6F$, this case is analogous to case $\pi = lock\_4$. □
- $\pi = lock\_7T$, this case is analogous to case $\pi = lock\_4$. □
- $\pi = lock\_7F$, this case is analogous to case $\pi = lock\_4$. □
- $\pi = lock\_8T$, this case is analogous to case $\pi = lock\_4$. □
- $\pi = lock\_8F$, this case is analogous to case $\pi = lock\_4$. □
- $\pi = lock\_9T$, this case is analogous to case $\pi = lock\_4$. □
- $\pi = lock\_9F$, this case is analogous to case $\pi = lock\_4$. □

Case 2. Conversely, suppose the generic executor is not p, $proc(\pi) \neq p$, then the transition function can not modify p's $level$ variables and program counter value. Consequently, the invariant is preserved by all actions. □

Invariant 18F (Possible *level* Ranges)**.** In all reachable states, *level* and i variables can differ at most of one unit.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow \forall p \in \mathbb{P} : s_0.i_p = s_0.level_p \vee s_0.i_p - 1 = s_0.level_p$$

Proof. By induction on reachable states of the automaton.
Basis. Let s_0 be an initial state of the automaton.

$$s_0 \in I(FL) \vdash \forall p \in \mathbb{P} : s_0.i_p = s_0.level_p \vee s_0.i_p - 1 = s_0.level_p$$

Let p be the generic process in initial states; observe that in initial states all processes have $i = 1$ and $level = 0$, consequently the property is satisfied. □
Inductive step. Let s_0 and s_1 be states of the automaton, let s_0 be reachable; we assume the invariant holds in state s_0 by inductive hypothesis.

$$\forall p \in \mathbb{P} : (s_0.i_p = s_0.level_p \vee s_0.i_p - 1 = s_0.level_p), reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash$$
$$\forall p \in \mathbb{P} : (s_1.i_p = s_1.level_p \vee s_1.i_p - 1 = s_1.level_p)$$

Let p be the generic process in state s_0. There are two cases:
Case 1. Suppose the generic executor is p: $proc(\pi) = p$. Observe that the only actions modifying variable i_p are $\pi \in \{lock\_1, lock\_10\}$, the others either do not modify any variable involved by the predicate or state the equality. Consequently, we focus on:

$$\pi \in \{lock\_1, lock\_10\}$$

- $\pi = lock\_1$, this action sets $i_p = 1$ and is executed when $s_0.pc_p = pc1$. By invariant 16F (page 59) we know that $s_0.i_p = 1$, consequently $s_0.level_p = 0$ or $s_0.level_p = 1$. Observe that when $s_0.level_p = 0$, setting $i_p = 1$ satisfies $s_1.level_p = s_1.i_p - 1$. When $s_0.level_p = 1$, setting $i_p = 1$ satisfies $s_1.level_p = i_p$. Consequently, the conclusion comes. □

- $\pi = lock\_10$, this action increments the value of i_p and is executed when $pc_p = pc10$. By inductive hypothesis $level_p = i_p$ or $level_p = i_p - 1$, there are two cases:

  Case 1.1. $s_0.level_p = i_p$, incrementing i satisfies $s_1.level_p = s_1.i_p - 1$. □

  Case 1.2. $s_0.level_p = i_p - 1$; by invariant 17F (page 60) we know that $s_0.i_p = level_p$. This is a contradiction. □

Case 2. Conversely, suppose the generic executor is not p, $proc(\pi) \neq p$, then the transition function can not modify p's *level* and i variables. Consequently, the invariant is preserved by all actions. □

Invariant 19F (Not yet scanned)**.** In all reachable states, qq is preserved unscanned after it is chosen and before it is scanned.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow \forall p \in \mathbb{P} : s_0.pc_p = pc7 \Rightarrow s_0.qq_p \notin s_0.S_p$$

Proof. By induction on reachable states of the automaton.
Basis. Let s_0 be an initial state of the automaton.

$$s_0 \in I(FL) \vdash \forall p \in \mathbb{P} : s_0.pc_p = pc7 \Rightarrow s_0.qq_p \notin s_0.S_p$$

Let p be the generic process in initial states; observe that in initial states all processes have pc set to $pcIdle$, consequently the consequent is vacuously satisfied. □
Inductive step. Let s_0 and s_1 be states of the automaton, let s_0 be reachable; we assume the invariant holds in state s_0 by inductive hypothesis.

$$\forall p \in \mathbb{P} : s_0.pc_p = pc7 \Rightarrow s_0.qq_p \notin s_0.S_p, reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash$$
$$\forall p \in \mathbb{P} : s_1.pc_p = pc7 \Rightarrow s_1.qq_p \notin s_1.S_p$$

Let p be the generic process in state s_0. There are two cases:
Case 1. Suppose the generic executor is p: $proc(\pi) = p$. We must analyse all actions of the automaton. Observe that the following actions

$$\pi \in \left\{ \begin{array}{l} lock\_invoke, lock\_1, lock\_2T, lock\_2F, lock\_3, lock\_4, lock\_5, \\ lock\_6F, lock\_7T, lock\_7F, lock\_8T, lock\_8F, lock\_9T, lock\_9F, \\ lock\_10, unlock\_1, lock\_response \end{array} \right\}$$

invalidate the premise of the consequent formula, vacuously satisfying it. Consequently, we focus on $\pi = lock\_6T$.

- $\pi = lock\_6T$, the precondition allows to execute the action only when $s_0.qq \notin s_0.S_p$. The action does not add qq to S_p, satisfying and preserving the property. □

Case 2. Conversely, suppose the generic executor is not p, $proc(\pi) \neq p$, then the transition function can not modify p's program counter and qq variable. Consequently, the invariant is preserved by all actions. □

Invariant 20F (Either in S or in C)**.** In all reachable states, sets S and C host complementary elements.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow \forall p, q \in \mathbb{P} : q \in s_0.S_p \Leftrightarrow \neg q \in s_0.C_p$$

Proof. By induction on reachable states of the automaton.
Basis. Let s_0 be an initial state of the automaton.

$$s_0 \in I(FL) \vdash \forall p, q \in \mathbb{P} : q \in s_0.S_p \Leftrightarrow \neg q \in s_0.C_p$$

Observe that in initial states $\forall p \in \mathbb{P} : S_p = \top \wedge C_p = \emptyset$, trivially satisfying the consequent. □
Inductive step. Let s_0 and s_1 be states of the automaton, let s_0 be reachable; we assume the invariant holds in state s_0 by inductive hypothesis.

$$\begin{array}{l} \forall p, q \in \mathbb{P} : q \in s_0.S_p \Leftrightarrow \neg q \in s_0.C_p, reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash \\ \forall p, q \in \mathbb{P} : q \in s_1.S_p \Leftrightarrow \neg q \in s_1.C_p \end{array}$$

Let p and q be generic processes. There are two cases:
Case 1. Suppose that p or q takes a step: $proc(\pi) = p \vee proc(\pi) = q$. There are two subcases:
Case 1.1. Suppose that p takes a step and q is suspended: $proc(\pi) = p$. Since q is suspended its local and private variables remain unchanged. Observe that the only actions modifying variables S and C are:

$$\pi \in \left\{ \; lock\_4, lock\_5, lock\_7T, lock\_7F, lock\_8F, lock\_8T \; \right\}$$

Consequently, we focus on them:

- $\pi = lock\_4$, this action sets $s_1.S = \{proc(\pi)\}$ and $s_1.C_p = s_1.S_p^c$. Since p takes a step, set S contains only p. Therefore, it is false that $\{q\} \in s_1.S_p$ and it is true that $\{q\} \in s_1.C_p$, satisfying the consequent. □
- $\pi = lock\_5$, analogous to case $\pi = lock\_4$. □
- $\pi = lock\_7T$, this action records qq_p as scanned, adding it to S and removing it from C. By inductive hypothesis we have that $\{q\} \in s_0.S_p \Leftrightarrow \{q\} \notin s_0.C_p$. This action adds qq to S and removes it from C; there are two subcases:

  Case 1.1.3.1. $qq = q$, then it is added to S and removed from C, satisfying the consequent. □

  Case 1.1.3.2. $qq \neq q$, then q was in S and not in C, satisfying the consequent. □
- $\pi = lock\_7F$, this action empties S and fills C, hence $s_1.S_p = \emptyset$ and $s_1.C_p = \top$. This trivially satisfy $\{q\} \in s_1.S_p \Leftrightarrow \{q\} \notin s_1.C_p$ because it is false that $\{q\} \in \emptyset$ and it is true that $\{q\} \in \top$.

- $\pi = lock\_8F$, analogous to case $\pi = lock\_7F$. □
- $\pi = lock\_8T$, analogous to case $\pi = lock\_7F$. □

Case 1.2. Conversely, suppose that q takes a step and p is suspended: $proc(\pi) = q$. This case is specular to **case 1.1**. □
Case 2. Conversely, suppose the generic executor is neither p nor q, $proc(\pi) \neq p \wedge proc(\pi) \neq q$, then the transition function can not modify their program counter value or their C and S variables. Consequently, the invariant is preserved by all actions. □

Invariant 21F (S is Complement of C)**.** In all reachable states, sets S and C are complementary.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow \forall p \in \mathbb{P} : s_0.S_p = s_0.C_p^c$$

Proof. By induction on reachable states of the automaton.
Basis. Let s_0 be an initial state of the automaton.

$$s_0 \in I(FL) \vdash \forall p \in \mathbb{P} : s_0.S_p = s_0.C_p^c$$

Let p be the process in initial states. Observe that initial states are characterized by having $\forall p \in \mathbb{P} : S_p = \emptyset \wedge C_p = \top$, trivially satisfying the consequent. □
Inductive step. Let s_0 and s_1 be states of the automaton, let s_0 be reachable; we assume the invariant holds in state s_0 by inductive hypothesis.

$$\begin{array}{l}\forall p \in \mathbb{P} : s_0.S_p = s_0.C_p^c, reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash \\ \forall p \in \mathbb{P} : s_1.S_p = s_1.C_p^c\end{array}$$

Let p be a generic process. There are two cases:
Case 1. Suppose that process p takes a step: $proc(\pi) = p$. Observe that the only actions that modify involved variables, S and C, are:

$$\pi \in \{\ lock\_4, lock\_5, lock\_7T, lock\_7F, lock\_8F\ \}$$

Consequently, we focus on these:

- $\pi = lock\_4$, this action sets $S_p = \{proc(\pi)\}$ and $C_p = \top \setminus \{proc(\pi)\}$, consequently satisfying the consequent. □
- $\pi = lock\_5$, this case is analogous to case $\pi = lock\_4$. □
- $\pi = lock\_7T$, this action adds qq to S_p and removes it from C_p. By inductive hypothesis we have $s_0.S_p = s_0.C_p^c$, then the addition of qq to S_p and the removal of qq from C_p correspond to claim that $s_1.S_p = s_0.S_p \cup \{s_0.qq_p\}$ and $s_1.C_p = s_0.C_p \setminus \{s_0.qq_p\}$. Consequently, rewriting the consequent formula substituting each set with its updated version, we get:

  $$s_1.S_p = s_1.C_p^c \Leftrightarrow s_0.S_p \cup \{s_0.qq_p\} = (s_0.C_p \setminus s_0.qq_p)^c$$

  Rewriting $s_0.S_p$ with the inductive hypothesis we get:

  $$\begin{array}{c}s_0.C_p^c \cup \{s_0.qq_p\} = (s_0.C_p \setminus s_0.qq_p)^c \Leftrightarrow \\ s_0.C_p^c \cup \{s_0.qq_p\} = (s_0.C_p \cap \{s_0.qq_p\}^c)^c \Leftrightarrow \\ s_0.C_p^c \cup \{s_0.qq_p\} = s_0.C_p^c \cup \{s_0.qq_p\}.\end{array}$$

  This is a tautology. □

- $\pi = lock\_7F$, this action sets $S_p = \emptyset$ and $C_p = \top$, consequently satisfying the consequent. □
- $\pi = lock\_8F$, this case is analogous to case $\pi = lock\_7F$. □

Case 2. Conversely, suppose the generic executor is not p, $proc(\pi) \neq p$, then the transition function can not modify C or S variables. Consequently, the invariant is preserved by all actions. □

The proof script of invariant 21F is a good example of the employment of rewrite rules over brute-force definition expansion. Rewrite rules quicken the proof re-execution and develop the proof step-by-step allowing the user to see each modification of the proof context. The disadvantage is that the user must know they exist. In case the user does not want or does not know how to employ them he/she should resort to decision procedures that give to the conclusive part of the proof a mysterious appearance.

Invariant 22F (C has Bounded Size.)**.** In all reachable states, each process has $|\mathbb{P}|$ candidate processes to scan at most.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow \forall p \in \mathbb{P} : |s_0.C_p| \leq |\mathbb{P}|$$

Proof. By induction on reachable states of the automaton.
Basis. Let s_0 be an initial state of the automaton.

$$s_0 \in I(FL) \vdash \forall p \in \mathbb{P} : |s_0.C_p| \leq |\mathbb{P}|$$

Let p be a generic process in the initial state s_0, observe that initial states are characterized by $C_p = \top$. By lemma 6F (page 67), $|\top| = |\mathbb{P}|$, that satisfies the consequent. □
Inductive step. Let s_0 and s_1 be states of the automaton, let s_0 be reachable; we assume the invariant holds in state s_0 by inductive hypothesis.

$$\forall p \in \mathbb{P} : |s_0.C_p| \leq |\mathbb{P}|, reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash$$
$$\forall p \in \mathbb{P} : |s_1.C_p| \leq |\mathbb{P}|$$

Let p be a generic process. There are two cases:
Case 1. Suppose th generic executor is p: $proc(\pi) = p$. Observe that all actions but the following trivially preserve the property:

$$\pi \in \{\ lock\_4, lock\_5, lock\_7T, lock\_7F, lock\_8F\ \}$$

Also recall that by inductive hypothesis $|s_0.C_p| \leq |\mathbb{P}|$ and by lemma 6F (page 67) we have $|\top| = |\mathbb{P}|$.

- $\pi = lock\_4$, this action sets $victim[i_p] = p$, $S = \{p\}$ and $C = \top \setminus \{p\}$. The removal of p from C_p, causes the decrease of C_p's cardinality, satisfying the consequent. □
- $\pi = lock\_5$, this case is analogous to case $\pi = lock\_4$. □
- $\pi = lock\_7T$, this action remove qq from C_p. By invariant 19F (page 61) we know that $s_0.qq \notin s_0.S_p$, consequently by invariant 20F (page 62) we know that $s_0.qq \in s_0.C_p$. The removal of an element that belongs to the set implies a decrease in the cardinality; that by previous observation equals $|\mathbb{P}|$. Hence, the transition causes the decrease, satisfying the consequent. □
- $\pi = lock\_7F$, this actions sets $C_p = \top$, satisfying the consequent. □
- $\pi = lock\_8F$, this case is analogous to case $\pi = lock\_7F$. □

Case 2. Conversely, suppose the generic executor is not p, $proc(\pi) = p$, then the transition function can not modify C_p. Consequently, the invariant is preserved by all actions. □

Invariant 23F (C and S' Sizes are complementary)**.** In all reachable states, sets S and C have complementary size.

$$\forall s_0 \in Q(FL) : reach(FL, s_0) \Rightarrow \forall p, q \in \mathbb{P}, n \in \mathbb{N} : |s_0.S_p| = n \Leftrightarrow |s_0.C_p| = |\mathbb{P}| - n$$

Proof. By induction on reachable states of the automaton.
Basis. Let s_0 be an initial state of the automaton.

$$s_0 \in I(FL) \vdash \forall p, q \in \mathbb{P}, n \in \mathbb{N} : |s_0.S_p| = n \Leftrightarrow |s_0.C_p| = |\mathbb{P}| - n$$

Let p and q be generic processes in state s_0, observe that in initial states, $C = \top$ and $S = \emptyset$. By lemma 6F (page 67) we know that $|\top| = |\mathbb{P}|$. Therefore, the consequent is satisfied. □
Inductive step. Let s_0 and s_1 be states of the automaton, let s_0 be reachable; we assume the invariant holds in state s_0 by inductive hypothesis.

$$\forall p, q \in \mathbb{P}, n \in \mathbb{N} : |s_0.S_p| = n \Leftrightarrow |s_0.C_p| = |\mathbb{P}| - n, reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash$$
$$\forall p, q \in \mathbb{P}, n \in \mathbb{N} : |s_1.S_p| = n \Leftrightarrow |s_1.C_p| = |\mathbb{P}| - n$$

Let p and q be generic processes. There are two cases:
Case 1. Suppose p is the generic executor: $proc(\pi) = p$. Observe that all actions but the following preserve the property:

$$\pi \in \{\ lock\_4, lock\_5, lock\_7T, lock\_7F, lock\_8F\ \}$$

Consequently, we focus on these:

- $\pi = lock\_4$, this action sets $victim[i_p] = p$, $S_p = \{p\}$ and $C_p = \top \setminus \{p\}$. The cardinality of S is 1, by lemma 6F (page 67) and by the removal of p we know that $|C| = |\mathbb{P}| - 1$. Hence, the consequent is satisfied. □
- $\pi = lock\_5$, this case is analogous to $\pi = lock\_4$. □
- $\pi = lock\_7T$, this action add qq to S_p and removes it from C_p. By invariant 19F (page 61), we know that $s_0.qq \notin s_0.S_p$, by lemma 6F (page 67) we know the full set has cardinality N. We induct on n.

  Basis. We split the proof into two base cases:

  Case (⇐). We must show the premise of the implication is unsatisfiable: it is because an element has been removed.

  $$|s_1.C_p| = |\mathbb{P}| - 0 \Rightarrow |s_1.S_p| = 0$$

  We assume by inductive hypothesis that $|s_1.C_p| = |\mathbb{P}|$. By invariant 19F (page 61) we know that $s_0.qq \notin s_0.S_p$, and by invariant 20F (page 62) we know that $s_0.qq \in s_0.C_p$. Observe that the removal of qq from $s_0.C_p$ must cause a decrease in the size of the set. By inductive hypothesis the size of set in the post-state is $|\mathbb{P}|$, consequently in the pre-state the cardinality is $|\mathbb{P}| + 1$. By invariant 22F (page 64) we know that the set C_p has an upper bound on the cardinality equal to $|\mathbb{P}|$. This is a contradiction. □

  Case (⇒). We must show the premise of the implication is unsatisfiable: it is because an element has been removed.

  $$|s_1.S_p| = 0 \Rightarrow |s_1.C_p| = |\mathbb{P}| - 0$$

  We assume by inductive hypothesis that $|s_1.S_p| = 0$. By invariant 19F (page 61) we know that $s_0.qq \notin s_0.S_p$, observe that the addition of $s_0.qq$ to $s_0.S_p$ must cause an increase in the size of the set. This invalidates the premise, making the implication vacuously satisfied. □

  Inductive step. The proof context is:

  $$\forall p, q \in \mathbb{P}, n \in \mathbb{N} : |s_0.S_p| = n \Leftrightarrow |s_0.C_p| = |\mathbb{P}| - n, reach(FL, s_0), s_0 \xrightarrow{\pi_{proc(\pi)}} s_1 \vdash$$
  $$\forall j : \Big(|s_1.S_p| = j \Leftrightarrow |s_1.C_p| = |\mathbb{P}| - j\Big) \Rightarrow \Big(|s_1.S_p| = (j+1) \Leftrightarrow |s_1.C_p| = |\mathbb{P}| - (j+1)\Big)$$

  Let j be a generic natural. We assume by inductive hypothesis that $|s_1.S_p| = j \Leftrightarrow |s_1.C_p| = |\mathbb{P}| - j$. By invariant 19F we know that $s_0.qq \notin s_0.S_p$, and by invariant 20F that $s_0.qq \in s_0.C_p$. The addition of qq to S_p and its removal from C_p, in conjunction with the inductive hypothesis allow to conclude that $|s_1.S_p| = (j+1) \Leftrightarrow |s_1.C_p| = |\mathbb{P}| - (j+1)$. This satisfies the consequent. □

- $\pi = lock\_7F$, this actions sets $S_p = \emptyset$ and $C_p = \top$. The cardinality of the empty set is 0, by lemma 6F (page 67) the cardinality of the full set is $|\mathbb{P}|$, consequently the consequent is satisfied. □
- $\pi = lock\_8F$, this case is analogous to $\pi = lock\_7F$. □

Case 2. Conversely, suppose the generic executor is not p, $proc(\pi) \neq p$, then the transition function can not modify C_p or S_p. Consequently, the invariant is preserved by all actions. □

4.2.4 Minor Lemmata

In this subsection we state and prove three minor lemmata that supported several intermediate results; we had to mechanically prove them because PVS does not provide handy analogues.

Lemma 4F (Unique Element)**.** Sets whose cardinality is one contain a unique element.

$$\forall(S : finiteset) : |S| = 1 \Rightarrow \Big((\exists p : p \in S) \wedge \big(\forall p, q : (p \in S \wedge q \in S) \Rightarrow p = q\big)\Big)$$

Proof. By direct proof. Let S be a generic finite set, we assume true $|S| = 1$. We split the proof into two cases:
Case 1. We prove $\exists p \in S$. By definition of $|S| \geq 1$ we know that an element belonging to S exists. This trivially proves the conclusion. □
Case 2. We prove $\forall p, q : (p \in S \wedge q \in S) \Rightarrow p = q\big)$. By definition of equinumerosity we know that some set S has cardinality n if and only if there exists a bijection from its elements to the set of natural numbers below n, its cardinality. We use this definition to prove the theorem; the proof context becomes:

$$\begin{array}{l} p \in S, q \in S, |S| = 1, \\ \forall n \in \mathbb{N}, (S : finiteset) : |S| = n \Leftrightarrow \exists\big(f : (S \to \{0, \ldots, n-1\})\big) : bijective(f) \vdash \\ p = q \end{array}$$

Let $n = 1$ by instantiation, then there exists a bijection f mapping S to naturals below 1. f is bijective and surjective. Expand the definition of injection:

$$\forall x, y \in S : f(x) = f(y) \Rightarrow x = y$$

Let $p = x$ and $q = y$ by instantiation and apply *contraposition*, the property of implication stating that:

$$\forall(P, Q : bool) : P \Rightarrow Q \Leftrightarrow \neg Q \Rightarrow \neg P$$

Define predicates $P := f(p) = f(q)$ and $Q := p = q$. Consequently:

$$\begin{array}{lll} f(p) = f(q) \Rightarrow p = q, & & \\ f(p) = f(q) \Rightarrow p = q & \Leftrightarrow & \neg\big(p = q\big) \Rightarrow \neg(f(p) = f(q)) \\ & \Rightarrow & p \neq q \Rightarrow f(p) \neq f(q) \end{array}$$

The proof context becomes:

$$\begin{array}{l} p \in S, q \in S, |S| = 1, surjective(f), f(p) = f(q) \Rightarrow p = q, p \neq q \Rightarrow f(p) \neq f(q) \vdash \\ p = q \end{array}$$

By rule $(\neg R)$ of Sequent Calculus (SC) and the consequent $p = q$, we deduce the antecedent $p \neq q$, therefore, we know that $f(p) \neq f(q)$. Since $f : S \to \{0\}$, $p \in S$, $q \in S$ and $p \neq q$ we reach a contradiction. □

The role of lemma 4F is the one to simplify mechanical proofs. In its mechanical proof, we took advantage of PVS' prelude file and the definition of set whose cardinality is greater equal one.

Lemma 5F (Two Elements)**.** Sets whose cardinality is two contain two distinct elements.

$$\forall(S : finiteset) : |S| = 2 \Rightarrow \Big((\exists p, q \in S \wedge p \neq q) \wedge \big(\forall p, q, r : p \in S \wedge q \in S \wedge r \in S \Rightarrow p = q \vee q = r \vee p = r\big)\Big)$$

Proof. By direct proof. Let S be a generic finite set, we assume its size is two. We split the proof into two branches:
Case 1. We prove $\exists p, q \in S \wedge p \neq q$.

$$\vdash \forall(S : finiteset) : |S| = 2 \Rightarrow (\exists p, q \in S \wedge p \neq q)$$

When cardinality of S is greater or equal two, there exist two distinct elements belonging to it. Let said elements be x and y such that $x \neq y$. This proves the conclusion. □
Case 2. We prove $\forall p, q, r : (p \in S \wedge q \in S \wedge r \in S) \Rightarrow (p = q \vee q = r \vee p = r)$.

$$\vdash \forall(S : finiteset) : |S| = 2 \Rightarrow \forall p, q, r : (p \in S \wedge q \in S \wedge r \in S) \Rightarrow (p = q \vee q = r \vee p = r)$$

Let p,q and r be generic elements of set S. The proof context becomes:

$$|S| = 2, p \in S, q \in S, r \in S \vdash p = q, q = r, p = r$$

The definition of equinumerosity states the existence of a bijection mapping elements of S to naturals below n, its cardinality. Since by hypothesis the cardinality of S is two, there exist a bijection from S to $\{0, 1\}$. Let f be such mapping, it is bijective, consequently injective and surjective. By definition of injection we know that:

$$f(p) = f(q) \Rightarrow p = q$$
$$f(q) = f(r) \Rightarrow q = r$$
$$f(p) = f(r) \Rightarrow p = r$$

Consequently, by contraposition.

$$p \neq q \Rightarrow f(p) \neq f(q)$$
$$q \neq r \Rightarrow f(q) \neq f(r)$$
$$p \neq r \Rightarrow f(p) \neq f(r)$$

By rule $(\neg R)$ of SC, we observe that proving $p = q \vee q = r \vee p = r$ corresponds to assume that $p \neq q \wedge q \neq r \wedge p \neq r$. Consequently, $f(p) \neq f(q)$, $f(q) \neq f(r)$ and $f(p) \neq f(r)$. Since $f : S \to \{0, 1\}$ and by hypothesis $p \in S$, $q \in S$, $r \in S$ and $p \neq q$, $q \neq r$ and $p \neq r$, we reach a contradiction. □

The role of lemma 5F is the one to simplify mechanical proofs. In its mechanical proof, we took advantage of PVS' prelude file and the definition of set whose cardinality is greater equal two.

Lemma 6F (Full Set has Full Size)**.** The set gathering all process IDs and the set of processes are equinumerous.

$$|\top| = |\mathbb{P}|$$

Proof. By direct proof. By definition of equinumerosity we know that some set has cardinality n if and only if there exists a bijection from its elements to the set of natural numbers below n. We use this definition to prove the theorem; the proof context becomes:

$$\forall n \in \mathbb{N}, (S : finiteset) : |S| = n \Leftrightarrow \exists\big(f : (S \to \{0, \ldots, n-1\})\big) : bijective(f) \vdash |\top| = |\mathbb{P}|$$

Let $n = |\mathbb{P}|$ and $S = \top$ by instantiation, to prove the conclusion we must identify a bijective function satisfying the definition. We split the equivalence into left and right implication.

$$|\top| = |\mathbb{P}| \Rightarrow \exists\big(f : (\top \to \{0, \ldots, |\mathbb{P}| - 1\})\big) : bijective(f),$$
$$\exists\big(f : (\top \to \{0, \ldots, |\mathbb{P}| - 1\})\big) : bijective(f) \Rightarrow |\top| = |\mathbb{P}| \vdash$$
$$|\top| = |\mathbb{P}|$$

There are three cases:
Case 1. Suppose $|\top| = |\mathbb{P}|$, then antecedent -1 is satisfied. Antecedent -1 implies antecedent -2 that in turn implies the conclusion. □
Case 2. Conversely, suppose that $|\top| \neq |\mathbb{P}|$, then antecedent -1 becomes useless and can be deleted. The proof context becomes:

$$\exists\big(f : (\top \to \{0, \ldots, |\mathbb{P}| - 1\})\big) : \mathit{bijective}(f) \Rightarrow |\top| = |\mathbb{P}| \vdash |\top| = |\mathbb{P}|$$

Suppose the premise of implication -1 is satisfied, then the cardinality of the fullset is $|\mathbb{P}|$ and the conclusion comes. □
Case 3. Conversely, suppose the premise is not satisfied, then antecedent -1 becomes useless and can be deleted. The proof context becomes:

$$\neg\big(\exists\big(f : (\top \to \{0, \ldots, |\mathbb{P}| - 1\})\big) : \mathit{bijective}(f)\big) \vdash |\top| = |\mathbb{P}|$$

Observe that *identity* is a bijective function that maps elements of set $\top$ to naturals below $|\mathbb{P}|$. Consequently, we reached a contradiction. □

Chapter 5

Conclusions

In the previous chapter we provided the full formalization of two well-know and famous algorithms satisfying mutual exclusion: Peterson-Lock and Filter-Lock. For each algorithm we provided the Input Output Automaton (IOA) encoding it and the properties required to prove it correct; for each formalization we provided its mechanical counterpart carrying the same elements expressed in PVS language. Proofs hereby provided are formal and purposely focused on relevant cases of each invariant, that is, they develop the argumentation completely; they can be compared against their mechanical versions where low-level details appear and discussions are removed. Several mechanical proofs of low-level and intermediate-level invariant properties are synthetic, that is, they do not develop the argumentation deeply and take advantage of powerful strategies. However, no loss of information occurs because detailed formal proofs have been provided in the previous chapter. In this chapter we discuss the conclusions we have drawn through the completion of the present work.

In section 5.1, we first discuss the role of Model Checking (MC) in Interactive Theorem Proving (ITP).

In section 5.2, we first discuss the role of Prototype Verification System (PVS) and its features: decision procedures and the knowledge base. The discussion covers, several aspects of ITP in PVS that to the best of our knowledge are not tackled in the works we have considered [9, 10, 15, 22]. The section closes with the sketch of an approach to become effective at ITP with PVS.

5.1 The Role of Model Checking (MC)

ITP is hard, this is why demonstrators are interested in to support it with MC: in [9, 10, 15] the authors discuss the difficulties they found in proving several data structures linearizable. One of them being the discovery of design errors that render the proving process pointless. According to the authors, the inclusion of MC in ITP is a defensive technique, wanting to avoid major wastes of time and energy. ITP is a recursive problem, that is, when some non-trivial property is required the user must identify and prove one or more intermediate results that support the property of interest. For each property, the user must provide the exact formal statement, and the proof. In case the desired result is very sophisticated, this task might be repeated several times generating many levels of indirection. In section 4.2, we have seen an example of this: the proof of invariant 7F (page 52) depends on several invariants that in turn depend on lower results. The discovery of supporting invariants is not error-free, that is, the user might include unprovable lemmata. In [9, 10, 15], MC is used to speed up ITP and make it error-free: supporting invariants are checked on the model and in case they hold, they are proved. Hence, Theorem Proving (TP) becomes an iterative process where ITP and MC sessions alternate. We employed the aforementioned methodology and found that invariant discovery can be approached in three ways: bottom-up, top-down and middle-out.

In bottom-up ITP the user proves invariants from the lowest to the highest level, meaning that no unproved invariant is ever used to support high-level results. This is the safest way for proving theorems, however it is also the slowest and most demanding one because whenever a new invariant is required the current proof must be interrupted and resumed only when all dependencies are proved. The advantage of this approach is that proved results are used as much as possible, the drawback is

that proofs supported by many intermediate results are often interrupted and this distracts the prover.

In top-down ITP the user proves theorems from the highest to the lowest level, that is, high-level invariants are stated and proved first. Whenever some supporting property is required, it is stated, employed and left unproved; once the current proof is over, attention can be directed to dependencies recursively. This approach is slightly risky because it could introduce inconsistent or overabundant supporting lemmata. However, it keeps the prover focused on the current proof, it is effective when the user has no doubt about supporting results and knows exactly how to prove them.

In middle-out ITP the user proves theorems employing top-down or bottom-up approaches locally. The element of interest becomes supporting results for high-level theorems or high-level results justified by low-level properties. This approach enjoys advantages and disadvantages from both techniques, it is the fastest and riskiest of all; however, it is safe when the property and its supporting results are know to be correct.

During our first proof attempt we exploited the aforementioned methodology: we first modelled the Filter-Lock in PROMELA (SPIN MC's meta-language [23]) and then formalized it in PVS specification language. We employed PVS to prove the algorithm correct while exploiting SPIN for checking invariants. Our proving and discovery methodology changed greatly, we tried the three directions preferring middle-out and bottom-up approaches. We resorted to top-down only when approaching analogue versions of some invariant. The alternation of MC and ITP is definitely useful for speeding up the proving process, still the introduction of MC is not enough for removing the actual bottleneck: comprehension of the problem on the human prover's side. All that being said, independently of the chosen direction we do not think this methodology is suitable to novice users approaching ITP. Hence, we discourage such practice for three reasons:

1. it fosters intellectual laziness, because it convinces the user that invariants can be searched on demand and that understanding the problem is not really necessary,

2. it should be considered only when it pays back and reasonable educated guesses can be made by the (experienced) user: beginners are more likely to get stuck in the process,

3. we observed experimentally that MC ceased to be of help very quickly. In fact, it could give no clue about three complex assertions: invariants 1F, 3F and 4F.

We think users approaching ITP should focus on finding proofs and invariants more intuitively in a top-down fashion and involve MC only after several successful verifications have been completed.

5.2 The Role of PVS' Features

In this section we discuss the role of PVS' features: the support provided by automation and decision procedures and the PVS knowledge base. In fact, these are double-edged weapons: we discuss their advantages and disadvantages, the difficulties they pose and means to overcome them.

5.2.1 Automation and Decision Procedures

Resources introducing ITP in PVS stress the importance of employing "the most powerful decision procedures available" and only in case of failed proof attempts turn to less powerful ones [11, 33]. PVS' decision procedures operate hierarchically, namely powerful commands are wrapping of many weaker ones with the introduction of rewriting steps; consequently, the most powerful command *grind* is responsible for running all possible decision procedures. PVS Version 7.1 is currently equipped with 16 commands applying one or more decision procedures, each one having a precise granularity [33, Chapter 4]. The PVS community uses the term "decision procedure" generically to identify heterogeneous techniques that joined together increase the power of the Proof Assistant (PA); decision procedures encompass: reasoning about equalities and linear inequalities over naturals and reals, propositional simplification, rewrites using registered definitions, beta reductions, simplification via binary decision diagrams, heuristic instantiations and more. In our perspective, powerful commands tend to backfire, especially in early proofs because despite their power, these procedures do not know

where they should focus their attention and simply do as much work as possible whenever it is possible. Hence, they develop all formulae in the current context including those that do not directly prove the conclusion, wasting time and electricity. Consequently, the application of general decision procedures that fail to prove some sub-branch result in extremely weird sequents that from our perspective are almost always indiscernible. On the other hand, when *grind* is capable of proving a sub-goal, the novice PVS users could find difficult to understand why the branch is discharged: the employment of *grind* could correspond to many rewrite steps and heuristic instantiations that give to the proof a mysterious appearance. Consequently, the user noticing the end of the proof finds itself to wonder why the branch is discharged. In fact, crucial details regarding proof branches can be forgotten in a few hours, in these cases subsequent (re)visits of the same proof are filled with recurring doubts. Therefore, forfeiting the employment of powerful commands is a defensive approach: those commands do not show which formulae are part of the conclusion and this is why in our proof-scripts, we often discharge proof branches employing the least powerful command. Hiding or deleting unnecessary formulae is an alternative technique that mitigates the problem; however, developing the few steps that show the conclusion comes with two advantages: it makes the proof clear on every (re)visit and it provides a convenient explanation that can be mapped on paper.

5.2.2 PVS' Knowledge Bases

PVS is shipped with two knowledge bases: "prelude file" and "NASA library". The prelude file is basically a text file containing an ordered list of theories and proved theorems. Its contents are highly sophisticated and give PVS the majority of its proving power: it contains hundreds of rewrite rules ranging from set theory to exotic induction schemes, the version shipped with PVS Version 7.1 contains 1804 theorems. We did not employ NASA library because it is extremely vast and its contents are too specific with respect to our needs. In our formalization, the prelude played a relevant role providing us lemmata that from a mechanical perspective are strictly required. The user approaching PVS and assessing material provided by National Areonautic and Space Administration (NASA) might suspect that basic commands are sufficient for proving all relevant theorems but this is not the case: when set theory comes into play the proof assistant is difficult to guide toward the conclusion via basic commands alone. This is due to the lack of means to reason about sets through commands; from an intuitive perspective these facts are obvious, but it takes adequate theorems to link premises and conclusions. Hence, during our formalization we took advantage of the prelude and resorted to the following theorems from finite sets theory:

$$
\begin{aligned}
card\_add &:= \forall(S : finiteset[T]), (x : T) : |S \cup \{x\}| = |S| + (\text{if } x \in S \text{ then 1 else 0}) \\
card\_remove &:= \forall(S : finiteset[T]), (x : T) : |S \setminus \{x\}| = |S| - (\text{if } x \in S \text{ then 1 else 0}) \\
same\_card\_subset &:= \forall(A, B : finiteset[T]) : A \subseteq B \wedge |A| = |B| \Rightarrow A = B \\
card\_1\_has\_1 &:= \forall(S : finiteset[T]) : |S| >= 1 \Rightarrow \exists x : x \in S \\
card\_2\_has\_2 &:= \forall(S : finiteset[T]) : |S| >= 2 \Rightarrow \exists x, y : x \in S \wedge y \in S \wedge x \neq y \\
card\_subset &:= \forall(A, B : finiteset[T]) : A \subseteq B \Rightarrow |A| \leq |B| \\
card\_bij &:= \forall(S : finiteset[T]), n \in \mathbb{N} : |S| = n \Leftrightarrow \exists(f : [0 - n] \to S) : bijective(f)
\end{aligned}
$$

Reported theorems are intuitively clear, the last one is based on the definition of equinumerosity. Their proofs are not difficult to carry out and comparing our proof style with the one used in the prelude, we observed that several ad hoc lemmata have been cleverly used to rewrite their statements appropriately. Rewriting rules are equalities used by PVS to rewrite formulae, we took advantage of what follows:

$$
\begin{aligned}
add\_as\_union &:= \forall(A : finiteset[T]), (x : T) : add(x, A) = A \cup \{x\} \\
card\_emptyset &:= |\emptyset| = 0 \\
complement\_complement &:= \forall(A : finiteset[T]) : (A^C)^C = A \\
complement\_emptyset &:= \emptyset^C = \top \\
complement\_fullset &:= \top^C = \emptyset \\
demorgan2 &:= \forall(A, B : finiteset[T]) : (A \cap B)^c = A^c \cup B^c \\
difference\_intersection &:= \forall(A, B : finiteset[T]) : A \setminus B = A \cap B^c \\
remove\_as\_difference &:= \forall(A : finiteset), (x : T) : remove(x, A) = A \setminus \{x\}
\end{aligned}
$$

These properties are extremely handy, the user that does not (or can not) access them must devise analogous results to complete the formalization.

5.3 Learning to Prove

PVS is remarkable: the mechanical rigour and its intuitive proof system render it a valuable analysis tool. However, in our opinion it can not be the only tool the user employs to clarify the correctness proof. PVS introduces low-level details in formalization, obscuring the formal approach and discouraging an abstract perspective.

Our first attempt at proving invariant 1F has been carried out on paper, this choice allowed us to workaround PVS' limitations and to skip obvious irrelevant steps, while allowing us to reason about the general idea that supports the theorem. We represented all formulae as show in chapter 3 and employed some colouring scheme for denoting why some branch was discarded. We are strongly convinced the novice user should approach ITP differently with respect to what we did, we sketch an approach we deem valuable. The novice user should:

1. prove simple algorithms correct focusing on formal proofs and not mechanical ones,
2. learn Sequent Calculus (SC) and its inference rules,
3. carry out simple mechanical proofs on paper,
4. rerun such proofs in PVS.

In our opinion each step, when faced in this order, allows to develop the skills the user must possess to be effective at mechanical proving.

1. Focuses on relevant aspects of the proof abstracting low-level details away,
2. structures the reasoning process, making it mechanical but most importantly inspectable,
3. teaches to merge abstract details with low-level ones without fostering intellectual laziness,
4. teaches to merge together long sequences of intermediate mechanical steps with the abstract reasoning that supports the proof.

Such steps do not involve MC, we stress again that its employment would be disruptive during the learning process.

Bibliography

[1] Myla Archer, Ben L. Di Vito, and César A. Muñoz. Developing User Strategies in PVS: A Tutorial. In *Proceedings of Design and Application of Strategies/Tactics in Higher Order Logics STRATA 2003*, pages 1–28, NASA/CP-2003-212448, NASA LaRC, Hampton VA 23681-2199, USA, 9 2003.

[2] Anish Athalye. CoqIOA: A Formalization of IO Automata in the Coq Proof Assistant. Master's thesis, Massachusetts Institute of Technology, 2017.

[3] Henk Barendregt. The Impact of the Lambda Calculus in Logic and Computer Science. *Bulletin of Symbolic Logic*, 3(2):181–215, 05 1997.

[4] Henk Barendregt and Erik Barendsen. Autarkic Computations in Formal Proofs. *Journal of Automated Reasoning*, 28(3):321–336, 2002. doi:10.1023/A:1015761529444.

[5] Rick Butler, Ben L. Di Vito, Alwyn Goodloe, Heber Herencia, Jeff Maddalon, Anthony Narkawicz, and Sam Owre. *NASA/NIA PVS Class 9-12/10/2012*, 2012. Online, last access 03 February 2021. URL: https://shemesh.larc.nasa.gov/PVSClass2012/.

[6] Cristian S. Calude and Declan Thompson. Incompleteness, Undecidability and Automated Proofs. In Vladimir P. Gerdt, Wolfram Koepf, Werner M. Seiler, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing*, pages 134–155, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-45641-6_10.

[7] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013. doi:10.7551/mitpress/9153.001.0001.

[8] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. *Model Checking and the State Explosion Problem*, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-35746-6_1.

[9] Robert Colvin, Simon Doherty, and Lindsay Groves. Verifying Concurrent Data Structures by Simulation. *Electronic Notes in Theoretical Computer Science*, 137:93–110, 07 2005. doi:10.1016/j.entcs.2005.04.026.

[10] Robert Colvin, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal Verification of a Lazy Concurrent List-Based Set Algorithm. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, pages 475–488, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. doi:10.1007/11817963_44.

[11] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A Tutorial Introduction to PVS. April 1995. URL: http://www.csl.sri.com/papers/wift-tutorial/.

[12] Nicolaas Govert de Bruijn. *AUTOMATH, a Language for Mathematics*, pages 159–200. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983. doi:10.1007/978-3-642-81955-1_11.

[13] Edsger W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, 8(9):569, sep 1965. doi:10.1145/365559.365617.

[14] Simon Doherty. Modelling and Verifying Non-Blocking Algorithms that use Dynamically Allocated Memory. Master's thesis, Victoria University of Wellington, 2003.

[15] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal Verification of a Practical Lock-Free Queue Algorithm. In David de Frutos-Escrig and Manuel Núñez, editors, *Formal Techniques for Networked and Distributed Systems – FORTE 2004*, pages 97–114, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. doi:10.1007/978-3-540-30232-2_7.

[16] Allen E. Emerson and Joseph Y. Halpern. Decision Procedures and Expressiveness in the Temporal Logic of Branching Time. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, page 169–180, New York, NY, USA, 1982. Association for Computing Machinery. doi:10.1145/800070.802190.

[17] Alberto Ercolani. Proving the Filter-Lock Correct in PVS. Master's thesis, University of Trento, 2021. URL: https://github.com/AlbertoErcolani/PetersonLock-final and https://github.com/AlbertoErcolani/FilterLock-final.

[18] Dov M. Gabbay, Jörg H. Siekmann, and John Woods. *Handbook of the History of Logic*, volume 9. North Holland, 225 Wyman Street, Waltham, MA 02451, USA, 2014.

[19] Gerhard Gentzen. Untersuchungen über das Logische Schließen. i. *Mathematische Zeitschrift*, 39:176–210, 1935. doi:10.1007/BF01201353.

[20] Leen Helmink, Martin Paul Alexander Sellink, and Frits W. Vaandrager. Proof-checking a Data Link Protocol. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages 127–165, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg. doi:10.1007/3-540-58085-9_75.

[21] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[22] Wim Hesselink. Mechanical Verification of Lamport's Bakery Algorithm. *Science of Computer Programming*, 78:1622–1638, 09 2013. doi:10.1016/j.scico.2013.03.003.

[23] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on software engineering*, 23(5):279–295, 1997. doi:10.1109/32.588521.

[24] Savas Konur. A Survey on Temporal Logics. *Frontiers of Computer Science*, 7(3):370–403, 2013. URL: http://arxiv.org/abs/1005.3199, arXiv:1005.3199.

[25] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[26] Nancy A. Lynch and Mark R. Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 2:219–246, 1989. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.83.7751.

[27] Paolo Masci and César A. Muñoz. An Integrated Development Environment for the Prototype Verification System. *Electronic Proceedings in Theoretical Computer Science*, 310:35–49, Dec 2019. URL: http://dx.doi.org/10.4204/EPTCS.310.5, doi:10.4204/eptcs.310.5.

[28] Joan Moschovakis. Intuitionistic Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Fall 2021 edition, 2021. https://plato.stanford.edu/archives/fall2021/entries/logic-intuitionistic/, Online, last access on 04 September 2021.

[29] Lawrence C. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5:363–397, 09 1989. doi:10.1007/BF00248324.

[30] Gary L. Peterson. Myths about the Mutual Exclusion Problem. *Information Processing Letters*, 12(3):115–116, 1981. URL: https://www.sciencedirect.com/science/article/pii/002001908190106X, doi:10.1016/0020-0190(81)90106-X.

[31] Michel Raynal. Concurrent Programming: Algorithms, Principles, and Foundations. In *Springer Berlin Heidelberg*, 2013. doi:10.1007/978-3-642-32027-9.

[32] John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, January 1965. doi:10.1145/321250.321253.

[33] Natarajan Shankar, Sam Owre, J. Rushby, and Dave WJ Stringer-Calvert. *PVS Version 7.1, PVS System Guide, PVS Prover Guide, PVS Language Reference*, 08 2020. https://pvs.csl.sri.com/doc/.

[34] Freek Wiedijk. *The Seventeen Provers of the World: Foreword by Dana S. Scott (Lecture Notes in Computer Science / Lecture Notes in Artificial Intelligence)*. Springer-Verlag, Berlin, Heidelberg, 2006. doi:10.1007/11542384.

Appendix A

List Of Assertions