SEMINAR REPORT
Seminar: Software Engineering in Winter term 2020

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

# Seminar "Verification and Model checking"

Ayush Pandey
Technische Universität Kaiserslautern, Department of Computer Science



## 1 Abstract

Modern information systems are growing in size, both, in terms of the number of components they are constructed from and the individual complexities of the components themselves. A single processor in a modern day personal computer has the capability of executing millions of instructions per second. The task of harnessing the power of these systems, however, comes with a challenge.

With the systems size increasing, the code base for software systems very often reaches millions of lines of code. It is therefore inevitable that some errors are present in the systems. Exhaustive testing of the functionalities is one way to ascertain the correctness, but it is also time-consuming and requires a significant amount of effort to cover all critical and edge cases.

Software systems implement a solution to a real world problem. Hence, it is possible to formally model them. The most common way of modelling systems is through labelled transition diagrams, also called state machines. These transition diagrams can be further represented as logical formulas with varying degrees of detail. With a successful model of the system, Model checking provides a way to formally check if this model meets the requirements of a given specification.

The problem we tackled in this seminar is a classical algorithm in the distributed and concurrent algorithms. Mutual exclusion [10] is a property which specifies concurrency control for the purpose of preventing race conditions over accesses to a shared resource. The requirement is that in a concurrent setting where multiple processes (or threads) use the same resource, a process never enters a critical section while another process is entering or is already in its critical section. One of the ways to introduce mutual exclusion in a system is to use locks. A classical example of a lock is **Peterson's Algorithm** or Peterson's Lock.

There are several tools which have been developed to automate and simplify the process of model checking. Some examples are: TLA$^+$, BLAST [9], Java Pathfinder [7], Promela [11], Isabelle [8] etc. In this report, we discuss the two major methods of software verification, namely, **Model checking** and **Deduction**. We also present the differences between the two approaches and how software verification can be done using TLA$^+$

# Contents

# 2 Introduction to Software Verification

Software verification aims towards checking whether the specified system fulfils the qualitative requirements that have been identified. Such verification is typically done using formal methods of mathematics.

There are two major approaches that can be taken to perform software verification, **Model checking** and **Deduction**. With model checking, a systematic exhaustive exploration of the states of the system is performed. This is usually only possible if the system follows a finite behavioural model or where the states can be represented finitely by abstraction. While model checking is often fully automatic, it does not perform well with systems which are significantly large.

The deductive approach deals with *proof obligations* and is also referred to as *Theorem proving*. The interpretation of the obligation decides the conformance of the system with the property being verified. Generally, the obligations are checked against the specification of the system using proof assistants. TLAPS is the proof assistant that TLA$^+$uses. Theorem proving is an involved process. It requires the user to know both, the mathematical foundation of the system, as well as, how the correctness can be specified using the constructs of the proof language. In this report, we present in detail how a proof is written in TLA$^+$.

# 3 TheTLA$^+$Language

## 3.1 Introduction

TLA$^+$is a formal specification language. TLA is an acronym for **Temporal Logic of Actions** [3]. It was developed by Leslie Lamport in 1999. The main use of TLA$^+$is towards specifying concurrent and distributed systems. The models in TLA$^+$follow a mathematical approach and are based on **First-order logic** and **Zermelo-Fraenkel set theory**. The purpose of employing a formal verification tool is to identify the flaws in the system before the model is realized in any programming language and a concrete implementation is undertaken. In TLA$^+$, the models are referred to as **specifications**. The term specification is used henceforth in this document.

TLA$^+$has an IDE which is used for specifying systems, performing the model checking and proving theorems. Of the two, the model checker has a more widespread and common use case.

## 3.2 Specifying a system in TLA$^+$

Consider as an example, the specification of a bank. Such a system requires account holders and transactions. Account-holders, each, have a certain amount of money, we call it **balance** and a transaction is used to transfer money from one account to another.

Every specification in TLA$^+$is part of a **module**. This module contains the temporal logic as well as the various properties and invariants that the system needs to follow. The module is structured as shown in Figure 1.

The specification in TLA$^+$typically starts with imports for the helper modules which are used to model the system. These helper modules contain commonly used definitions which come in

MODULE *<Module name>*

<Imports>
<Variables>
<Initial predicate>
<State predicates>
<Next relation>

<Property definitions>

Figure 1: Structure of a specification in TLA$^+$

handy when specifying bigger systems. In our case, since the system performs arithmetic on the balance in the accounts, we need the **Integers** module.

The variables in the specification, define the state of the system. In our example, we need variables for the account holders, their accounts and the amount of money being transferred.

These variables are used in predicates which specify what the current state of the system is and based on a condition, what the next state should be. Such predicates can contain *primed* as well as *non-primed* occurrences of the variables. The non-primed occurrences specify the current value of the variable and the primed occurrences specify the value in the next state. The sequence of states that a system can go through is called a **behaviour**. Every specification has an **Init** clause which defines the state in which the system starts and a **Next** clause which defines how the system progresses. The Next clause can be a combination of multiple state clauses. In our example system, the states are **Withdraw** and **Deposit**.

The properties we need the model to conform to are called **assertions**. There are different types of assertions that can be specified in TLA$^+$, but the most common one is an **Invariant**. An invariant asserts a condition that should be true for every state of every possible behaviour. Another type of assertions that can be made using TLA$^+$are the properties asserting that something eventually happens. Such properties are called as *Liveness properties*.

The specification looks like Figure 2 after defining the components of the module.

After specifying the system, we can begin checking it against the target assertions. For example, the bank should not allow an account holder to withdraw more money than is present in his/her account (overdrafting). Such a property is an invariant on the account balance and should hold in every state. It can be written as follows:

$NoOverdrafts \triangleq \forall p \in people : acc[p] \geq 0$

This property states that for every person in the system, the account balance in every state should be greater than or equal to 0.

With this example, we see how easy it is to specify a system using TLA$^+$as well as how to check properties. In the further sections, we perform a thorough analysis of another verification problem as well as introduce the TLA$^+$theorem prover.

MODULE *bank*

VARIABLES $people, acc, sender, receiver, amount, pc$

$vars \triangleq \langle people, acc, sender, receiver, amount, pc \rangle$

$Init \triangleq people = \{"alice", "bob"\}$
$\quad \wedge\ acc = [p \in people \mapsto 5]$
$\quad \wedge\ sender = "alice"$
$\quad \wedge\ receiver = "bob"$
$\quad \wedge\ amount \in 1..6$
$\quad \wedge\ pc = "Withdraw"$

$Withdraw \triangleq pc = "Withdraw"$
$\quad \wedge\ acc' = [acc \text{ EXCEPT } ![sender] = acc[sender] - amount]$
$\quad \wedge\ pc' = "Deposit"$
$\quad \wedge\ \text{UNCHANGED } \langle people, sender, receiver, amount \rangle$

$Deposit \triangleq pc = "Deposit"$
$\quad \wedge\ acc' = [acc \text{ EXCEPT } ![receiver] = acc[receiver] + amount]$
$\quad \wedge\ pc' = "Done"$
$\quad \wedge\ \text{UNCHANGED } \langle people, sender, receiver, amount \rangle$

$Terminating \triangleq pc = "Done2"$
$\quad \wedge\ \text{UNCHANGED } vars$

$Next \triangleq Withdraw \vee\ Deposit \vee\ Terminating$

$Spec \triangleq Init \vee\ \Box[Next]_{vars}"$

Figure 2: Example specification of a bank system

# 4 Description of the verification problem

## 4.1 Introduction to Peterson's Lock

Peterson's lock appeared first in the journal *Information Processing Letters* in 1981. It was proposed by Gary L. Peterson in the paper called *Myths About the Mutual Exclusion Problem* [6]. Although there exist several variants of the algorithms that solve the problem of mutual exclusion for different settings, the classical algorithm only works for 2 processes and we confine our discussion to the same.

Peterson's algorithm uses 2 variables **flag** and **turn**. The flag is used to indicate the intent of entering the critical section. If a process $P_0$ sets its flag to true, the other process ($P_1$ ) can observe the flag and know about its intent. The turn is a shared variable. It contains the identifier of the process whose turn it is to enter the critical section.

```
flag :[false,false]
turn

For process P0:                              For process P1:

flag[0] = true;                              flag[1] = true;
turn = 1;                                    turn = 0;

while (flag[1] == true and turn ==1)         while (flag[0] == true and turn ==0)
{                                            {
//Busy Waiting                               //Busy Waiting
}                                            }

//Critical section                           //Critical section

flag[0] = false;                             flag[1] = false;
```

Figure 3: Peterson's Algorithm for 2 processes

According to Peterson's algorithm, the process $P_0$ can enter the critical section only when the flag for $P_1$ is false and turn is set to 0. The algorithm is shown in figure 3.

## 4.2 Properties satisfied by Peterson's algorithm

There are three important properties that the algorithm satisfies.

1. **Mutual exclusion** : Since the variable `turn` favours one of the processes and both the processes check the value of `turn` before entering the critical section, it is never true that both $P_0$ and $P_1$ will be in the critical section.
2. **Progress**: If only one of the processes wishes to enter the critical section, it will observe the flag of the other process to be false and can successfully do so. If both processes wish to enter the critical section, then due to the shared `turn` variable, at least one of the processes will be able to make progress.
3. **Bounded waiting**: The maximum number of times a process is bypassed by another, after it has set its `flag` to true is bounded by the number of processes running in the system. For the original Peterson's algorithm, this number is 1. If $P_0$ wishes to enter the critical section and $P_1$ bypasses it, as soon as $P_1$ exits the critical section, it will set its own `flag` to false and allow $P_0$ to enter the critical section.

## 4.3 Specifying Peterson's algorithm in TLA$^+$

The module for Peterson's algorithm consists of 3 variables, namely, flag, `turn` and `state`. The variable `state` is used to store the current step at which each of the processes is. It can take one

of the following values:

1. When the process is initialised, `state = "Start"`
2. When the process has changed the flag and requests its turn, `state = "RequestTurn"`
3. When the process is busy waiting, `state = "Waiting"`
4. When the process has entered the critical section, `state = "CriticalSection"`

The module is initialised as shown in the Figure 4

MODULE *peterson_lock*

EXTENDS $Integers, TLAPS$

VARIABLES $turn, state, flag$
$vars \triangleq \langle turn, state, flag \rangle$
$ProcSet \triangleq \{0, 1\}$
$States \triangleq \{"Start", "RequestTurn", "Waiting", "CriticalSection"\}$

$Not(i) \triangleq 1 - i$

Figure 4: Peterson's algotirhm initialisation. The set `States` containes all the possible states of the system. The operator *Not(i)* gives the process identifier of the other process running in the system

The state predicates that govern the behaviour of the system are defined in the following sections.

### 4.3.1 Initial State

The processes begin with their flags set to FALSE and the variable `turn` has an arbitrary value from 0 or 1.

$$
\begin{aligned}
&Init \triangleq flag = [i \in ProcSet \mapsto \text{FALSE}\ ] \\
&\quad \land\ Turn \in \{0, 1\} \\
&\quad \land\ Start = [i \in ProcSet \mapsto "Start"]
\end{aligned}
$$

### 4.3.2 Set Flags

The processes take the first step by setting their flags to TRUE to indicate their intent of entering the critical section. The variable `turn` remains unchanged.

$$
\begin{aligned}
&SetFlag(p) \triangleq state[p] = "Start" \\
&\quad \land\ flag' = [flag\ \text{EXCEPT}\ ![p] = \text{TRUE}\ ] \\
&\quad \land\ state' = [state\ \text{EXCEPT}\ ![p] = "RequestTurn"] \\
&\quad \land\ \text{UNCHANGED}\ \langle turn \rangle
\end{aligned}
$$

### 4.3.3 Set Turn

Both the processes can continue to set the value of the variable `turn`. However, based on the interleaving, the value written by one of the processes will be overwritten. In this state, the `flag` remains unchanged.

$$
\begin{aligned}
&SetTurn(p) \triangleq state[p] = \text{"}RequestTurn\text{"} \\
&\quad \land\ turn' = Not(p) \\
&\quad \land\ state' = [state \text{ EXCEPT } ![p] = \text{"}Waiting\text{"}] \\
&\quad \land \text{ UNCHANGED } \langle flag \rangle
\end{aligned}
$$

### 4.3.4 Enter Critical section

To enter the critical section, the processes check the `flag` and `turn` variables and only when the condition stands true, they enter. The condition is that the `flag` of the other process should be false or the `turn` should be set to the process identifier of the checking process. In this state, changing the `turn` and `flag` is not allowed.

$$
\begin{aligned}
&EnterCriticalSection(p) \triangleq state[p] = \text{"}Waiting\text{"} \\
&\quad \land\ (flag[Not(p)] = \text{FALSE} \ \lor\ turn = p) \\
&\quad \land\ state' = [state \text{ EXCEPT } ![p] = \text{"}CriticalSection\text{"}] \\
&\quad \land \text{ UNCHANGED } \langle turn, flag \rangle
\end{aligned}
$$

### 4.3.5 Exit Critical section

This state is used to clean up the flag variables. After the process exits the critical section, it sets its `flag` to `false` and changes its `state` variable to "Start".

$$
\begin{aligned}
&ExitCriticalSection(p) \triangleq state[p] = \text{"}CriticalSection\text{"} \\
&\quad \land\ flag' = [flag \text{ EXCEPT } ![p] = \text{FALSE }] \\
&\quad \land\ state' = [state \text{ EXCEPT } ![p] = \text{"}Start\text{"}] \\
&\quad \land \text{ UNCHANGED } \langle turn \rangle
\end{aligned}
$$

### 4.3.6 Next relation

The next relation is a logical disjunction of the state relations and bounds the temporal successors to be one of the specified states. It is written as follows:

$$
\begin{aligned}
&Next \triangleq \exists p \in ProcSet : \\
&\quad \lor\ setFlag(p) \\
&\quad \lor\ SetTurn(p) \\
&\quad \lor\ EnterCriticalSection(p) \\
&\quad \lor\ ExitCriticalSection(p)
\end{aligned}
$$

The spec for the system is a conjunction of the Initial state and the Temporal successors according to the Next relation, written as:

$Spec \triangleq Init \wedge \ \Box[Next]_{vars}$[1]

# 5 Model Checking Peterson's Algorithm with TLA$^+$

The property that we want the algorithm to satisfy is **Mutual exclusion**. Mutual exclusion dictates that for processes that tun concurrently and have access to the same shared resource do not access the resource at the same time. While the processes are accessing the shared resource, they are said to be in their **critical section**. We state mutual exclusion in TLA$^+$for our specification as follows:

$MutualExclusion \triangleq \neg(state[0] = "CriticalSection" \wedge \ state[1] = "CriticalSection")$

This predicate is supplied to the **TLC model checker** [13] in TLA$^+$which performs the state space exploration using the specification of the system and for every state, the model checker verifies that our property $MutualExclusion$ holds. The model checker then returns a state exploration graph. The results of the state exploration of Peterson's algorithm can be found in Appendix B.

# 6 The TLA$^+$ Proof System (TLAPS)

## 6.1 Introduction to Hierarchical proofs

TLA$^+$has several constructs that allow for writing mathematical proofs. Such proofs are not confined towards theorems based in pure mathematics, but also include proofs for properties of computer systems and algorithms. TLA$^+$supports writing hierarchically structured proofs. This method was introduced by Uri Leron [5]. Structured proofs break the initial theorem into smaller steps. Each step is a **proof obligation**. Every proof obligation requires a proof itself. This way, a hierarchy is developed until the obligations cannot be broken further i.e. we have reached a trivial case or the proof is **obvious**, i.e., it follows from an assumption which has been proven true.

## 6.2 Writing a simple proof in TLA$^+$

There are two sections to every proof. The first specifies the theorem and the second contains the proof statements. Let us consider a very simple theorem which states that the greater-than ($>$) relation is transitive. The theorem can be written in TLA$^+$in the following manner.

THEOREM $Transitive \triangleq$

[1] $\Box$ is a modal temporal operator. It is defined in the PTL module in TLA$^+$8. Here, the operator specifies that Next relation 4.3.6 should hold true in any state if the system starts from the initial state 4.3.1.

ASSUME NEW $x \in Nat$,
NEW $y \in Nat$,
NEW $z \in Nat$,
$x > y$,
$y > z$
PROVE $x > z$
PROOF
OBVIOUS

In this theorem, we start with the ASSUME construct. Theorems in TLA$^+$do not borrow definitions of variables. Thus, we specifically need to tell the proof system about the variables we are going to use as well as the domain of the values that the variables can take.

The proof did not require any expansion and can be written as an obvious statement. Such proofs are referred to as **Leaf proofs**. In our example, we did not have to break down the obligation because it is implied from the definition of the greater-than operator and the definition of *x, y, z* from the Naturals(*Nat*) module. These definitions are included when we import the *Naturals* module in our specification.

## 6.3 TLA$^+$Proof constructs

### 6.3.1 BY

BY is a general construct which is used whenever we borrow a definition from a sub-proof, an earlier proof statement of an assumption and use it to define our current obligation. It is typically only used in conjunction with other constructs.

### 6.3.2 SUFFICES

In an ordinary proof step, we make some assumptions and use them to write a proof statement. This is structured as ASSUME ... PROVE ... clause. SUFFICES reverses the role of these 2 clauses. SUFFICES allows us to state that a certain portion of the proof is sufficient when it counts towards proving our parent obligation.

### 6.3.3 CASE

In certain mathematical proofs, the obligations flow non-linearly i.e. at a certain step, one of the many decisions is picked. This is resolved using the CASE construct. The CASE construct evaluates a condition and then evaluates the definitions corresponding to the TRUE cases. The execution is non-exclusive, meaning, that multiple case clauses can be executed within a sub-proof.

### 6.3.4 USE

There are some definitions which, once proven, need to be made available to the remaining proof statements. This can be done either by using them with the BY ... DEF ... construct in every

step of the proof. But such a proof quickly becomes difficult to follow. It can be simplified by making the definitions available to all the proof steps by the USE ... DEF ... construct.

### 6.3.5 QED

The QED step asserts the main goal of the proof. It is placed at the end of every proof section. in the QED step, we collect and check all the definitions needed for the proof to be verified and cite them using the BY keyword.

## 6.4 Architecture of the TLA$^+$Proof system

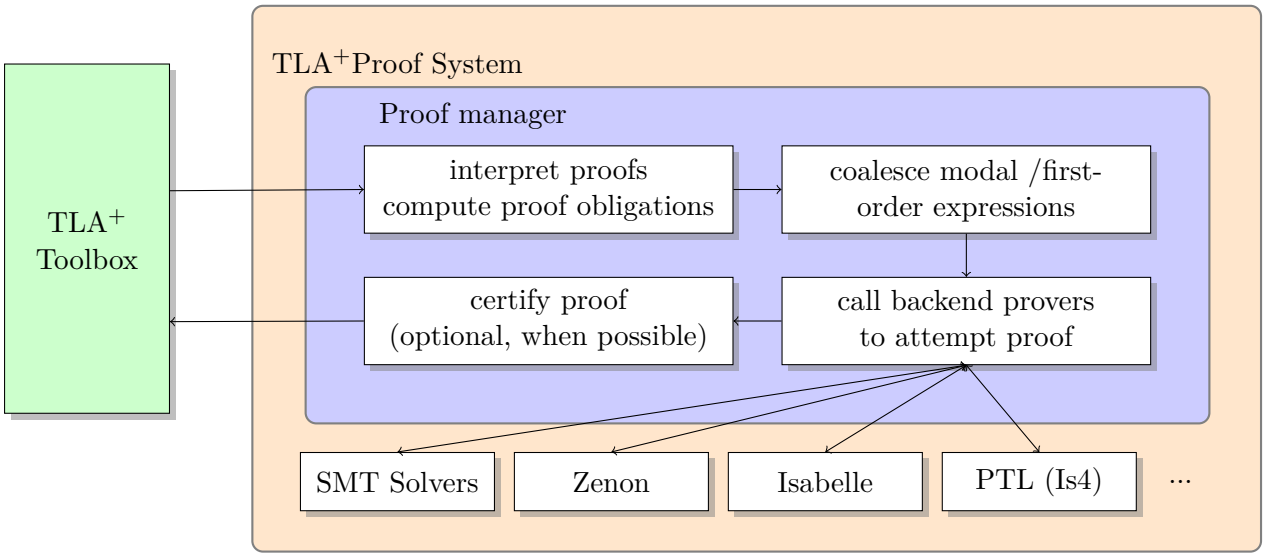


Figure 5: Architecture of the TLA+ Proof system [1]

The front end for communicating with the proof system is the **TLA$^+$toolbox**. Through the toolbox, we provide the proof system with module definitions, the theorems we want to prove as well as the proof scripts. The processing for any proof starts with the **interpretation step**. The proof manager parses the module specification and uses the definitions in the module to compute the **obligations** that it needs to expand to perform the further processing of the proof. These obligations are successively replaced with simpler formulas until no further expansion or replacement of an obligation is possible. The obligations are then collected and combined to form an exhaustive logical expression that is to be proven.

This logical expression is then transformed into the target languages that the theorem provers accept. The proof manager calls all the possible solvers to attempt the proof. In case the proving fails for all of them, the obligation that needs further explicit proof definition is reported back. If the obligations are all proven successfully by at least one of the solvers in the specified time limit, the result is reported back to the user. In case of an optional step which involves mechanically

checking the proof produced by a solver with another solver, a cross-check between solvers is performed. For example, a proof generated by the SMT solver could be checked by Isabelle.

# 7 Formal proof of Peterson's algorithm in TLA+

## 7.1 Structure of the proof

For Peterson's algorithm, we want to prove that a property (Mutual Exclusion) is always satisfied for every possible behaviour of the system. This proof is structured as a standard **Invariance proof**.

We begin by proving that mutual exclusion is satisfied by the initial state. This is a trivial step since none of the processes have made progress. After this, we attempt to prove that if the system enters a state in which mutual exclusion is satisfied, it leaves the state with a configuration of the system such that mutual exclusion is satisfied in the next state. This step is applied inductively to all the possible intermediate steps that the system takes. In the end state, the partial inductive proof, from each of the former states in the behaviour, is collected to prove that the system satisfies mutual exclusion.

## 7.2 Proof obligations in TLA$^+$

### 7.2.1 Invariants for the proof

In order to get around writing repetitive logical formulas, auxiliary definitions are provided as a part of module specification. For our proof, we combine the the following invariants with the AND ( $\wedge$ ) operator:

1. **Execution Invariant**: This invariant gives a logical formula which enforces logical value boundaries on the state variables (`flag` and `turn`). It asserts the following:
   a) For a process in a state other than the start state, the flag should be set to TRUE .
   b) If a process $P_0$ is in the critical section, then $P_1$ should not be in its critical section and vice versa.
   c) If a process does not have to wait to enter the critical section, the then variable `turn` should be set to its process identifier.

   This is written in logical form as follows:

$$
\begin{aligned}
&ExecutionInvariant \triangleq \forall i \in ProcSet : \\
&\qquad state[i] \in States \setminus \{"Start"\} \implies flag[i] \\
&\wedge\ state[i] \in \{"CriticalSection"\} \implies state[Not(i)] \notin \{"CriticalSection"\} \\
&\wedge\ state[Not(i)] \in \{"Waiting"\} \implies turn = i
\end{aligned}
$$

2. **TypeInvariant**: This invariant enforces the type and value constraints on the state variables i.e. state should always be one of the values of the State set, turn should always be 0 or 1 and flag for every process should be either TRUE or FALSE . This is written in

logical form as follows:

$$TypeInvariant \triangleq state \in [ProcSet \rightarrow States] \\ \wedge\ turn \in ProcSet \\ \wedge\ flag \in [ProcSet \rightarrow \{true, false\}]$$

## 7.3 Proof Theorem

For mutual exclusion, we intend to prove that at any given time, only one process is in the critical section. This can be specified as

$$MutualExclusion \triangleq \neg(state[0] = \text{"}CriticalSection\text{"} \wedge\ state[1] = \text{"}CriticalSection\text{"})$$

The theorem then uses the temporal **global operator**, written as $\Box$, to form the theorem.

$$Theorem \triangleq \Box MutualExclution$$

The **global** operator ( $\Box$ ) specifies that the property should hold in every state on the entire subsequent path in the behaviour. Since our system diverges into different behaviour paths from a single initial state, the global operator applies this constraint on all the possible sates and behaviours of the system.

## 7.4 Proof Steps

### 7.4.1 Base case for inductive steps

We begin by attempting the proof for the following logical formula. This evaluates to the invariant being TRUE in the initial state.

$$Init \implies Inv \qquad (<1>1)$$

This step is easy enough for the SMT solver to solve without us providing an intermediate proof step. It uses the definition of the Init state, the state variables as well as the Invariant (Inv) which is the logical AND of the Execution and Type invariants.

### 7.4.2 Inductive steps

The Inductive steps prove that for any state in which the invariants are true, applying the 'Next' clause leads the system into a state in which the invariants remain true. The top level proof obligation for this step is the following:

$$Inv \wedge\ [Next]_{vars} \implies Inv' \qquad (<1>2)$$

The obligation <1>2 is not trivial since it is a collection of all the possible state changes that the Next clause can make. This leads to us creating a new proof level. One of the ways we can prove this obligation is to assume that the Invariant 'Inv' and the 'Next' clause are proven already. This is written using the SUFFICES 6.3.2 construct. We let the solver expand these obligations with the assumption that the Invariants are enough to prove this step and then attempt the proof for the Invariant in the next state i.e. Inv'. This is written in TLA$^+$as follows:

$$\begin{array}{l}\text{SUFFICES ASSUME } Inv, Next \text{ PROVE } Inv' \\ \text{BY DEFS } ExecutionInvariant, TypeInvariant, Inv, Vars\end{array} \quad (<2>1)$$

It is possible to write this and not run into problems because the proofs for the *ExecutionInvariant* and *TypeInvariant* is written in the subsequent steps of the proof level and collected in the QED step

The next part of the proof consists of the proof definitions for the invariants. We begin by proving that TypeInvariant holds in the next state. This is written as follows:

$$\begin{array}{l}TypeInvariant' \\ \text{BY } <2>1 \\ \text{DEFS } Inv, TypeInvariant, Next, proc, Not, States, ProcSet, \\ SetFlag, SetTurn, EnterCriticalSection, ExitCriticalSection\end{array} \quad (<2>2)$$

This step of the proof is also easy for the SMT solver to expand using the definitions and then prove.

The other invariant that completes this proof level is the *ExecutionInvariant*. The proof for the execution invariant is quite involved. We start by choosing two random processes from our process set. This leads to two possible combinations. Either both the processes are the same, which is one part of the proof, or the processes are different, another part of the proof. These broken cases are easy for the solver to expand and prove on its own, so we just need to give it the definitions that it needs to expand. The proof for this step is written as follows:

$$ExecutionInvariant' \quad (<2>3)$$

$$\begin{array}{l}\text{SUFFICES ASSUME NEW } j \in ProcSet \text{PROVE } ExecutionInvariant!(j)' \\ \text{BY DEFS } ExecutionInvariant, ProcSet\end{array} \quad (<3>1)$$

The proof for the *ExecutionInvariant* now expects the case distinctions. This is done using the CASE 6.3.3 construct. For this, we pick a random process i from the process set and use the definitions of this process to evaluate the condition. It can be written as follows:

PICK $i \in ProcSet : proc(i)$
BY $< 2 > 1$
DEFS $Next, ProcSet, proc, SetFlag, SetTurn,$
$EnterCriticalSection, ExitCriticalSection$ (<3>2)

CASE $i = j$
BY $< 2 > 1, < 3 > 2$
DEFS $ExecutionInvariant, TypeInvariant, Inv, proc,$
$Not, ProcSet, SetFlag, SetTurn, EnterCriticalSection, ExitCriticalSection$ (<3>3)

CASE $i \neq j$
BY $< 2 > 1, < 3 > 2$
DEFS $ExecutionInvariant, TypeInvariant, Inv, proc,$
$Not, ProcSet, SetFlag, SetTurn, EnterCriticalSection, ExitCriticalSection$ (<3>4)

The QED step for the execution invariant uses step <3>3 and <3>4 to define the sub-proof. The step <2>2 and <2>3 are then used to define the proof for <1>2.

### 7.4.3 Implied Mutual Exclusion

This step combines the invariants and proves that in a state where the invariants hold, mutual exclusion holds also. This is easily expanded by the solver using the following definitions.

$Inv \implies MutualExclusion$
BY DEF $Inv, MutualExclusion, ProcSet, Not,$
$ExecutionInvariant, TypeInvariant$ (<1>3)

### 7.4.4 QED step for the Theorem obligation

In the last obligation of the proof, the results from the previous subproofs are combined using the PTL procedure 8. It combines the results from <1>1, <1>2, <1>3 as follows:

QED BY $< 1 > 1, < 1 > 2, < 1 > 3$, PTL 8
DEFS $MutualExclusion, Spec, ExecutionInvariant, TypeInvariant, Inv,$ (<1>4)
$proc, Not, ProcSet, SetFlag, SetTurn, EnterCriticalSection, ExitCriticalSection$

The PTL procedure introduces the notion of time and reasons with the predecessor and successor states. In concurrent algorithms, the notion of time plays a very important role. For an algorithm as simple as Peterson's lock, where the number of critical states is very few, the state exploration results in a significantly large number of distinct cases with multiple possible

interleavings. We need to use PTL for collecting the results of the intermediate proof obligations because for all the states, specified by the Next relation in our specification, there is no predefined temporal order in which they have to occur. TLA$^+$proof system has a dedicated backend solver for handling PTL logic. When we tell the proof system to use PTL, the obligations are handled by it, as in step <1>4.

# 8 Conclusion

The goal of this report has been to present how TLA$^+$can be used to specify computer systems as well as how simple and straightforward the process is. One of the major caveats in this regard is the steep initial learning curve that is required to be able to use the constructs provided by TLA$^+$effectively. Writing a simple system with only very few variables and states, as shown in Section 3, is very easy but as the system grows in complexity, the construction grows equally quickly both in size and complexity. It is very easy to overlook the edge cases while defining such a system and encounter problems like, invariants not being followed by the system or reaching an eventual deadlock.

Although TLA$^+$is very elegant and it becomes very easy to understand a specification once the initial learning of the system constructs is dealt with, there are certain aspects of the language that make it difficult to learn and for computer science students, researchers as well as engineers, to be able to follow along. One major contributor to this is the way we think about the problems in computer science. We are, more often than not, dealing with procedural languages in which the flow of control and data is more or less straightforward. With specification languages, the way of thinking changes and emphasis shifts from the functionality of the system to the mathematical definition of that functionality. This requires re-engineering our thinking towards what the system does rather than reasoning about how the system does it.

It would be remiss to not mention where TLA$^+$is significantly better when thinking about systems engineering and other similar practical applications. With a mathematical foundation, thinking about systems as well as checking them is raised one level above flow diagrams and textual specifications. It becomes very easy to check exactly what part of the system leads to faults and how the system reaches this state without having to deal with external factors like computer architectures, networking concepts and other dependencies. This is especially good for designing algorithms because any fundamental issue is easily identifiable before an implementation is undertaken.

Another aspect which makes TLA$^+$and similar specification languages superior is the ability to write theorems and proofs. There have been instances where mathematicians have accepted the fact that the proofs were too difficult to manually verify. One such documented instance is by R. Zahler in the Science journal [14]. TLA$^+$proves fruitful here by allowing for mechanical checking of proofs. the proof obligations as introduced in Section 6 are written and automatically checked by solvers. One advantage of doing this is that while presenting the proof, certain detailed sub-proofs can be hidden considering the level of detail that the audience expects. The fully checked proof and the proof definition still exists as a nested entity.

There are many more constructs and properties that the language offers e.g. Fairness specifications(weak and strong), liveliness checks, which this report has conveniently glossed over. However, these constructs offer even stronger capabilities for specifying systems and come in handy when modelling complex systems. For our problem, there was no need to use such constructs, hence the omission. Even without these, it is very easy to see how TLA$^+$is not just another language that is used to specify a system, but a tool which, if used in the right way is very powerful to solve some very finicky problems that computer science practitioners regularly face.

# Appendix A

## Constant Operators

$F(x_1, ..., x_n) \triangleq exp$ Defines $F$ to be an operator such that $F(e_1, ..., e_n)$ equals $exp$ with each identifier $x_k$ replaced by $e_k$.

### Sets

| | |
|---|---|
| $\{e_1, ...e_n\}$ | [Set containing elements $e_i$] |
| $\{x \in S : p\}$ | [Set containing elements x in $S$ satisfying p] |
| $\{e : x \in S\}$ | [Set of elements e such that x in $S$] |

### Functions

| | |
|---|---|
| $f[e]$ | [Function Application] |
| $[x \in S \mapsto e]$ | [Function $f$ such that $f[x] = e$ for $x \in S$ ] |
| $[f$ EXCEPT $![e_1] = e_2]$ | [Function $\widehat{f}$ equal to $f$ except $\widehat{f}[e_1] = e_2$. An element in $e_2$ equals $f[e_1]$] |

## Action Operators

| | |
|---|---|
| $e'$ | [The value of $e$ in the final state of a step] |
| $[A]_e$ | $[A \vee (e' = e)]$ |
| UNCHANGED $e$ | [e' = e] |

## PTL: Propositional Temporal Logic

Propositional Temporal Logic [2] introduces the notion of time in the logical evaluation of formulas. One easy way of doing to is to introduce a clock like variable which keeps track of the discrete time units. This, albeit correct, becomes difficult to manage when referring to specific instances in time. PTL thus introduces certain operators which, when evaluated at any given logical time instant, give information about what the Interpretation of the variables is. Some examples of such operators are:

1. $\circ A$: A holds at the time point immediately after the reference point. (*nexttime operator*).
2. $\Box A$: A holds at all time points after the reference point. (*always or henceforth operator*).
3. $\Diamond A$**:** There is a time point after the reference point at which A holds. *(sometime or eventually operator).*
4. *A atnext B*: A will hold at the next time point that B holds. ( *first time or atnext operator*).
5. *A until B*: A holds at all following time points up to a time point at which B holds. (*until operator*).

# Appendix B

## Peterson's algorithm model checking: State exploration results

1. **Total states found**: 36
2. **Distinct states found**: 20
3. **State counts for actions**:
   a) **Init**: States :- 2 ; Distinct :- 2
   b) **SetFlag**: States :- 12 ; Distinct :- 9
   c) **SetTurn**: States :- 12 ; Distinct :- 6
   d) **EnterCriticalSection**: States :- 4 ; Distinct :- 3
   e) **ExitCriticalSection**: States :- 6 ; Distinct :- 0

## Peterson's algorithm model Checking: State exploration Graph

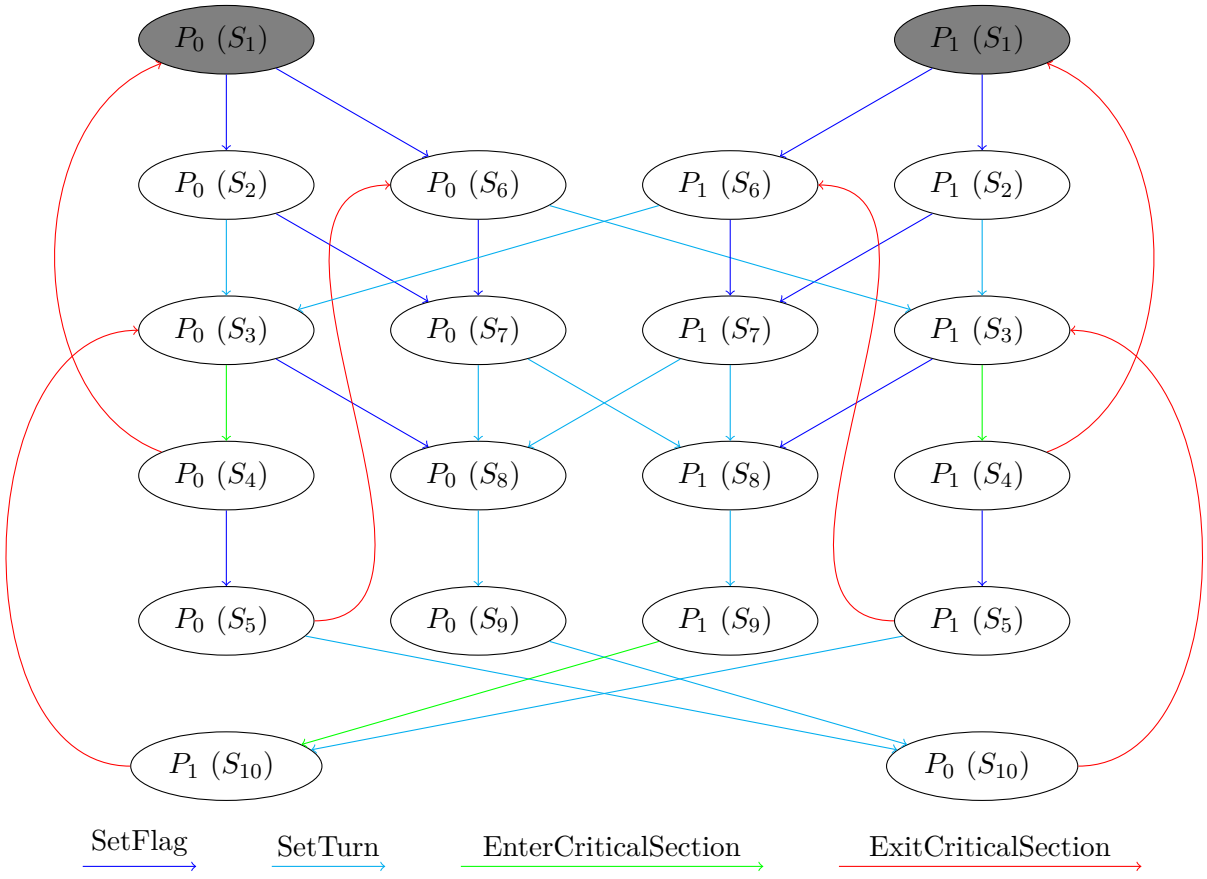


Figure 6: State exploration diagram for Peterson's algorithm

| State Identifier | State Clause |
|---|---|
| $P_0\ (S_1)$ | $flag = (0 :>$ FALSE $; 1 :>$ FALSE $)$<br>$\wedge\ state = (0 :>$ "$Start$"$; 1 :>$ "$Start$"$)$<br>$\wedge\ turn = 1$ |
| $P_0\ (S_2)$ | $flag = (0 :>$ TRUE $; 1 :>$ FALSE $)$<br>$\wedge\ state = (0 :>$ "$RequestTurn$"$; 1 :>$ "$Start$"$)$<br>$\wedge\ turn = 1$ |
| $P_0\ (S_3)$ | $flag = (0 :>$ TRUE $; 1 :>$ FALSE $)$<br>$\wedge\ state = (0 :>$ "$Waiting$"$; 1 :>$ "$Start$"$)$<br>$\wedge\ turn = 1$ |
| $P_0\ (S_4)$ | $flag = (0 :>$ TRUE $; 1 :>$ FALSE $)$<br>$\wedge\ state = (0 :>$ "$CriticalSection$"$; 1 :>$ "$Start$"$)$<br>$\wedge\ turn = 1$ |
| $P_0\ (S_5)$ | $flag = (0 :>$ TRUE $; 1 :>$ TRUE $)$<br>$\wedge\ state = (0 :>$ "$CriticalSection$"$; 1 :>$ "$RequestTurn$"$)$<br>$\wedge\ turn = 1$ |
| $P_0\ (S_6)$ | $flag = (0 :>$ FALSE $; 1 :>$ TRUE $)$<br>$\wedge\ state = (0 :>$ "$Start$"$; 1 :>$ "$RequestTurn$"$)$<br>$\wedge\ turn = 1$ |
| $P_0\ (S_7)$ | $\wedge\ flag = (0 :>$ TRUE $; 1 :>$ TRUE $)$<br>$\wedge\ state = (0 :>$ "$RequestTurn$"$; 1 :>$ "$RequestTurn$"$)$<br>$\wedge\ turn = 1$ |
| $P_0\ (S_8)$ | $\wedge\ flag = (0 :>$ TRUE $; 1 :>$ TRUE $)$<br>$\wedge\ state = (0 :>$ "$Waiting$"$; 1 :>$ "$RequestTurn$"$)$<br>$\wedge\ turn = 1$ |
| $P_0\ (S_9)$ | $\wedge\ flag = (0 :>$ TRUE $; 1 :>$ TRUE $)$<br>$\wedge\ state = (0 :>$ "$Waiting$"$; 1 :>$ "$Waiting$"$)$<br>$\wedge\ turn = 0$ |
| $P_0\ (S_{10})$ | $\wedge\ flag = (0 :>$ TRUE $; 1 :>$ TRUE $)$<br>$\wedge\ state = (0 :>$ "$CriticalSection$"$; 1 :>$ "$Waiting$"$)$<br>$\wedge\ turn = 0$ |
| $P_1\ (S_1)$ | $flag = (0 :>$ FALSE $; 1 :>$ FALSE $)$<br>$\wedge\ state = (0 :>$ "$Start$"$; 1 :>$ "$Start$"$)$<br>$\wedge\ turn = 0$ |
| $P_1\ (S_2)$ | $flag = (0 :>$ FALSE $; 1 :>$ TRUE $)$<br>$\wedge\ state = (0 :>$ "$Start$"$; 1 :>$ "$RequestTurn$"$)$<br>$\wedge\ turn = 0$ |
| $P_1\ (S_3)$ | $flag = (0 :>$ FALSE $; 1 :>$ TRUE $)$<br>$\wedge\ state = (0 :>$ "$Start$"$; 1 :>$ "$Waiting$"$)$<br>$\wedge\ turn = 0$ |
| $P_1\ (S_4)$ | $flag = (0 :>$ FALSE $; 1 :>$ TRUE $)$<br>$\wedge\ state = (0 :>$ "$Start$"$; 1 :>$ "$CriticalSection$"$)$<br>$\wedge\ turn = 0$ |

| State Identifier | State Clause |
|---|---|
| $P_1$ $(S_5)$ | $flag = (0 :>$ TRUE $; 1 :>$ TRUE $)$<br>$\wedge\ state = (0 :>$ "$RequestTurn$"$; 1 :>$ "$CriticalSection$"$)$<br>$\wedge\ turn = 0$ |
| $P_1$ $(S_6)$ | $flag = (0 :>$ TRUE $; 1 :>$ FALSE $)$<br>$\wedge\ state = (0 :>$ "$RequestTurn$"$; 1 :>$ "$Start$"$)$<br>$\wedge\ turn = 0$ |
| $P_1$ $(S_7)$ | $\wedge\ flag = (0 :>$ TRUE $; 1 :>$ TRUE $)$<br>$\wedge\ state = (0 :>$ "$RequestTurn$"$; 1 :>$ "$RequestTurn$"$)$<br>$\wedge\ turn = 0$ |
| $P_1$ $(S_8)$ | $\wedge\ flag = (0 :>$ TRUE $; 1 :>$ TRUE $)$<br>$\wedge\ state = (0 :>$ "$RequestTurn$"$; 1 :>$ "$Waiting$"$)$<br>$\wedge\ turn = 0$ |
| $P_1$ $(S_9)$ | $\wedge\ flag = (0 :>$ TRUE $; 1 :>$ TRUE $)$<br>$\wedge\ state = (0 :>$ "$Waiting$"$; 1 :>$ "$Waiting$"$)$<br>$\wedge\ turn = 1$ |
| $P_1$ $(S_{10})$ | $\wedge\ flag = (0 :>$ TRUE $; 1 :>$ TRUE $)$<br>$\wedge\ state = (0 :>$ "$Waiting$"$; 1 :>$ "$CriticalSection$"$)$<br>$\wedge\ turn = 1$ |

Table 1: State legend for the state exploration diagram in Figure 6

# Appendix C

## TLA$^+$specification for Peterson's algorithm.

MODULE *peterson_lock*

EXTENDS $Integers, TLAPS$

VARIABLES $turn, state, flag$

$vars \triangleq \langle turn, state, flag \rangle$

$ProcSet \triangleq \{0, 1\}$

$States \triangleq \{"Start", "RequestTurn", "Waiting", "CriticalSection"\}$

$Not(i) \triangleq 1 - i$

$Init \triangleq flag = [i \in ProcSet \mapsto \text{FALSE}\ ]$
$\quad \wedge\ Turn \in \{0, 1\}$
$\quad \wedge\ Start = [i \in ProcSet \mapsto "Start"]$

$SetFlag(p) \triangleq state[p] = "Start"$
$\quad \wedge\ flag' = [flag\ \text{EXCEPT}\ ![p] = \text{TRUE}\ ]$
$\quad \wedge\ state' = [state\ \text{EXCEPT}\ ![p] = "RequestTurn"]$
$\quad \wedge\ \text{UNCHANGED}\ \langle turn \rangle$

$SetTurn(p) \triangleq state[p] = "RequestTurn"$
$\quad \wedge\ turn' = Not(p)$
$\quad \wedge\ state' = [state\ \text{EXCEPT}\ ![p] = "Waiting"]$
$\quad \wedge\ \text{UNCHANGED}\ \langle flag \rangle$

$EnterCriticalSection(p) \triangleq state[p] = "Waiting"$
$\quad \wedge\ (flag[Not(p)] = \text{FALSE}\ \vee\ turn = p)$
$\quad \wedge\ state' = [state\ \text{EXCEPT}\ ![p] = "CriticalSection"]$
$\quad \wedge\ \text{UNCHANGED}\ \langle turn, flag \rangle$

$ExitCriticalSection(p) \triangleq state[p] = "CriticalSection""$
$\quad \wedge\ flag' = [flag\ \text{EXCEPT}\ ![p] = \text{FALSE}\ ]$
$\quad \wedge\ state' = [state\ \text{EXCEPT}\ ![p] = "Start"]$
$\quad \wedge\ \text{UNCHANGED}\ \langle turn \rangle$

$Next \triangleq \exists p \in ProcSet :$
$\quad \vee\ setFlag(p)$
$\quad \vee\ SetTurn(p)$

$\lor\ EnterCriticalSection(p)$
$\lor\ ExitCriticalSection(p)$

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

$proc(self) \triangleq$
$\lor\ setFlag(self)$
$\lor\ SetTurn(self)$
$\lor\ EnterCriticalSection(self)$
$\lor\ ExitCriticalSection(self)$

$ExecutionInvariant \triangleq \forall i \in ProcSet : $"
$state[i] \in States \setminus \{"Start"\} \implies flag[i]$
$\land\ state[i] \in \{"CriticalSection"\} \implies state[Not(i)] \notin \{"CriticalSection"\}$
$\land\ state[Not(i)] \in \{"Waiting"\} \implies turn = i$

$TypeInvariant \triangleq state \in [ProcSet \to States]$
$\land\ turn \in ProcSet$
$\land\ flag \in [ProcSet \to \{true, false\}]$

$MutualExclusion \triangleq \neg(state[0] = "CriticalSection" \land\ state[1] = "CriticalSection")$

$Inv \triangleq ExecutionInvariant \land\ TypeInvariant$

---

## TLA$^+$proof specification for mutual exclusion

MODULE *peterson_lock*

...TLA$^+$spec for peterson's lock 8...

THEOREM $Spec \implies \Box MutualExclusion$
PROOF
$\langle 1\rangle 1. Init \implies Inv$
BY DEFS $Init, Inv, TypeInvariant, ExecutionInvariant, vars, States, ProcSet$

$\langle 1\rangle 2. Inv \land\ [Next]_{vars} \implies Inv'$
BY DEFS $Init, Inv, TypeInvariant, ExecutionInvariant, vars, States, ProcSet$

$\langle 2\rangle 1$. SUFFICES ASSUME $Inv, Next$ PROVE $Inv'$
BY DEFS $ExecutionInvariant, TypeInvariant, Inv, vars$

$\langle 2\rangle 2. TypeInvariant'$
BY $\langle 2\rangle 1$
DEFS $Inv, TypeInvariant, Next, proc, Not, States, ProcSet,$

$SetFlag, SetTurn, EnterCriticalSection, ExitCriticalSection$

$\langle 2 \rangle 3. ExecutionInvariant'$

$\langle 3 \rangle 1.$ SUFFICES ASSUME NEW $j \in ProcSet$ PROVE $ExecutionInvariant!(j)'$
BY DEFS $ExecutionInvariant, ProcSet$

$\langle 3 \rangle 2.$ PICK $i \in ProcSet : proc(i)$
BY $\langle 2 \rangle 1$ DEFS $Next, ProcSet, proc, SetFlag, SetTurn,$
$EnterCriticalSection, ExitCriticalSection$

$\langle 3 \rangle 3.$ CASE $i = j$
BY $\langle 2 \rangle 1, \langle 3 \rangle 2$ DEFS $ExecutionInvariant, TypeInvariant, Inv, proc,$
$Not, ProcSet, SetFlag, SetTurn, EnterCriticalSection, ExitCriticalSection$

$\langle 3 \rangle 3.$ CASE $i \neq j$
BY $\langle 2 \rangle 1, \langle 3 \rangle 2$ DEFS $ExecutionInvariant, TypeInvariant, Inv, proc, Not, ProcSet,$
$SetFlag, SetTurn, EnterCriticalSection, ExitCriticalSection$

$\langle 3 \rangle.$ QED BY $\langle 3 \rangle 3, \langle 3 \rangle 4$

$\langle 2 \rangle 4.$ QED BY $\langle 2 \rangle 2, \langle 2 \rangle 3$ DEF $Inv$

$\langle 1 \rangle 3. Inv \implies MutualExclusion$
BY DEFS $Inv, MutualExclusion, ProcSet, Not, ExecutionInvariant, TypeInvariant$

$\langle 1 \rangle 4.$ QED BY $\langle 1 \rangle 1, \langle 1 \rangle 2, \langle 1 \rangle 3, \langle 1 \rangle 4,$ PTL
DEFS $MutualExclusion, Spec, ExecutionInvariant, TypeInvariant, Inv,$
$proc, Not, ProcSet, SetFlag, SetTurn, EnterCriticalSection, ExitCriticalSection$

---

## References


[1] D. Cousineau, S. Merz (2010): *The TLA$^+$ Proof System*. https://tla.msr-inria.inria.fr/tlaps/doc/IFM2010/tutorial.pdf. [Online; accessed 04-January-2021].

[2] Fred Kröger (1987): *Temporal Logic of Programs*. *EATCS Monographs on Theoretical Computer Science* 8, Springer.

[3] Leslie Lamport (1994): *The Temporal Logic of Actions*. *ACM Trans. Program. Lang. Syst.* 16(3), pp. 872–923, doi:10.1145/177492.177726. Available at https://doi.org/10.1145/177492.177726.

[4] Leslie Lamport (2012): *How to write a 21st century proof*. *Journal of Fixed Point Theory and Applications* 11, doi:10.1007/s11784-012-0071-6.

[5] Uri Leron (1983): *Structuring Mathematical Proofs*. *The American Mathematical Monthly* 90(3), pp. 174–185. Available at http://www.jstor.org/stable/2975544.

[6] Gary L. Peterson (1981): *Myths About the Mutual Exclusion Problem*. *Inf. Process. Lett.* 12(3), pp. 115–116, doi:10.1016/0020-0190(81)90106-X. Available at https://doi.org/10.1016/0020-0190(81)90106-X.

[7] NASA Ames Research Center. Robust Software Engineering Group (2020): *JPF..the swiss army knife of java verification*. http://javapathfinder.sourceforge.net/. [Online; accessed 10-November-2020].

[8] Makarius Wenzel, Lawrence C. Paulson & Tobias Nipkow (2008): *The Isabelle Framework*. In Otmane Aït Mohamed, César A. Muñoz & Sofiène Tahar, editors: *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, *Lecture Notes in Computer Science* 5170, Springer, pp. 33–38, doi:10.1007/978-3-540-71067-7_7. Available at https://doi.org/10.1007/978-3-540-71067-7_7.

[9] Wikipedia contributors (2020): *BLAST model checker — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=BLAST_model_checker&oldid=983467388. [Online; accessed 10-November-2020].

[10] Wikipedia contributors (2020): *Mutual exclusion — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Mutual_exclusion&oldid=988511871. [Online; accessed 20-November-2020].

[11] Wikipedia contributors (2020): *Promela — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Promela&oldid=967605570. [Online; accessed 10-November-2020].

[12] Pierre Wolper (1981): *Temporal Logic Can Be More Expressive*. In: *22nd Annual Symposium on Foundations of Computer Science, Nashville, Tennessee, USA, 28-30 October 1981*, IEEE Computer Society, pp. 340–348, doi:10.1109/SFCS.1981.44. Available at https://doi.org/10.1109/SFCS.1981.44.

[13] Yuan Yu, Panagiotis Manolios & Leslie Lamport (1999): *Model Checking TLA$^+$ Specifications*. In Laurence Pierre & Thomas Kropf, editors: *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, *Lecture Notes in Computer Science* 1703, Springer, pp. 54–66, doi:10.1007/3-540-48153-2_6. Available at https://doi.org/10.1007/3-540-48153-2_6.

[14] RAPHAE ZAHLER (1983): *Errors in Mathematical Proofs*. *Science* 193(4248), pp. 98–98. Available at https://science.sciencemag.org/content/193/4248/98.1.