Proving Safety Properties

Leslie Lamport

18 May 2019
Minor correction: 16 July 2019

This note explains how to use the TLC model checker, the TLA$^+$ Toolbox, and the TLAPS proof system to write rigorous informal proofs of safety properties of specifications. Learning to write such proofs is the first step in learning to write formal, machine-checked proofs of these properties. An elementary knowledge of TLA$^+$, the Toolbox, and TLC is assumed.

Text colored like this in the table of contents and like this in the body of the paper are clickable links.

Contents

Part I
Invariance

1 Introduction

The most common type of property one checks that a specification `Spec` satisfies is *invariance*—the assertion that some state formula `I` is true of all states in every behavior satisfying `Spec`. The statement that `Spec` satisfies this property is expressed mathematically by this theorem:

```
THEOREM  Spec => []I
```

When this theorem is true, we say that `I` is an *invariant* of `Spec`. Part I of this document explains how to prove such a theorem. Part II explains how to prove more general safety properties asserting that a specification implements a higher-level specification.

One can write completely formal proofs in TLA$^+$ that can be checked by the TLAPS proof system. This note does not explain how to do that. It does explain how to use the TLC model checker to help you find a proof, and how to use the TLA$^+$ Toolbox and the TLAPS proof system to help you decompose the complete proof into simpler steps. While the resulting proof is not completely formal, it is much more trustworthy than the prose proofs typically written by mathematicians and computer scientists. It is also the first step in writing a completely formal proof and checking it with TLAPS.

The explanation in Part I of how to prove invariance properties is based on a single simple example: a mutual exclusion protocol.

2 Mutual Exclusion

The problem of achieving mutual exclusion was introduced by Edsger Dijkstra in a classic 1965 paper [1]. It assumes a set of processes, each containing a section of code called its critical section. The problem is to devise an algorithm in which processes can access their critical sections, but no two processes can be executing their critical sections at the same time.

We consider only the case in which there are two processes, which we call processes 0 and 1. We will write our algorithms in PlusCal, but the PlusCal code is explained in terms of its TLA$^+$ translation. The two processes will be described by a process declaration

```
process (proc \in {0,1}) { ... }
```

which produces the next-state action defined by

```
Next  ==  \E i \in {0, 1} : proc(i)
```

where the definition of `proc` is determined by the body of the process declaration. The process's critical section is represented in that body by the statement

```
cs: skip ;
```

A process is considered to be in its critical section when control in that process is at the point labeled `cs`, where the process is about to execute this `skip` statement. Executing the `skip` statement does nothing except change the process's control state to the labeled statement immediately following it.

The TLA$^+$ translation of the algorithm introduces a variable `pc` whose value describes the control state of all the processes. The value of `pc` in all reachable states of the algorithm is a function with domain `{0,1}`, and `pc[i]` is equal to the string `"cs"` when control in process `i` is at label `cs`. The invariant `Mutex` that a mutual exclusion algorithm must satisfy is defined by:

```
Mutex  ==  ~((pc[0] = "cs") /\ (pc[1] = "cs"))
```

This invariance property is trivially satisfied by an algorithm in which no process can never reach label `cs`. In addition to the invariance of `Mutex`, a mutual exclusion algorithm must also satisfy some liveness condition asserting that critical sections will be executed if processes want to execute them. However, since we're interested here only in invariance, we'll ignore liveness.

It might seem that we haven't adequately modeled the mutual exclusion problem because the purpose of mutual exclusion is to make critical sections that execute multiple actions act like a single atomic action. By assuming that a critical section is an atomic `skip` action, aren't we making the mutual exclusion problem trivial? The answer is that we're not assuming that execution of the critical section is a single atomic action. Considering only safety, the statement `cs: skip;` is completely equivalent to:

```
cs: either skip
        or goto cs ;
```

The extra steps performed by choosing the `or` clause and repeating the statement are stuttering steps, which are allowed by any algorithm. If we add extra variables to any algorithm and replace any `skip` statement in that algorithm by a sequence of unlabeled statements that can modify only those extra variables, the resulting algorithm implements the original algorithm. This means that if we rename the specification `Spec` produced by the TLA$^+$ translation of the original algorithm to `A` and rename the corresponding TLA$^+$ spec of the second algorithm to `B`, then

```
THEOREM  B => A
```

would be true. Steps allowed by `B` that change an extra variable are stuttering steps allowed by `A`. It would be awkward to write algorithm `B` in PlusCal, since the extra steps it takes would have to leave `pc` unchanged. However, it would be easy to replace the critical section by PlusCal code to obtain an algorithm that implements `A` under a refinement mapping that maps `pc[i][`l`]` to `"cs"` for every label l in the critical section.

3 The Simple Protocol

There is an extremely simple protocol for achieving mutual exclusion for processes 0 and 1. Each process `i` in `{0, 1}` executes the following code, where `flag` is a Boolean-valued function with domain `{0,1}`.

```
s1: flag[i] := TRUE ;
s2: await flag[1-i] = FALSE ;
cs: skip;
```

The `await` statement can be executed only when the condition `flag[1-i] = FALSE` is true. Note that `1-i` is the process other than `i`. We call this a *protocol* because it's a code snippet that can be used in an algorithm, but it isn't a complete algorithm.

The following simple argument shows that this protocol ensures mutual exclusion. If both processes were at label `cs`, then the process `i` that reached `cs` last must have executed statement `s2` when process `1-i` was at `cs`, which is impossible because `flag[1-i]` equaled `TRUE` at that time.

This kind of proof, based on reasoning about the order in which things happened, is what I call a *behavioral* proof. This particular behavioral proof is quite convincing, and it is in fact correct. However, I and others have found that behavioral proofs don't work well for more complicated algorithms. It's easy to write convincing behavioral proofs of incorrect algorithms. Here, you'll learn a better way to reason about invariance.

```
---------------------- MODULE SimpleMutex ----------------------
EXTENDS Integers
(*****
 --algorithm SimpleMutex {
      variables flag = [i \in {0, 1} |-> FALSE] ;
      process (proc \in {0,1}) {
        s1: while (TRUE) {
              flag[self] := TRUE ;
        s2:   await flag[1-self] = FALSE ;
        cs:   skip ;
        s3:   flag[self] := FALSE           }     }     }
*****)
===================================================================
```

Figure 1: The Simple Mutex Algorithm

To reason rigorously about the simple protocol, we need to turn it into a complete algorithm. This is done in the module of Figure 1. Create a spec named *SimpleMutex* in the Toolbox, copying the body of the module from the figure and pasting it into the Toolbox's editor. Note that in the body of a `process` declaration, `self` is the name of the process—in this case, 0 or 1.

Run the PlusCal translator to put the definition of `Spec`, the algorithm's specification, into the module. As usual, `Spec` is defined to equal `Init /\ [][Next]_vars`. (Remember that `[Next]_vars` equals `Next \/ (vars'=vars)`, so it's true on a step $\langle s, t \rangle$ if and only if `Next` is true on $\langle s, t \rangle$ or the expression `vars` has the same value in state s and state t.) Observe that, for every label in the code, there is a subaction of the next-state action `Next` that describes the execution of the piece of code from that label to the next label. For example, corresponding to the label `s3` is this action:

```
s3(self) == /\ pc[self] = "s3"
            /\ flag' = [flag EXCEPT ![self] = FALSE]
            /\ pc' = [pc EXCEPT ![self] = "s1"]
```

that is enabled when `pc[self]` equals `"s3"`, sets `flag[self]` to `FALSE` and sets `pc[self]` to `"s1"`.

We want to prove that `Mutex` is an invariant of `Spec`. It's a lot easier to prove something if it's true, so we should first let TLC check that `Mutex` is indeed an invariant of `Spec`. Add our definition

```
Mutex  ==  ~((pc[0] = "cs") /\ (pc[1] = "cs"))
```

to the module and have TLC check its invariance. The first thing TLC will report is that the algorithm deadlocks. Its error trace shows that both processes can wind up at the control point `s2` and be stuck there forever. This protocol is used in a number of mutual exclusion algorithms, including Dijkstra's original solution. Algorithms that use the protocol avoid this deadlock situation by having one of the processes give up and set its flag to `FALSE`, letting the other one proceed. But now, we just want to prove that the protocol ensures mutual exclusion and we don't care that it can deadlock. So, tell TLC to ignore deadlock and check invariance again. It finds that `Mutex` is an invariant of `Spec`. Moreover, it checks the complete algorithm, not just a small model of it, so TLC has actually checked that `Spec => []Mutex` is true for all possible behaviors. There's no need to prove the formula. But since this is about proving invariance, not about the protocol, we will prove that `Mutex` is an invariant of `Spec`.

4 How to Prove Invariance

4.1 What is a Proof?

We write a proof when TLC cannot check an algorithm on a large enough model to give us enough confidence that the algorithm is correct. "Proof" can mean anything from the prose proofs commonly used by mathematicians to completely formal, machine-checked proofs that can be written in TLA$^+$ and checked with the TLAPS proof system. Prose proofs are unreliable and can easily overlook the corner cases that often cause errors in algorithms, so they don't provide much confidence. A machine-checked proof is extremely reliable, but the high degree of confidence it provides may not be worth the effort required to write it. An actual implementation can have errors even if the high-level algorithm is correct, and time spent writing a machine-checked proof of the algorithm might be better spent trying to eliminate those other errors. With TLA$^+$ you can write proofs that are more reliable than the ones mathematicians write but require less work than completely machine-checked proofs.

TLA$^+$ proofs are hierarchically structured. A proof is either a leaf proof or else a sequence of statements, each with its proof. A leaf proof is a list of facts from which TLAPS can deduce the stated result. We write TLA$^+$ proofs in a top-down fashion. If TLAPS cannot deduce a statement from hypotheses and already proved facts, we write its proof as a sequence of simpler steps and then write the proof of each of those steps (in any order). It can be a lot of work because it may require multiple levels of steps to get

TLAPS to prove something that seems obvious to us. This is especially likely for a step whose proof requires reasoning about high-level data structures such as graphs.

Instead of writing a complete proof, we can stop this process of hierarchical decomposition when we get to a step that we are confident is true. If TLAPS can't prove it with a leaf proof, we omit the TLA$^+$ proof of the step and instead write in a comment an informal prose proof of why the step is true—the way a mathematician would. The simpler the step is, the more confidence we can have in a prose proof. The more levels deep we go, the simpler the steps are and the more confidence we can have in the correctness of our proof. This provides a tradeoff between the effort of writing the proof and the confidence it provides.

Like other aspects of TLA$^+$, these hierarchical proofs will seem strange when you first see them. But you'll see that they are the natural way to write proofs of algorithms. The higher levels of the hierarchy are determined by the structure of the results we are proving. In fact, most of the higher level decomposition can be performed for us by Toolbox commands. Correctness proofs of algorithms differ from mathematical proofs because they involve checking many details and usually involve only very simple math. Hierarchical structure is the only way to make sure we check all the details.

4.2 The Structure of an Invariance Proof

We want to prove the theorem `Spec => []Mutex`, where `Spec` equals `Init /\ [][Next]_vars`. The best way to do this apparently dates back to the 1940s, but its current use was initiated by Robert Floyd in 1967 [2]. To prove that a state formula `Inv` is true in all states of a behavior, by induction it suffices to prove:

1. `Inv` is true in the initial state.

2. If `Inv` is true in any state of the behavior, then it is true in the next state of the behavior.

To show that 1 is true for any behavior, it suffices to prove that the initial predicate `Init` implies `Inv`. To show that 2 is true for any behavior, it suffices to show that for any step (pair of states) $\langle s, t \rangle$ satisfying `[Next]_vars`, if `Inv` is true in state s then it's true in state t. This leads to the following proof rule, where the two theorems of the hypothesis imply 1 and 2.

Inductive Invariance Rule

```
        THEOREM  Init => Inv
  and   THEOREM  Inv /\ [Next]_vars => Inv'
imply   THEOREM  Init /\ [][Next]_vars => []Inv
```

Remember that `THEOREM F` asserts that formula `F` is true for all behaviors, where a behavior is *any* sequence of states. A formula `Inv` satisfying the hypotheses of this rule is called an *inductive invariant* of `Spec`.

It's easy to see that for our example, the initial condition `Init` implies that no process is in its critical section, so the first hypothesis of the rule is satisfied for `Inv` equal to `Mutex`. But the second hypothesis is not satisfied. As a counterexample, let s be a state with `pc[0]="cs"`, `flag[0]=FALSE`, and `pc[1]="s2"`; and let t be the state obtained from s by executing process 1's statement $s2$. The formula `Mutex /\ [Next]_vars` is true on the step $\langle s, t \rangle$ because `Mutex` is true in state s and `Next` is true on the step, but `Mutex'` is not true on the step because `Mutex` is false in state t.

Of course this state s is not a reachable state of the algorithm, since process 0 cannot be at `cs` with `flag[0]` equal to `FALSE`. So, it's tempting to say that we don't have to worry about such a state s. Before going down that path, let me relate a little experience I had. Sometime in the 1980s, I was visited by a computer scientist who had recently published a concurrent algorithm. I asked him to show me the algorithm and explain his correctness proof. He began to sketch a formula he said was an inductive invariant. At some point I stopped him and pointed out a state satisfying his formula and a step of the algorithm from that state that made the formula false. He said we didn't have to worry about that state, and gave a behavioral argument that the state wasn't reachable. I explained to him that for a formula `Inv` to be an inductive invariant, a step of the algorithm from *any* state satisfying `Inv` had to leave `Inv` true. We then tried to find an inductive invariant for his algorithm. This led us to discover that the "unreachable" state I had found was indeed reachable, and his claimed invariant was incorrect. (I don't remember if the algorithm was incorrect or only its proof was.)

To write an inductive invariance proof, we need to find an inductive invariant `Inv`. If that inductive invariant satisfies

```
THEOREM  Inv => Mutex
```

then it proves that `Mutex` is an invariant of `Spec`. This is because the theorem states that `Inv => Mutex` is true for all states, so if `Inv` is true for all states in a behavior then `Mutex` is true for all states in that behavior. Hence `Spec => []Inv` implies `Spec => []Mutex`.

```
THEOREM  Spec => []Mutex
 <1>1. Init => Inv
 <1>2. Inv /\ [Next]_vars => Inv'
 <1>3. Inv => Mutex
 <1>4. QED
```

Figure 2: The high-level proof structure.

These observations lead to the high-level proof structure of Figure 2. Every non-leaf proof ends with a `QED` step that asserts the goal we are trying to prove, in this case `Spec => []Mutex`. This goal follows from the three steps `<1>1` – `<1>3` and the fact that `Spec` is defined to equal `Init /\ [][Next]_vars`. Thus, we expect the `QED` step to have a leaf proof telling TLAPS to use those three steps and the definition of `Spec`. The step and this leaf proof are written as:[1]

```
<1>4. QED
  BY <1>1, <1>2, <1>3 DEF Spec
```

However, because the `QED` step involves temporal-logic reasoning, we have to tell TLAPS to use a special back-end prover called PTL (for Propositional Temporal Logic), which we do by changing the `QED` step's proof to

```
BY <1>1, <1>2, <1>3, PTL DEF Spec
```

We also have to import the special `TLAPS` module in which the symbol `PTL` is defined, which we can do either by putting it in an `EXTENDS` statement at the beginning of the module or by adding the statement

```
INSTANCE TLAPS
```

anywhere before the theorem.[2] Now get TLAPS to check the proof of step `<1>4` by clicking anywhere in the step or its proof and running the Prove Step or Module command. (You can do that by either right clicking and selecting the command or typing control+g control+g .) The Toolbox colors the step green to indicate that TLAPS has successfully checked the proof.

Even when not writing a proof to be completely checked by TLAPS, I recommend having TLAPS check all the `QED` steps. Usually, a proof's goal should be a simple consequence of its steps, so the `QED` step should have a `BY` proof.

[1]Instead of `BY` we can write `PROOF BY`, which readers not familiar with TLA$^+$ proofs might find easier to understand.

[2]This requires the Toolbox to be set up to use the TLAPS library modules. Instructions to set it up should be available on the TLAPS web site.

5 Finding the Inductive Invariant

An inductive invariant almost always needs to assert type correctness of the variables. So the first thing to do is define the type-correctness invariant:

```
TypeOK == /\ flag \in [{0, 1} -> BOOLEAN]
          /\ pc \in [{0, 1} -> {"s1", "s2", "cs", "s3"}]
```

You can copy and paste it into the `SimpleMutex` module.

As shown by the example of why `Mutex` isn't an inductive invariant, invariance of `Mutex` depends on the fact that `flag[i]` equals `TRUE` when `pc[i]` equals `"s2"`. It's also clearly important that `flag[i]` equals `TRUE` when `pc[i]` equals `"cs"`. Instead of trying to figure out exactly what the important facts are about the relation between the values of `flag[i]` and `pc[i]`, we can just state what that relation is. It's easy to see from the code that `flag[i]` equals `TRUE` if and only if `pc[i]` equals `"s2"`, `"cs"`, or `"s3"`. So the inductive invariant should assert:

```
\A i \in {0,1} :
   (flag[i] = TRUE) <=> (pc[i] \in {"s2", "cs", "s3"})
```

The conjunction of this formula and `TypeOK` looks like it should be an inductive invariant, but it doesn't imply `Mutex`. Mutual exclusion is implemented by the algorithm because process `i` can reach label `cs` only when `flag[1-i]` equals `FALSE`. So, an obvious approach is to have `Inv` imply that `(pc[i]="cs") => (flag[1-i] = FALSE)`. Putting this all together, we get the definition:

```
Inv == /\ TypeOK
       /\ \A i \in {0,1} :
             /\ (flag[i] = TRUE) <=> (pc[i] \in {"s2", "cs", "s3"})
             /\ (pc[i] = "cs") => (flag[1-i] = FALSE)
```

With this definition, `Inv` implies `Mutex` because it implies that (a) a process's `flag` value equals `TRUE` if it's in its critical section and (b) if a process is in its critical section, then the other process's `flag` value equals `FALSE`. Copy and paste the definition into module `SimpleMutex`.

For algorithms that aren't so simple, writing proofs is hard. We want to do everything we can to check what we have to prove before trying to prove it. An inductive invariant is an invariant, so the first thing to check is that `Inv` is an invariant of `Spec`. Run TLC to check if it is. TLC reports that it isn't, because the algorithm can reach a state with `pc[0]="cs"` and `flag[1]=TRUE`, making the last line in the definition of `Inv` false for `i=0`.

This state has `pc[1]="s2"`, and is reached by process 1 executing statement `s1`, which sets `flag[1]` to `TRUE`, when process 0 is at `cs`.

The obvious way to fix this problem with `Inv` is to let it allow such a state by changing the last line of its definition to

```
/\ (pc[i] = "cs") => (flag[1-i] = FALSE) \/ (pc[1-i] = "s2")
```

Make this change to the definition and have TLC check if it's now an invariant. TLC finds that it is. But before trying to prove that it is an inductive invariant, we can let TLC try to check if the three steps `<1>1`–`<1>3` (of Figure 2 on page 8) are true.

TLC has already checked that `<1>1` is true, because invariance of `Inv` implies that `Inv` is true in every possible initial state. We can similarly get TLC to check `<1>3` by checking that `Mutex` is an invariant of a spec whose initial formula is `Inv`. The simplest such spec that TLC can check is one whose next-state action leaves all the variables unchanged. The TLA$^+$ translation of a PlusCal algorithm defines `vars` to be the tuple of all its variables, so a next-state relation that leaves all those variables unchanged can be written as

```
UNCHANGED vars
```

In the Toolbox, create a TLC model with a behavior specification having this `UNCHANGED` formula as its next-state relation and `Inv` as its initial predicate. Add `Mutex` as an invariant to be checked and run TLC on the model. TLC finds no error, so `<1>3` is true.

Let's now have TLC check step `<1>2`. That step asserts that starting in a state satisfying `Inv` and taking a `Next` step or a stuttering step produces a state that also satisfies `Inv`. This is true if and only if `Inv` is an invariant of the spec `Inv /\ [][Next]_vars`. To check that it is, create in the Toolbox a TLC model with a behavior specification having `Inv` as its initial predicate and `Next` as its next-state relation. Then have TLC check if `Inv` is an invariant of this spec. TLC finds that it is, so step `<1>2` is true.

Of course, in most cases, TLC can check these steps only on a small model of the spec, not on the complete spec. So there will still be something to prove. A closer look at the specs we've checked reveals that model checking should be less effective at finding errors in steps `<1>2` and `<1>3` than it is for finding errors in ordinary invariance checking. The specs we used to check these steps had `Inv` as their initial predicate. Formula `Inv` is not a typical initial predicate of a spec. TLC's algorithm for computing the set of initial states, which is explained in Section 14.2.6 of *Specifying Systems*, implies that it computes the initial states satisfying `Inv` by finding all states satisfying `TypeOK` and throwing away the ones that don't satisfy the second conjunct of `Inv`. Formula `TypeOK` is satisfied by all type-correct states, and most specs have an enormous number of such states. Even with models for which model checking the spec doesn't take very long, computing the set of type-correct states can cause TLC to run for a very long time, often running out of memory. Usually, checking `<1>2` and `<1>3` this way is practical only for the tiniest of models—models that are too small to provide much confidence that the steps are correct. However, even very tiny models are good at catching errors—especially when checking step `<1>2`. You should always do this checking before trying to write a proof.

A method for doing this checking on small, randomly chosen sets of type-correct states has recently been developed. It is described in the document Using TLC to Check Inductive Invariance. Preliminary tests indicate that it works quite well at quickly finding errors in an inductive invariant, even on models for which ordinary invariance checking takes quite a long time. It is worth trying before you start to write a proof.

6 Writing the Proof

If you've done all the checking you can with TLC, it's time to write the proof—which means proving steps `<1>1`–`<1>3`. Proofs of those steps use ordinary mathematical reasoning; they require no understanding of algorithms or mutual exclusion. Unfortunately, writing rigorous proofs is a science in which most computer scientists and mathematicians are unschooled. Fortunately, TLAPS will help you learn how to do it.

Before writing a proof, you should understand a little bit about "ordinary" math. What mathematicians call a *variable* is what programmers, computer scientists, and TLA$^+$ users call a *constant.* When you solved a problem involving the "variable" x in high-school algebra, the value of x was fixed throughout the problem. Usually, the problem was figuring out what that value was, given certain facts about it—for example that the value satisfied the equality $x^2 - 2x = -1$. The TLA$^+$ variables `flag` and `pc` of the simple mutex algorithm don't have fixed values; their values can be different in different states of an execution. However, when proving steps `<1>1` and `<1>3`, we can pretend that those TLA$^+$ variables are mathematical variables—that is, constants. This is because each of those steps is an assertion about states. To prove such an assertion true, we have to prove it true of a single, arbitrary state—that is, an arbitrary fixed assignment of values to those two variables. This is exactly what we do in ordinary math: we prove that an assertion is true for an arbitrary, fixed assignment of values to its (mathematical) variables. So we can prove `<1>1` and `<1>3` by pretending that the TLA$^+$ variables `flag` and `pc` are ordinary mathematical variables—that is, TLA$^+$ constants.

6.1 Proving Step `<1>1`

Here's how I went about trying to use TLAPS to prove `<1>1`. This is a simple enough assertion that I figured TLAPS should be able to prove it by just expanding the definitions of `Init` and `Inv`, as well as the definition of `TypeOK`, which appears in the definition of `Inv`. So, I tried this:

```
<1>1. Init => Inv
  BY DEF Init, Inv, TypeOK
```

Run the Prove Step command on this step. TLAPS fails to prove it, so the Toolbox colors the step red and raises a window showing exactly what formula TLAPS was trying unsuccessfully to prove. It would be a good idea to look at that formula to see if we can figure out why the proof failed. But instead of doing that, I decided to decompose the proof into smaller steps and see which of those steps TLAPS can't prove. This is often the best way to find a problem when the decomposition can be done for us by the Toolbox and requires no thinking.

6.1.1 The First Decomposition

The way to break a proof into smaller steps without thinking is to let the structure of the formula we're trying to prove tell us how. In this case, the first decomposition is one that is so natural to mathematicians that they usually don't even bother to write it down: they prove `Init => Inv` by assuming that `Init` is true and proving that `Inv` is true. What they are doing can be expressed in TLA$^+$ by replacing step `<1>1` by:[3]

```
<1>1a. ASSUME Init
       PROVE  Inv
```

Both `<1>1` and `<1>1a` assert the same thing: that `Init` implies `Inv`. However, their proofs are written differently. The proof of a step is decomposed

[3]Naming proof steps with sequential numbers is just a convention. Instead of naming our four steps `<1>1`–`<1>4`, we could have named them `<1>Tom`, `<1>Dick2`, `<1>16`, and `<1>3Harry`.

the same way we decomposed the proof of the entire theorem: as a sequence of steps ending with a `QED` step. We can use the fact that `Inv` is the conjunction of `TypeOK` and the formula `\A i \in ...` to decompose the proof of `<1>1` as follows:

```
<1>1. Init => Inv
  <2>1. Init => TypeOK
  <2>2. Init => \A i \in {0,1} : ...
  <2>3. QED
    BY <2>1, <2>2 DEF Inv
```

This same basic decomposition of the proof of `<1>1a` can be written:

```
<1>1a. ASSUME Init
       PROVE  Inv
  <2>1a. TypeOK
  <2>2a. \A i \in {0,1} : ...
  <2>3a. QED
    BY <2>1a, <2>2a DEF Inv
```

In both proofs, the `QED` stands for the goal to be proved. In the proof of `<1>1`, that goal is `Init => Inv`. In the proof of `<1>1a`, that goal is `Inv`. The proofs of `<2>1a` and `<2>2a` can assume the truth of the assumption `Init`.

The proof of `<1>1a` is nicer than the proof of `<1>1`; the assumption that `Init` is true is made right at the beginning. In the proof of `<1>1`, it will have to be made twice—in the proofs of both `<2>1` and `<2>2`. However, I think the statement of `<1>1` is nicer than that of `<1>1a`. We can have both by writing:

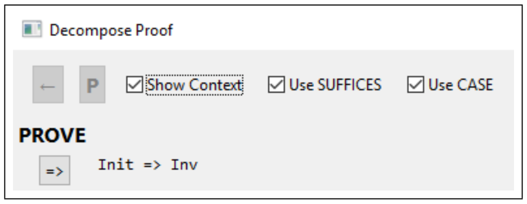
```
<1>1. Init => Inv
  <2>0. SUFFICES ASSUME Init
                 PROVE  Inv
    OBVIOUS
  <2>1a. TypeOK
  <2>2a. \A i \in {0,1} : ...
  <2>3a. QED
    BY <2>1a, <2>2a DEF Inv
```

In general, the step `SUFFICES T` asserts that proving `T` proves the current goal, and changes the current goal of the proof to that of the proof of `T`. The proof of a `SUFFICES T` step must show that `T` implies the current goal. The proof `OBVIOUS` is essentially an empty `BY` proof (which is not syntactically legal),[4] meaning that the step follows by ordinary mathematical reasoning

[4]You can write `PROOF OBVIOUS` instead of `OBVIOUS`.

from what it has already been told to use. (You'll see later how it can be told to use facts that don't appear in a `BY` proof.)

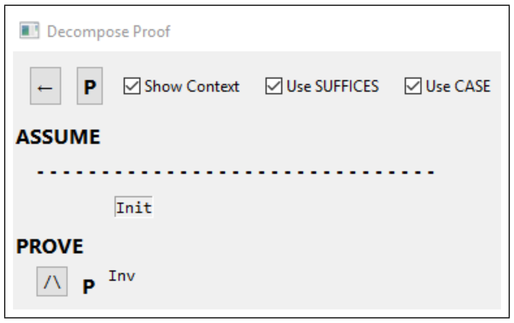
This last decomposition can be performed for us by the Toolbox using the Decompose Proof command. Running that command on step `<1>1` (by clicking on the step or its `BY` proof and typing control+g control+d) raises this window:



PROVE

```
Init => Inv
```

(Make sure that the Use SUFFICES option is selected.) Click on the $\boxed{\texttt{=>}}$ button to decompose the proof of the implication into a `SUFFICES ASSUME/PROVE` proof (but not to change it in the module yet), which changes the command's window to:



ASSUME

```
------------------------------
        Init
```

PROVE

```
Inv
```

Clicking on the $\boxed{\textsf{P}}$ button on the top line would tell the command to write that decomposition in the module and stop. Instead of doing that, click on the button labeled $\boxed{\texttt{/\textbackslash}}$. This decomposes the proof of the conjunction into two steps, replacing the original `BY` proof of step `<1>1` with this:

```
<2> SUFFICES ASSUME Init
             PROVE  Inv
  OBVIOUS
<2>1. TypeOK
  BY DEF Init, Inv, TypeOK
<2>2. \A i \in {0,1} :
         /\ (flag[i] = TRUE) <=> (pc[i] \in {"s2", "cs", "s3"})
         /\ (pc[i] = "cs") => (flag[1-i] = FALSE) \/ (pc[1-i] = "s2")
  BY DEF Init, Inv, TypeOK
<2>3. QED
  BY <2>1, <2>2 DEF Inv
```

The original `BY` proof has become the proof of both steps `<2>1` and `<2>2`. One significant change from the decomposition proposed above is that the `SUFFICES` step has no name, just the level number `<2>`. Were that step named `<2>0`, TLAPS would use the assumption `Init` only when we directed it to do so, which we can do by putting `<2>0` in `BY` proofs. A fact or assumption from an unnumbered step is automatically used by TLAPS. In this example, the assumption `Init` is automatically used in proving `<2>1` and `<2>2`.

6.1.2 Decomposing the Proof of `<2>2`

Tell TLAPS to check the proof of `<1>1` by clicking on that step and running the Prove Step or Module command. TLAPS successfully checks the proofs of the `SUFFICES` step and the `QED` step—which I knew it would because those steps were written by the Decompose Proof command. However, it fails on the proofs of `<2>1` and `<2>2`. At this point, I looked at what TLAPS was trying to prove and quickly saw the problem. Perhaps you can see it too. But let's examine proof decomposition some more before fixing the problem.

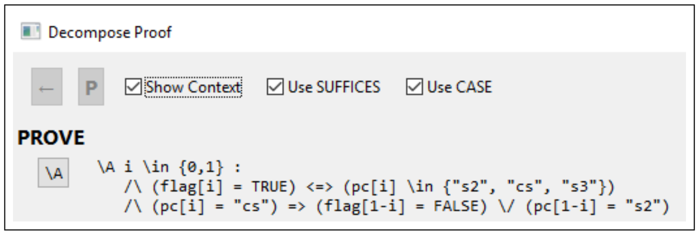
Step `<2>2` is an assertion of the form

```
<2>2. \A i \in {0, 1} : ...
```

If a mathematician writes an assertion like that in a proof (which she usually does by writing `\A i \in {0,1}` in prose), she will prove it by assuming that `i` is an element of `{0,1}` and proving formula "..."—usually without even mentioning that's what she's doing. Her proof can be expressed in TLA$^+$ as replacing step `<2>2` with

```
<2>2a. ASSUME NEW i, i \in {0,1}
       PROVE  ...
```

As we did when decomposing the implication `<1>1`, we can do this decomposition without changing step `<2>2`, instead putting the `ASSUME` / `PROOF` in a level-`<3>` `SUFFICES` step. Since the "..." is a conjunction of two formulas, we can decompose its proof into two steps. Again, we do all this with the Decompose Proof command. Putting the cursor in step `<2>2` or its `BY` proof and running the command raises this window:

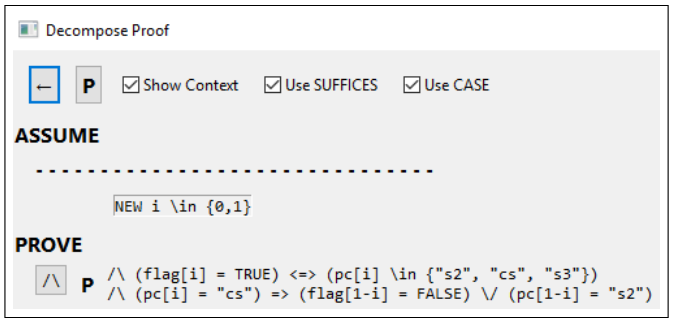

Decompose Proof

☑ Show Context ☑ Use SUFFICES ☑ Use CASE

PROVE

```
\A i \in {0,1} :
   /\ (flag[i] = TRUE) <=> (pc[i] \in {"s2", "cs", "s3"})
   /\ (pc[i] = "cs") => (flag[1-i] = FALSE) \/ (pc[1-i] = "s2")
```

Click on the $\boxed{\texttt{\\A}}$ button to decompose the proof of the implication into a `SUFFICES ASSUME/PROVE` proof and change the command window to:



```
ASSUME
        NEW i \in {0,1}
PROVE
 /\ (flag[i] = TRUE) <=> (pc[i] \in {"s2", "cs", "s3"})
 /\ (pc[i] = "cs") => (flag[1-i] = FALSE) \/ (pc[1-i] = "s2")
```

As before, click on the button labeled $\boxed{\texttt{/\\}}$. This decomposes the proof of the conjunction into two steps, replacing the original `BY` proof of step `<2>2` (which was originally the `BY` proof of `<1>1`) with this:

```
<3> SUFFICES ASSUME NEW i \in {0,1}
             PROVE  /\ (flag[i] = TRUE) <=> (pc[i] \in {"s2", "cs", "s3"})
                    /\ (pc[i] = "cs") => (flag[1-i] = FALSE) \/ (pc[1-i] = "s2")
  OBVIOUS
<3>1. (flag[i] = TRUE) <=> (pc[i] \in {"s2", "cs", "s3"})
  BY DEF Init, Inv, TypeOK
<3>2. (pc[i] = "cs") => (flag[1-i] = FALSE) \/ (pc[1-i] = "s2")
  BY DEF Init, Inv, TypeOK
<3>3. QED
  BY <3>1, <3>2
```

Now run the Prove Step or Module command on step `<2>2`. TLAPS proves all the steps except `<3>1`. The time has come to stop decomposing and start thinking. Step `<3>1` is true because the initial predicate `Init` implies `flag[i] = FALSE` and `pc[i] = "s1"`; so both sides of the equivalence equal `FALSE`. Now let's look at the obligation that the error window tells us TLAPS is trying to prove:

```
ASSUME NEW VARIABLE flag,
       NEW VARIABLE pc,
       /\ flag = [i \in {0, 1} |-> FALSE]
       /\ pc = [self \in ProcSet |-> "s1"] ,
       NEW CONSTANT i \in {0, 1}
PROVE  flag[i] = TRUE <=> pc[i] \in {"s2", "cs", "s3"}
```

The hard part of figuring out what went wrong is to look at this statement and see what it actually says, not what you think it should mean. TLAPS knows absolutely nothing about mutual exclusion, the PlusCal code, or anything else that is not mentioned in the statement. If you gave this statement to a mathematician to prove, she would tell you that you're asking her to prove something about `pc[i]`. She knows that `pc` is a function and that `i` equals 0 or 1, but she knows nothing about `pc[i]` because she doesn't even know if `i` is in the domain of `pc`. She knows that the domain of `pc` is a set called `ProcSet`, but she has no idea what that set is. The problem is that I forgot to tell TLAPS to use the definition of `ProcSet`. Go back to the two `BY` steps that TLAPS failed to check, steps `<2>1` and `<3>1`, and add `ProcSet` to their `DEF` clauses. Run the Prove Step or Module command on step `<1>1`, and TLAPS will now prove it. In fact, there's no need to decompose the proof of `<1>1`. TLAPS should be able to check that it is proved by:

```
    BY DEF Init, Inv, TypeOK, ProcSet
```

This example illustrates a problem of using PlusCal when writing proofs: The TLA$^+$ translation defines a few symbols that don't appear in the PlusCal code, but have to appear in the proof. Fortunately, there are few such symbols—usually just the three symbols `pc`, `ProcSet`, and `proc`, as well as the canonical names `Init`, `Next`, `vars`, and `Spec`. And you'll almost never be able to write a useful inductive invariant if you forget about `pc`.

6.2 Proving Step <1>2

6.2.1 Reduction to Ordinary Math

Let's now consider step `<1>2`, which is always the hardest to check. This step is different from steps `<1>1` and `<1>3` because those steps assert state predicates—formulas true or false on a state. We could prove them with ordinary mathematical reasoning by considering TLA$^+$ variables to be ordinary mathematical variables, known to us as constants. Step `<1>2` asserts an action formula, which is true or false on a step, which is a pair of states. We can't use ordinary mathematical reasoning about action formulas because they contain the priming operator $'$, which means "in the next state". Prime is not an operator of ordinary math. For example, an ordinary mathematical operator like unary minus ($-$) preserves equality–that is, $x = y$ implies $-x = -y$. That's true if x and y are constant expressions, or if they contain variables. That's why we could use ordinary math to prove `<1>1` and `<1>3`. However, priming does not preserve equality. If x and y are variables, then $x = y$ doesn't imply $x' = y'$ because x and y equal in one state doesn't imply that they're equal in the next state.

Ordinary mathematical operators are constant operators. A constant operator is one that can be defined in terms of the TLA$^+$ operators appearing in the lists of constant operators and miscellaneous constructs in the *Summary of TLA$^+$* and in Tables 1 and 2 (pages 268-269) of *Specifying Systems*. Priming distributes over constant operators. For the constant operator $+$, this means that $(x + y)'$ equals $x' + y'$ for any expressions x and y. Moreover, since any constant has the same value in all states of a behavior, c' equals c for any constant expression c. This implies that if we expand all the definitions in module `SimpleMutex` and distribute all primes "inward" over constant operators, any expression defined in the module can be rewritten in terms of only constants such as `TRUE`, constant operators, variables, and primed variables. Note that bound identifiers, such as `i` in `\A i \in {0,1} : ...` and in `[i \in {0,1} |-> ...]`, are constants. Therefore, `[i \in {0,1} |-> ...]'` equals `[i \in {0,1} |-> (...)']`.

If we expand the definitions from `SimpleMutex` in the statement of step `<1>2`, we get a formula containing only ordinary mathematical operators (constant operators) and the symbols `flag`, `pc`, `flag'`, and `pc'`. This formula is true for all steps (pairs of states) if and only if it is true for all possible assignments of the values to the symbols `flag`, `pc`, `flag'`, and `pc'`. Hence, it is true if it can be proved with ordinary mathematical reasoning when these four symbols are considered to be four different, completely unrelated mathematical variables (TLA$^+$ constants). So, that's what we have to do to prove step `<1>2`.

We don't actually have to completely expand all definitions so only variables are primed. For example, we can prove some simple results about the formula of step `<1>2` without expanding the definition of `Inv`; and we will do so in decomposing its proof. We just consider `Inv` and `Inv'` to be two independent, arbitrary values.

6.2.2 Decomposing the Proof

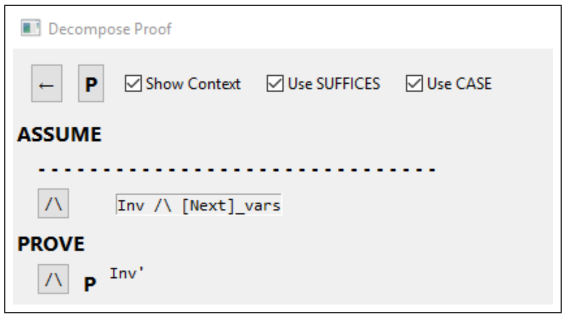
The forthcoming release of TLAPS may be able to check a simple `BY` proof of `<1>2` that expands all necessary definitions. However, that will usually not be the case for more complicated algorithms. So, let's examine how to decompose the proof. Recall that the step is:

```
<1>2. Inv /\ [Next]_vars => Inv'
```

Since it's an implication, we can decompose its proof as

```
<2> SUFFICES ASSUME Inv /\ [Next]_vars
             PROVE  Inv'
   OBVIOUS
...
```

Run the Decompose Proof command on step `<1>2`. (You can do that even though it has no `BY` proof.) As before, click on the $\boxed{\texttt{=>}}$ button, which changes the command's window to:

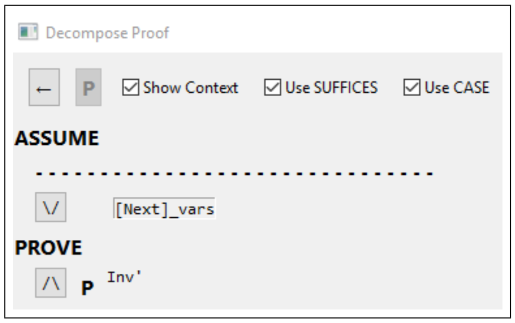


Make sure the Use CASE option is selected.

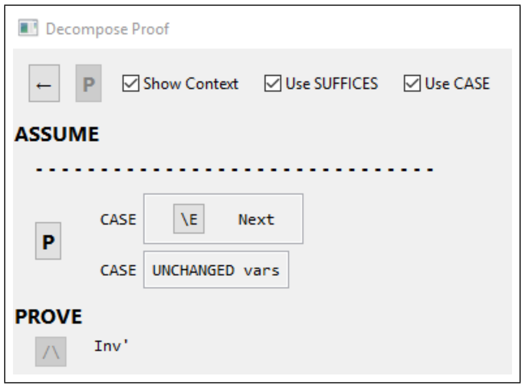
As before, we could decompose the proof to prove the two conjuncts of formula `Inv'` separately by clicking on the $\boxed{\texttt{/\\}}$ button at the bottom, next to `Inv'`. However, I've found that it's usually better to use a different decomposition instead. Click on the other $\boxed{\texttt{/\\}}$ button, next to the formula `Int /\ [Next]_vars`. This tells the Toolbox to replace the `SUFFICES` step with

```
<2> SUFFICES ASSUME Inv, [Next]_vars
             PROVE  Inv'
   OBVIOUS
```

It simply replaces the single assumption `Int /\ [Next]_vars` with the two assumptions `Int` and `[Next]_vars`. Just doing this accomplishes nothing; the two steps are completely equivalent. However, it changes the command's window to:



The $\boxed{\texttt{\\/}}$ button next to `[Next]_vars` tells us that we can further decompose the proof by using the fact that `[Next]_vars` is a disjunction—namely, it's the disjunction of `Next` and `UNCHANGED vars`. Clicking on that $\boxed{\texttt{\\/}}$ button changes the command window to:



```
Decompose Proof
←  P  ☑ Show Context  ☑ Use SUFFICES  ☑ Use CASE
ASSUME
  ------------------------------
      CASE  \E  Next
  P
      CASE  UNCHANGED vars
PROVE
  /\  Inv'
```

Clicking on the $\boxed{\textsf{P}}$ button will then generate this proof of `<1>2`:

```
<2> SUFFICES ASSUME Inv,
                    [Next]_vars
             PROVE  Inv'
  OBVIOUS
<2>1. CASE Next
<2>2. CASE UNCHANGED vars
<2>3. QED
  BY <2>1, <2>2
```

A statement `CASE P` is an abbreviation for `ASSUME P PROVE G` where `G` is the current goal. Thus, step `<2>1` is an abbreviation for

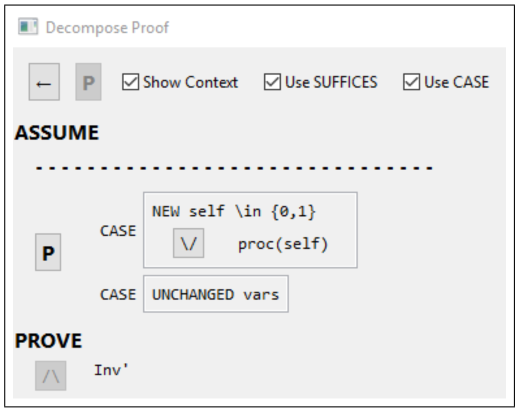
```
<2>1. ASSUME Next
      PROVE  Inv'
```

since `Inv'` is the current goal of the proof.

This proof is what would be generated by clicking on the window's $\boxed{\textsf{P}}$ button. However, the $\boxed{\texttt{\\E}}$ button indicates that we can further decompose the `Next` case because `Next` has the form `\E i \in S : ...`. An assumption of that form is used by simply assuming that the formula "..." is true for some value of `i` in `S`. Clicking on that $\boxed{\texttt{\\E}}$ button tells TLAPS that we want to replace step `<2>1` above with

```
<2>1. ASSUME NEW self \in {0,1},
             proc(self)
      PROVE  Inv'
```

It also changes the command window to:



ASSUME
- -

```
P   CASE  NEW self \in {0,1}
          \/   proc(self)
    CASE  UNCHANGED vars
```

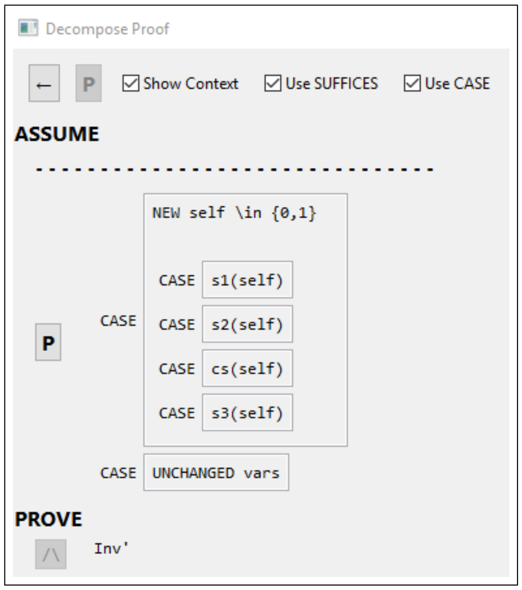
PROVE

```
/\  Inv'
```

The $\boxed{\backslash/}$ button next to `proc(self)` indicates that a further decomposition of that case can be done because `proc(self)` is a disjunction. The definition of `proc(self)` in the module tells us that it equals

```
s1(self) \/ s2(self) \/ cs(self) \/ s3(self)
```

Click on that $\boxed{\backslash/}$ button. It changes the command's window to:



ASSUME
- -

```
    CASE  NEW self \in {0,1}
            CASE s1(self)
            CASE s2(self)
P           CASE cs(self)
            CASE s3(self)
    CASE  UNCHANGED vars
```

PROVE

```
/\  Inv'
```

confirming that it has split step `<2>1` into those four separate cases. Finally, click on the $\boxed{\text{P}}$ button to put into the module the proof of `<1>2` in Figure 3.

```
<1>2. Inv /\ [Next]_vars => Inv'
  <2> SUFFICES ASSUME Inv,
                      [Next]_vars
               PROVE  Inv'
    OBVIOUS
  <2>1. ASSUME NEW self \in {0,1},
               s1(self)
        PROVE  Inv'
  <2>2. ASSUME NEW self \in {0,1},
               s2(self)
        PROVE  Inv'
  <2>3. ASSUME NEW self \in {0,1},
               cs(self)
        PROVE  Inv'
  <2>4. ASSUME NEW self \in {0,1},
               s3(self)
        PROVE  Inv'
  <2>5. CASE UNCHANGED vars
  <2>6. QED
    BY <2>1, <2>2, <2>3, <2>4, <2>5 DEF Next, proc
```

Figure 3: The decomposition of step `<1>2`.

We are left with the problem of proving steps `<2>1` – `<2>5`. The proof of `<2>5` is trivial, since the truth of `Inv'` (the current goal) follows trivially from the truth of `Inv` (assumed in the `SUFFICES` statement) and the assumption `UNCHANGED vars`, which asserts that all the variables that appear in `Inv` are unchanged. So TLAPS will be able to check an `OBVIOUS` proof.

The steps whose proofs are not trivial are `<2>1` – `<2>4`. Each of those steps asserts that the state transition described by each labeled statement of the algorithm preserves the truth of the inductive invariant `Inv`. Since their goal is `Inv'`, which is a conjunction, they can each be further decomposed with the Decompose Proof command.

6.3 The Hard Part

I've discussed the proofs of steps `<1>1` and `<1>2`. The proof of `<1>3` is similar to that of `<1>1`, and I won't discuss it. The explanation of how we decomposed the proofs was quite long. With a bit of practice, the actual decomposition is performed very quickly with just a few clicks, using the Decompose Proof Toolbox command.

You have seen all the basic methods of decomposing a proof that can be performed with the Decompose Proof command. Further information about the command, including the available options, are described in its help page. That page can be viewed by clicking on the $\boxed{?}$ button in the upper-right corner of the command's window, or by clicking here. Real algorithms are more complex than `SimpleMutex`. But their invariance proofs all have the same basic structure. The proofs of steps `<1>1` and `<1>3` are usually easy; TLAPS can often check them with `BY` proofs that just tell it what definitions to expand.

The hard part of invariance proofs of real algorithms are the proofs corresponding to steps `<2>1`–`<2>4` in the proof of `<1>2` of `SimpleMutex`, which are shown in Figure 3. Those proofs can usually be decomposed further without thinking, based solely on the structure of what is to be proved, with a few clicks of the Decompose Proof command. Many of the resulting steps will have trivial proofs—for example, a step asserting that an action of the algorithm leaves true a part of the invariant all of whose variables are left unchanged by that action.

I'll illustrate how that's done for step `<1>2` in Figure 3. Since this is the step in which a process enters its critical section, it probably has the most interesting proof. The goal `Inv'` is a conjunction, so we can use the Decompose Proof command to obtain this proof:

```
<3>1. TypeOK'
<3>2. (\A i \in {0,1} :
          /\ (flag[i] = TRUE) <=> (pc[i] \in {"s2", "cs", "s3"})
          /\ (pc[i] = "cs") => (flag[1-i] = FALSE) \/ (pc[1-i] = "s2"))'
<3>3. QED
  BY <3>1, <3>2 DEF Inv
```

Step `<3>2` looks like the more difficult step to prove, so let's prove it. We can decompose it the same way we decomposed the proof of step `<2>2` of the proof of `<1>1`, in Section 6.1.2 on page 15. This yields:

```
<4> SUFFICES ASSUME NEW i \in ({0,1})'
             PROVE  (/\ (flag[i] = TRUE) <=> (pc[i] \in {"s2", "cs", "s3"})
                     /\ (pc[i] = "cs") => (flag[1-i] = FALSE) \/ (pc[1-i] = "s2"))'
  OBVIOUS
<4>1. ((flag[i] = TRUE) <=> (pc[i] \in {"s2", "cs", "s3"}))'
<4>2. ((pc[i] = "cs") => (flag[1-i] = FALSE) \/ (pc[1-i] = "s2"))'
<4>3. QED
  BY <4>1, <4>2
```

Step `<4>2` is the more interesting one, so we'll prove it. Step `<2>2` asserts that an `s2` step preserves the truth of the invariant `Inv`. So, our proof

```
s2(self) == /\ pc[self] = "s2"
            /\ flag[1-self] = FALSE
            /\ pc' = [pc EXCEPT ![self] = "cs"]
            /\ flag' = flag
```

Figure 4: The $s2$ action.

will rely heavily on assumption `s2(self)` of `<2>2` and its definition, which appears in Figure 4.

When you start trying to prove `<4>2`, you should realize that the argument depends on whether `i` and `self` are the same process or different processes. This suggests that the two cases should be made separate steps and proved separately. The Decompose Proof command won't do that for us; we have to do it manually to obtain:

```
<5>1. CASE i = self
<5>2. CASE i /= self
<5>3. QED
  BY <5>1, <5>2
```

Since we typed this ourselves, we should have TLAPS check the proof of the QED step to make sure we didn't make a mistake. We might as well check the proofs added by the Decompose Proof command as well, so we tell TLAPS to check the entire proof of step `<2>2`. (We do this by clicking anywhere in the step itself and running the Prove Step or Module command.) The Toolbox colors yellow the steps that have no proofs.

Step `<5>1` seems easy enough to prove that an informal paragraph proof should be convincing enough. Here's what I might write:

```
<5>1. CASE i = self
  (****************************************************************)
  (* The <2>2 assumption s2(self) and the definition of s2(self),  *)
  (* which by the case assumption equals s2(i), imply             *)
  (* flag[1-i] = FALSE and flag[1-i]' = flag[1-i].  Hence, <4>2 is *)
  (* true by propositional logic.                                 *)
  (****************************************************************)
```

I'm assuming that the reader of the proof understands that

- `<4>2` is the current goal.
- Because priming distributes over the constant operators `=>`, `\/`, and `=`, formula `<4>2` equals

  ```
  ... => (flag[1-i]' = FALSE) \/ ...
  ```

- By propositional logic, any formula of the form `... => TRUE \/ ...` equals `TRUE`.

Were these three facts obvious to you? If not, or if you're writing the proof for someone for whom they're not obvious, you should expand the proof to explain that they're being used. The best way to do that is to decompose the proof into substeps—for example:

```
<6>1. /\ flag[1-i] = FALSE
      /\ flag[1-i]' = flag[1-i]
<6>2. flag[1-i]' = FALSE
<6>3. QED
```

But if you want to write rigorous proofs of real algorithms, those three facts should be obvious to you. If not, you'll need to practice on very simple examples like this algorithm—and perhaps brush up on your propositional logic.

Proving `<5>2` seems to require considering separately the two cases depending on whether `pc[i]` equals or doesn't equal `"cs"`. So, I wrote the following proof. Read it and make sure you understand it.

```
<5>2. CASE i /= self
  <6>1. CASE pc[i] = "cs"
    (***************************************************************)
    (* The <5>2 case assumption implies i = 1-self, so the <2>2    *)
    (* assumption s2(self) and the definition of s2 imply          *)
    (* flag[i] = FALSE.  However, the <6>1 case assumption and the *)
    (* definition of Inv, assumed in the level <2> SUFFICES step,  *)
    (* imply flag[i] = TRUE.  This contradiction shows that the    *)
    (* <6>1 case is impossible.                                    *)
    (***************************************************************)
  <6>2. CASE pc[i] /= "cs"
    (***************************************************************)
    (* The <2>2 assumption s2(self), the definition of s2(self),   *)
    (* and the <5>2 case assumption imply that pc[i] = pc[i]'.  By *)
    (* the <6>2 case assumption, this implies that pc[i]' = "cs"   *)
    (* is false, so <4>2 is vacuously true.                        *)
    (***************************************************************)
  <6>3. QED
    BY <6>1, <6>2
```

This proof of `<6>2` would convince most people. However, TLAPS would not be able to check it. If you decomposed the proof into simpler steps, you'd discover that TLAPS would not be able to prove `pc[i] = pc[i]'`.

This seems surprising, since we're assuming and `s2(self)`, whose definition asserts

```
pc' = [pc EXCEPT ![self] = "cs"]
```

The assumption `i /= self` therefore seem to imply that the value of `pc[i]` is unchanged. However, it implies this only if `pc` is a function and `i` is in its domain. To deduce that, we need to assume that `TypeOK` is true, which is implied by our assumption that `Inv` is true. Thus, TLAPS has to be told to use the definitions of `Inv` and `TypeOK`.

For this simple example, we can avoid worrying about what TLAPS needs to know and simply have it expand all definitions and use all the facts it knows. However, that doesn't work for a more complicated algorithm because TLAPS will often fail to find a proof if it wastes too much time trying to use irrelevant facts. Having to tell TLAPS to use exactly the facts it needs is one reason that it can take a lot of work to translate an informal proof into a TLA$^+$ proof that TLAPS can check. Another reason is that while TLAPS is better at some kind of reasoning than humans, it can be a lot worse at other kinds. For example, it isn't very good at deciding when to split a proof into cases, so you may have to help it by decomposing the proof of a step with `CASE` statements.

Writing this kind of rigorous informal proof is useful even if you intend to complete the proof to one that TLAPS can check. Trying to get TLAPS to prove something that isn't true can waste a lot of time, so it's worthwhile to first make sure that the step is true. The informal proof can also guide you in writing the complete proof. Moreover, it will help others understand the final proof.

You have seen most of what you need to know about the TLA$^+$ constructs for writing proofs. All of those constructs are explained in the document *TLA+ Version 2: A Preliminary Guide*. The important constructs that you haven't yet seen are `USE`, `DEFINE`, and `PICK`. They are explained in sections 7.4 and 7.5 of the aforementioned document.

6.4 Using One Invariant to Prove Another

Sometimes we want to prove more than one invariant. The first invariant has to be proved as explained above, using an inductive invariant. But that's not necessary for the rest, because we can use invariants we've already proved to prove others. Here's the generalization of the Inductive Invariance Rule that let's us use the invariance of `Inv1` to help prove the invariance of `Inv2`.

Generalized Inductive Invariance Rule

```
          THEOREM  Init /\ Inv1 => Inv2
     and  THEOREM  Inv1 /\ Inv2 /\ [Next /\ Inv1']_vars => Inv2'
   imply  THEOREM  []Inv1 => (Init /\ [][Next]_vars => []Inv2)
```

The rule is valid because, for any behavior, `[]Inv1` implies that `Inv1` and `Inv1'` are true at any point in the behavior. The hypothesis `Inv1` in the first theorem is seldom necessary. If `Inv1` is an inductive invariant, then The "`/\ Inv'`" is not necessary in the second theorem because it's implied by `Inv1 /\ Next`. The rule is embodied in the following high-level proof structure:

```
THEOREM  Thm1  ==  Spec => []Inv1

THEOREM  Spec => []Inv2
<1>1. Init /\ Inv1 => Inv2
<1>2. Inv1 /\ Inv2 /\ [Next /\ Inv1']_vars => Inv2'
<1>3. QED
  BY <1>1, <1>2, Thm1, PTL DEF Spec
```

We can use TLC to check steps `<1>1` and `<1>2` much as we did for the ordinary Inductive Invariance Rule. TLC can check `<1>1` by checking that `Inv2` is an invariant of a specification with initial predicate `Init /\ Inv1` and next-state action `UNCHANGED vars`. It can check `<1>2` by checking that `Inv2` is an invariant of a specification with initial predicate `Inv1 /\ Inv2` and next-state action `Next /\ Inv1'`.

7 Examples

The following examples give you the opportunity to write your own proofs of invariance. Even if you don't write complete proofs, you should at least find the necessary inductive invariants with the help of TLC.

7.1 Peterson's Algorithm

Peterson's algorithm is a well-known, very simple two-process mutual exclusion algorithm. It avoids the deadlock of the simple protocol in our example by introducing a variable `turn`, whose value is always either 0 or 1, and changing process `i`'s `await` statement to

```
await (flag[1-i] = FALSE) \/ (turn = 1-i)
```

Thus, if both processes are at their `await` statements, process `1-turn` will be able to enter its critical section. A process `i` sets `turn` equal to `i` when it exits the critical section. If the other process is waiting then, it can enter the critical section before process `i` can enter again.

The algorithm is described on this web page. It's easy to prove mutual exclusion with a simple modification of the inductive invariant of the simple algorithm used here.

7.2 Another Simple Example

Our next example appeared in this note published in the *ACM SIGACT News*. Although very simple, it's a high-level abstraction of one aspect of a very complex cache coherence algorithm [3]. I originally devised it and proved its correctness to help me understand that algorithm. A modified version of the cache-coherence algorithm is on the Web.

The algorithm presented here is executed by a set of processes, which we think of as arranged in a ring facing inwards. There are two variables, `x` and `y`, whose values are functions having as domain the set of processes. Initially `x[i]` equals 0 for each process `i`; the initial value of `y[i]` doesn't matter. Each process `i` performs two actions: It first sets `x[i]` to 1; it then sets `y[i]` to `x[j]`, where `j` is the process to its right, and terminates.

Here's the algorithm written in PlusCal. The set of process identifiers is the set `0..(N-1)`, where `N` is a positive integer. It's convenient to make the initial values of all the `y[i]` integers; those initial values are somewhat arbitrarily taken to be 0.

```
--algorithm Simple {
    variables x = [i \in 0..(N-1) |-> 0],
              y = [i \in 0..(N-1) |-> 0] ;
    process (proc \in 0..N-1) {
      a: x[self] := 1 ;
      b: y[self] := x[(self-1) % N]  }       }
```

The problem is to prove that, when the algorithm terminates, `y[i]` equals 1 for at least one process `i`. Here's a simple behavioral proof. Suppose `i` is the last process to terminate. Its right-hand neighbor `(i-1) % N` must have already set `x[(i-1) % N]` to 1, so process `i` must have set `y[i]` to 1 when it executed action `b` and terminated.

The statement that all processes have terminated is expressed in TLA$^+$ as `\A i \in 0..(N-1) : pc[i] = "Done"`. The problem is therefore to prove the invariance of this formula:

```
(\A i \in 0..(N-1) : pc[i] = "Done")
    =>  (\E i \in 0..(N-1) : y[i] = 1)
```

The key to the proof is finding the inductive invariant that implies this formula. You can compare your solution to one on the web that contains a complete TLAPS-checked proof.

When `N` equals 2, this algorithm is essentially the simple mutual exclusion protocol of Section 3. To see why, let's first rewrite that protocol as follows by adding a variable `y[i]` local to each process and changing the `await` to an equivalent waiting loop.

```
s1: flag[i] := TRUE ;
s2: y[i] := flag[1-i] ;
    if (y[i] = TRUE) {goto s2} ;
cs: skip;
```

Let's now replace `TRUE` by 1 and `flag` by `x`. Since `1-i` equals `(i-1) % 2` for `i` in `{0,1}`, we can then rewrite the protocol as:

```
s1: x[i] := 1 ;
s2: y[i] := x[(i-1) % 2] ;
    if (y[i] = 1) {goto s2} ;
cs: skip;
```

Each process in the simple mutual exclusion protocol begins by executing a process of our example algorithm for `N` equal to 2. Therefore, at least one of the processes must find `y[i]` equal to 1 when executing statement `s2`, so both processes can't enter their critical sections.

7.3 Loop Invariants

Before you write a piece of code that is supposed to compute something and then terminate, you should decide what the code is supposed to compute. What it computes is usually expressed as a *postcondition*—an assertion Q that should be true when the code terminates. Depending on what that code is for, Q might be written as a mathematical formula or in prose.

Avoiding errors requires understanding why a piece of code satisfies its postcondition. Nontrivial code almost always contains one or more loops, and the key to understanding why it satisfies its postcondition is understanding why its loops satisfy their postconditions.

A loop satisfies its postcondition because it satisfies a *loop invariant*. A loop invariant for this PlusCal loop

```
a: while (test) { ... }
```

is an invariant of the form `(pc = "a") => P` where `P /\ (test = FALSE)` implies the loop's postcondition. (Formula `P` is usually called the loop invariant, the "`(pc = "a") =>`" being left implicit.)

Below are three examples of loops for which you are asked to find the loop invariant and prove its invariance. Of course, proving its invariance requires finding an inductive invariant that implies it. All three examples are PlusCal algorithms that consist entirely of a single **while** loop, each iteration of which is a single atomic action. The program contains only the single label `a` for the **while** loop, so `pc` can equal only `"a"` and `"Done"`. The formula `P`, as well as the formula `(pc = "a") => P`, will be invariant. Solutions to these three problems can be found in the GitHub TLA$^+$ examples repository. However, you should write your own solutions before looking at them.

7.3.1 Summing a Sequence

Our first example is a very simple algorithm that computes the sum

```
seq[1] + seq[2] + ... + seq[Len(seq)]
```

of a sequence `seq` of integers. To make model checking easier, `seq` is taken to be a sequence of elements in an unspecified set `Values` that we assume is a set of integers. Here is the PlusCal code:

```
--fair algorithm SumSequence {
    variables seq \in Seq(Values), sum = 0, n = 1 ;
    { a: while (n =< Len(seq))
          { sum := sum + seq[n] ;
            n := n+1              }   }    }
```

The module containing the algorithm should contain:

```
CONSTANT Values
ASSUME Values \subseteq Int
```

The postcondition satisfied by the algorithm is:

```
sum = seq[1] + seq[2] + ... + seq[Len(seq)]
```

To write this as a TLA$^+$ formula, we define an operator `SeqSum` so the sum equals `SeqSum(seq)`. Here is the definition:

```
RECURSIVE SeqSum(_)
SeqSum(s) == IF s = << >> THEN 0 ELSE s[1] + SeqSum(Tail(s))
```

The postcondition to be satisfied by the algorithm is the invariance of

```
(pc = "Done") => (sum = SeqSum(seq))
```

To prove that the algorithm satisfies this postcondition, you will need to find a suitable loop invariant. You should use TLC to help you. (You should first use it to check that you and I have not made any error, and the algorithm really does satisfy the postcondition.) To run TLC on this algorithm, you will have to create a model in which the definition of `Seq` is changed so that `Seq(S)` is a set containing only suitably short sequences. The following defines `Seq(S)` to consist of the set of all sequences of elements in `S` that have length at most 3.

```
Seq(S) == UNION {[1..i -> S] : i \in 0..3}
```

You should use very small values of 3, starting with 1 and increasing them by 1 until TLC takes too long to do its checking.

7.3.2 Binary Search

Binary search is a classic algorithm for searching a list of items sorted by key to find one with a given key. We can most simply describe the algorithm by assuming that the items consist only of keys, which are integers. So we assume as input a sorted sequence `seq` of integers and an integer `val`, and we let the algorithm set `result` to the computed output. The postcondition is that the `result` equals a number `i` such that `seq[i]` equals `val` if such an `i` exists; otherwise, `result` equals `0`.

Again, to make model checking easier, we assume that `seq` is a sequence of values in some set `Values` of integers. The assumption that the sequence is sorted is expressed by requiring it to be an element of the set `SortedSeqs` of all sorted sequences of elements in `Values`, which can be defined by:

```
SortedSeqs  ==
  {ss \in Seq(Values) : \A i, j \in 1..Len(ss) :
                           (i < j) => (ss[i] =< ss[j])}
```

The algorithm is in Figure 5. You will have to define the postcondition as well as the loop invariant. However, writing a correct binary search algorithm is tricky. Before looking at Figure 5, try writing your own binary search algorithm in PlusCal and checking it with TLC. It should begin with the variable declarations

```
seq \in SortedSeqs, val \in Values,
--fair algorithm BinarySearch {
   variables seq \in SortedSeqs, val \in Values,
             low = 1, high = Len(seq), result = 0 ;
   { a: while (low =< high /\ result = 0) {
          with (mid = (low + high) \div 2, mval = seq[mid])
           { if (mval = val) { result := mid}
               else if (val < mval) { high := mid - 1}
               else {low := mid + 1}                   } } }  }
```

Figure 5: The Binary Search Algorithm.

7.3.3 Quicksort

Quicksort is a well-known sorting algorithm invented by Tony Hoare. If you're not familiar with Quicksort, please read a conventional description of the algorithm before looking at the version described here. The Wikipedia web page is one place you might look. You can ignore questions of efficiency and details of how the partition procedure is implemented.

Now for our algorithm. As with binary sort, we assume that the input is a sequence `seq` of integers. The algorithm sorts the sequence in place, so the postcondition has two parts:

- The final value of `seq` is a permutation of its initial value.
- The final value of `seq` is sorted.

The definition of `SortedSeqs` in the binary search algorithm shows how to express the second condition precisely. Most people have trouble expressing the first condition. Your first impulse might be to describe what a permutation is in terms of an algorithm for generating the set `PermsOf(s)` of all permutations of a sequence `s`. The simplest algorithm I can think of is based on the idea of constructing an arbitrary permutation `ps` of `s` by starting with `ps` equal to the empty sequence and repeatedly removing an arbitrary element from `s` and appending it to the end of `ps`. As a fun exercise, I suggest writing a PlusCal algorithm based on this idea to compute `PermsOf(s)`. I think you'll find that the algorithm is pretty complicated. A more useful fun exercise is to write a recursive TLA$^+$ definition of `PermsOf`, based on the same idea. It will be more compact, but still hard to understand.

Rather than thinking algorithmically, we need to think mathematically to find a simple definition of `PermsOf`. The first thought we might have is that one sequence is a permutation of the other if both sequences have the same set of elements. We quickly realize that's not right because the sequences `<<1, 1, 2>>` and `<<1, 2, 2>>` have the same set of elements. That idea works only if the elements in the sequence are distinct.

Further thought reveals that to specify a permutation of a sequence of `n` elements, we just have to specify a permutation of the sequence `<<1,...,n>>`, which has distinct elements. For example if `s` is a 3-element sequence, the permutation `<<3,1,2>>` of `<<1,2,3>>` specifies the permutation `<<s[3],s[1],s[2]>>`.

Remember that a sequence is a function, and `<<3,1,2>>` is the function `f` with domain `1..3` such that `f[1]` equals 3, `f[2]` equals 1, and `f[3]` equals 2. The sequence `<<s[3],s[1],s[2]>>` is the sequence `sf` defined by `sf[i] = s[f[i]]` for all `i` in `1..3`. Mathematicians call `sf` the *composition* of the functions `s` and `f`, and they write it $s \circ f$. Since TLA$^+$ uses $\circ$ to mean concatenation of sequences, let's instead write it as `s**f`, which we can define by:

```
s**f  == [i \in 1..Len(f) |-> s[f[i]]]
```

We can now define `PermsOf(s)` to be the set of all `s**f` such that `f` is a permutation of `<<1,...,Len(s)>>`. A permutation of `<<1,...,Len(s)>>` is a function `f` in `[1..Len(s) -> 1..Len(s)]` such that every number `j` in `1..Len(s)` equals `f[i]` for exactly one number `i` in `1..Len(s)`. Mathematicians sometimes call such a function an *automorphism* of the set `1..Len(s)`. For a finite set `S`, a function `f` in `[S -> S]` is an automorphism of `S` if either of the following two equivalent conditions hold:

- `\A x, y \in S : (x /= y) => (f[x] /= f[y]`
- `\A y \in S : \E x \in S : f[x] = y`

We can generalize all this to define `PermsOf(s)` not just for a sequence `s`, but for an arbitrary function `s`:

```
PermsOf(s) ==
  LET Automorphisms(S)  ==
         {f \in [S -> S] : \A y \in S : \E x \in S : f[x] = y}
      g ** f  ==  [x \in DOMAIN f |-> g[f[x]]]
  IN  { s ** f : f \in Automorphisms(DOMAIN s) }
```

Having defined `PermsOf`, we can get to the Quicksort algorithm. The most interesting part of the algorithm is the *Partition* procedure. That's what determines how efficient the algorithm is, and it's the hardest thing to get right. But for this example, we're going to ignore how it's implemented.

We're going to write an algorithm in which the *Partition* procedure's choosing of a pivot index within the subinterval and rearranging the elements of the sequence `seq` are performed as part of a single atomic action. We define the constant operator `Partitions` so that if `s` is the current value of the variable `seq`, then `Partitions(I,p,s)` is the set of all new values of `seq` that can be produced by executing a *Partition* operation on the subinterval `I` using pivot index `p` in `I`. The definition is:

```
Partitions(I, p, s) ==
  {t \in PermsOf(s) :
     /\ \A i \in (1..Len(s)) \ I : t[i] = s[i]
     /\ \A i, j \in I : (i =< p) /\ (p < j) => (t[i] =< t[j])}
```

Finally, we come to the PlusCal algorithm. Again, `seq` is initially a sequence with values in some subset `Values` of `Int`. To be able to state the postcondition, it uses a variable `seq0` to hold the initial value of `seq`. The value of `seq0` is initially set equal to `seq` and is never changed. The algorithm uses constant operators `Min` and `Max`, defined so `Min(S)` and `Max(S)` equal the minimum and maximum elements of a finite, nonempty set `S` of integers.

Since PlusCal allows procedures that can call themselves recursively, it's easy to use it to write the usual version of Quicksort with a recursive procedure that takes the current subinterval `I` as argument(s), chooses the pivot index `p`, and sets `seq` to an arbitrary element of `Partitions(I,p,s)`. But instead, our PlusCal algorithm does not use recursion, and it generalizes the usual version. Before looking at it, see if you can devise a non-recursive Quicksort algorithm. I've found that most computer scientists can't do it in ten minutes.

The algorithm is in Figure 6. You should be able to figure out how it works and how to write its postcondition. Here's a hint for finding the inductive invariant. The algorithm removes from the set `U` subintervals `I` consisting of a single number. Think of what the inductive invariant would be if the algorithm didn't remove those intervals and instead terminated when they were the only intervals in `U`.

```
--fair algorithm Quicksort {
  variables  seq \in Seq(Values) \ {<< >>}, seq0 = seq,
             U = {1..Len(seq)};
  { a: while (U # {})
        { with (I \in U)
            { if (Cardinality(I) = 1)
                { U := U \ {I} }
              else
                { with (p \in Min(I)..(Max(I)-1),
                        I1 = Min(I)..p,
                        I2 = (p+1)..Max(I),
                        newseq \in Partitions(I, p, seq))
                    { seq := newseq ;
                      U := (U \ {I}) \cup {I1, I2} }   }  }  }  }  }
```

Figure 6: The Quicksort Algorithm

Part II
Implementation

8 Introduction

After invariance, the next most common safety property of a TLA$^+$ specification that one proves is that it implements a higher-level specification under a refinement mapping. Implementation under a refinement mapping is explained in Lecture 10 of the TLA$^+$ Video course.

The statement that one specification S_1 implements another specification S_2 under a refinement mapping is usually expressed in TLA$^+$ in this form:

```
THEOREM  Spec => HL!Spec
```

where

- S_1 is the specification named `Spec` in the current module
- S_2 is the specification named `Spec` in a module `HigherLevel` that is imported into the current module with a statement

  ```
  HL == INSTANCE HigherLevel WITH ...
  ```

- The refinement mapping is the substitution of expressions of the current module for the declared constants and variables of module `HigherLevel` specified by the `WITH` clause.

We describe how to prove such a theorem when the specifications are canonical TLA$^+$ safety specifications of the form `Init /\ [][Next]_vars`. If the lower-level specification S_1 also has a conjunct specifying a fairness condition, that conjunct can be ignored because a fairness condition does not affect whether the spec satisfies a safety property. One of the advantages of writing specifications in TLA$^+$ is that any (finite or countably infinite) conjunction of weak and/or strong fairness conditions on actions that imply the spec's next-state action `Next` is a fairness condition.

As with proofs of invariance, we explain (i) what needs to be proved, (ii) how to use TLC to check that what needs to be proved is true, and (iii) how to use TLA$^+$ proof constructs to write and check the high-level structure of a rigorous prose proof.

9 The Two-Phase Handshake Protocol

Our goal is to prove the correctness of a simple widely-used hardware protocol by showing that it implements a high-level specification of that protocol. The purpose of the protocol is to allow two processes, called *pro* and *con*, to alternately perform operations. First *pro* can perform an operation, then *con* can perform an operation, then *pro* can perform an operation, and so on. Since we are interested only in safety, we do not require that any operation must be performed.

To model this protocol, we represent the state being read and modified by the two processes with a variable `s` whose initial value equals an unspecified constant `s0`. We represent a *pro* operation by the statement `s := P(s)` and a *con* operation by `s := C(s)`, where `P` and `C` are unspecified constant operators. For simplicity, we assume that the processes' operations are atomic actions.

We want a protocol to ensure that initially `s` equals `s0` and that the statements `s := P(s)` and `s := C(s)` can only be executed alternately. This requirement is obviously described by the algorithm in this module:

```
-------------------- MODULE Alternation --------------------
CONSTANTS s0, P(_), C(_)

(*****
--algorithm Alternation
  { variable s = s0 ;
    { a: while (TRUE)
          {    s := P(s) ;
            b: s := C(s) } } }
*****)
=============================================================
```

Create a specification with `Alternation` as its root module and run the PlusCal to TLA$^+$ translator.

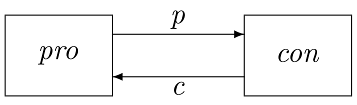
The parameters of the specifcation `Spec` created by the translation are the three constants and the variables `s` and `pc`. We are interested in the executions of the processes' two operations, which are represented by the assignments to `s`. The variable `pc` is used only to describe the desired sequencing of those operations. An implementation of this specification should contain the same three constants and the variable `s`. It may contain any other constants and variables. The implementation's spec should implement formula `Spec` of module `Alternation` under a refinement mapping (instantiation) that substitutes its constants `s0`, `P`, and `C` and variable `s` for the

```
----------------------- MODULE TwoPhase -----------------------
EXTENDS Integers

CONSTANTS s0, P(_), C(_)

(*****
--algorithm TwoPhase
  { variable s = s0, p = 0 ; c = 0;
     process (pro = "a")
       { a1 : while (TRUE)
               {     await p = c ;
                 a2: s := P(s) ;
                 a3: p := (p + 1) % 2  } }

     process (con = "b")
       { b1 : while (TRUE)
               {     await p /= c ;
                 b2: s := C(s) ;
                 b3: c := (c + 1) % 2  } } }
*****)
===============================================================
```

Figure 7: The two-phase handshake.

constants and variable of the same name, but can substitute any expression for the variable `pc`.

Our example implementation is a hardware protocol called the *two-phase handshake*. The protocol can be pictured like this



where p and c are one-bit values, with p set by *pro* and read by *con* and c set by *con* and read by *pro*. The actual algorithm is in Figure 7.

Try executing this algorithm by hand. You'll see that there is only one possible execution. (More precisely, that's true if we ignore the stuttering steps that any algorithm allows.) You'll also notice that, for each `i` in `{0,1}`, the expression `(i+1)%2` equals the "complement" of `i`. Create a specification with this root module and run the translator on it.

10 Finding the Refinement Mapping

To prove that the two-phase handshake implements the specification `Spec` of module `Alternation`, we must add to module `TwoPhase` the statement

```
A == INSTANCE Alternation WITH pc <- ...
```

for some expression "...", and then prove:

```
THEOREM  Spec => A!Spec
```

I like to give the expression "..." the name `pcBar`. So, add this to module `TwoPhase`:

```
pcBar ==

A  ==  INSTANCE Alternation WITH pc <- pcBar
```

and let's see how to fill in the definition of `pcBar`.

We must define `pcBar` so that in any execution of the two-phase algorithm, the values of `s` and `pcBar` are the values of `s` and `pc` allowed by specification `Spec` of module `Alternation`. Of course, this means that the value of `pcBar` in any reachable state of the two-phase algorithm must be either `"a"` or `"b"`.

The trick to writing the definition is to realize that `pcBar` must change when the spec of `Alternation` allows `pc` to change. Since the `Alternation` algorithm changes `pc` at the same time as it changes `s`, this means that the value of `pcBar` must be changed only by executing statements `a2` and `b2` of the two-phase algorithm.

It's clear that `pcBar` should equal `"a"` when process *pro* is at `a2`, and it should equal `"b"` when process *con* is at `b2`. Since its value must change when statement `a2` or `b2` is executed, we see that we must have:

- `pcBar = "a"` if `pc["a"] = "a2"` or `pc["b"] = "b3"`.
- `pcBar = "b"` if `pc["b"] = "b2"` or `pc["a"] = "a3"`.

This tells us the value of `pcBar` in all cases except when `pc["a"] = "a1"` and `pc["b"] = "b1"` are both true. In executing the two-phase algorithm, you should have observed that in this case `s := P(s)` is the next assignment to `s` executed if and only if `p = c` is true.

It's now straightforward to define `pcBar`. Most programmers would probably write it with nested `IF` / `THEN` / `ELSE` expressions. I like this definition because it's symmetric in the two processes:

```
pcBar == CASE pc["a"] = "a2" \/ pc["b"] = "b3" -> "a"
           [] pc["b"] = "b2" \/ pc["a"] = "a3" -> "b"
           [] OTHER -> IF p=c THEN "a" ELSE "b"
```

Put this definition in the module and use TLC to check the theorem. Create a model that checks the temporal property `A!Spec`. The theorem should hold for any values of the constants `s0`, `P`, and `C`. Those values should be chosen so there aren't too many reachable states. The order of execution of the statements doesn't depend on the value of `s`. So, to check that the theorem holds, it suffices to check it for some values of the constants so that, in any reachable state, (i) `C(s) /= P(s)` and (ii) `C(s)` and `P(s)` are unequal to `s`. (Condition (ii) prevents assignments to `s` from implementing stuttering steps allowed by `A!Spec`.) Let's use these values:

```
s0   <- 0
C(x) <- (x + 1) % 3
P(x) <- (x - 1) % 3
```

TLC checks that the property `A!Spec` is satisfied, so there's no need to prove the theorem. But, of course, we'll prove it anyway.

11 The Proof

11.1 The Proof Rule

We need to prove the theorem `Spec => A!Spec` in module `TwoPhase`. Since `A!Spec` equals `A!Init /\ [][A!Next]_A!vars`, the obvious proof rule to use is:

```
          THEOREM  Init => A!Init
      and THEOREM  [Next]_vars => [A!Next]_A!vars
    imply THEOREM  Init /\ [][Next]_vars  =>
                      A!Init /\ [][A!Next]_A!vars
```

In checking the theorem `Spec => A!Spec`, we checked that the first theorem is true. However, the second theorem is not true. It asserts a property of all steps $\langle t, u \rangle$, including steps in which t is an unreachable state. Suppose t is a state in which process *pro* is at statement `a2` and process *con* is at statement `b2`. (This state is not reachable.) Let u_p be the state obtained by executing `a2` and let u_c be the state obtained by executing `b2`. The steps $\langle t, u_p \rangle$ and $\langle t, u_c \rangle$ both satisfy `Next`, but they can't both satisfy `A!Next` if the values of `s` in states u_p and u_c are different.

The rule is valid as long as the second theorem's formula is true on all steps $\langle t, u \rangle$ in which t and u are reachable states. We can assert that a state is reachable by asserting that it satisfies an invariant of the specification. This observation leads to the following valid proof rule:

Safety Implementation Rule

```
      THEOREM  Init => A!Init
  and THEOREM  Init /\ [][Next]_vars => []Inv
  and THEOREM  Inv /\ Inv' /\ [Next]_vars => [A!Next]_A!vars
imply THEOREM  Init /\ [][Next]_vars
                  =>  THEOREM  A!Init /\ [][A!Next]_A!vars
```

11.2 Finding the Invariant

Let's use TLC to help us find the invariant `Inv`. The first thing to try is the type-correctness invariant. Type correctness of `p` and `c` obviously assert that they're in `{0,1}`. The usual type invariant for a function like `pc` is that it's in

```
[{"a", "b"} -> {"a1", "a2", "a3", "b1", "b2", "b3"}]
```

But it's obviously important that control in each process is always at a label in that process. So, we define this type invariant:

```
TypeOK ==
  /\ p \in {0, 1}
  /\ c \in {0, 1}
  /\ pc \in [{"a", "b"} -> {"a1", "a2", "a3", "b1", "b2", "b3"}]
  /\ pc["a"]  \in {"a1", "a2", "a3"} /\ pc["b"]  \in {"b1", "b2", "b3"}
```

Since we make no assumptions about `s0`, `P`, and `C`, we can make no type assertion about `s`.

Add this definition of `TypeOK` to module `TwoPhase`. We want to have TLC check

```
THEOREM  TypeOK /\ TypeOK' /\ [Next]_vars => [A!Next]_A!vars
```

If the conjunct `TypeOK'` were not there, we could use the same trick we used for checking inductive invariance and have TLC check:

```
THEOREM  TypeOK /\ [][Next]_vars => [][A!Next]_A!vars
```

Since `TypeOK /\ UNCHANGED vars` obviously implies `TypeOK'`, it's not hard to see that we can check the desired theorem by having TLC check:

```
THEOREM  TypeOK /\ [][Next /\ TypeOK']_vars => [][A!Next]_A!vars
```

Alternatively, we can observe that `TypeOK` is actually an inductive invariant of `Spec`. This means that `TypeOK /\ [Next]_vars` implies `TypeOK'`, so the `TypeOK'` conjunct in the theorem is redundant and can be omitted. Since it's simpler, let's do that. Have TLC check

```
THEOREM  TypeOK /\ [][Next]_vars => [][A!Next]_A!vars
```

by creating a model with initial predicate `TypeOK`, next-state relation `Next`, and `[][A!Next]_A!vars` as a temporal property to be checked.

Running TLC on this model produces an error when computing the initial state. The error message indicates that the value of `s` is `null`, meaning that the initial predicate hasn't assigned a value to `s`. We must add a conjunct to the initial predicate to specify the type of `s`. We can't do that for the spec, but we can do it for the model. For the values of `s0`, `P`, and `C` that we've chosen, that conjunct is `s \in 0..2`. Add this conjunct to the initial predicate in the model and run TLC again. It reports that the property `[][A!Next]_A!vars` is violated.

To help see what's going wrong, use the Error-Trace Explorer to show the value of `pcBar` in the error trace. One possible error trace, which is the one you'll get if you run TLC with a single thread, starts with `p` and `c` equal to 0, process *pro* at `a1`, and process *con* at `b2`; and it executes statement `a1`. This changes `pcBar` from `"b"` to `"a"` but leaves `s` unchanged.

The first state in the error trace is not reachable because process *con* can't be at `b2` when `p` and `c` are equal. When *con* is not at `b1`, the values of `p` and `c` must be unequal. Similarly, when *pro* is not at `a1`, the values of `p` and `c` must be equal. This suggests trying this definition for our invariant:

```
Inv == /\ TypeOK
       /\ (pc["a"] /= "a1") => (p = c)
       /\ (pc["b"] /= "b1") => (p /= c)
```

Add it to the module. Change the model to use `Inv /\ (s \in 0..2)` as the initial predicate and `Next /\ Inv'` as the next-state relation, and run TLC on it. TLC reports no error, so this definition of `Inv` will work—if it's an invariant. If it isn't, we can try to strengthen it to make it invariant—which will ensure that `Inv /\ [Next /\ Inv']_vars` still implies `[A!Next]_A!vars`.

Running TLC shows that `Inv` is an invariant of `Spec`. To prove that it's invariant, we need to find an inductive invariant that implies it. We start by using TLC to check if the invariant `Inv` itself is inductive. We do this by making `Inv` an invariant to be checked in the model with

`Inv /\ (s \in 0..2)` as its initial predicate and `Next` as its next-state relation . TLC finds no error, so `Inv` is an inductive invariant.

We know that `Inv` satisfies the required hypotheses of the Safety Implementation Rule because this example is small enough so TLC can essentially check those hypotheses for the actual specification rather than for just a small model of it. (Correctness of the algorithm doesn't depend on the values of the constants, and our invariant doesn't mention the variable `s` , so checking it on properly chosen values of the constants is good enough.) In a real spec, we could do this kind of checking only on a model that's too small to give us much confidence that `Inv` satisfies those hypotheses. The method in Using TLC to Check Inductive Invariance, mentioned at the end of Section 5 that checks for small, randomly chosen sets of type-correct states of a larger model can be used here as well.

11.3 Writing the Proof

When we've used TLC to check as best we can that the hypotheses of the Safety Implementation Rule are true, we can start writing a proof. Begin by either adding an `INSTANCE TLAPS` statement to the module or adding `TLAPS` to the module's `EXTENDS` statement.

Because the invariance of `Inv` is an interesting property of the algorithm, it's best to prove it as a separate theorem. So, add this to the module:

```
THEOREM  Invariance  ==  Spec => []Inv
```

We give the theorem the name `Invariance` so it can be used in our implementation proof. Since `Inv` is an inductive invariant of `Spec` , the high-level proof of this theorem is:

```
<1>1. Init => Inv
<1>2. Inv /\  [Next]_vars => Inv'
<1>3. QED
  BY <1>1, <1>2, PTL DEF Spec
```

Invariance proofs were discussed in Part I, so we will not examine this proof.

The theorem asserting that the two-phase handshake algorithm implements the specification `Spec` of module `Alternation` , and its high-level proof, are in Figure 8. Because the definition of `pcBar` will be used throughout the proof, we add the `USE` step that tells TLAPS to use that definition as if it appeared in the `DEF` clause of all `BY` proofs. We can tell TLAPS not to use the definition in some part of the proof with the step

```
HIDE DEF pcBar
```

```
THEOREM Spec => A!Spec
<1> USE DEF pcBar
<1>1. Init => A!Init
<1>2. Inv /\ Inv' /\ [Next]_vars => [A!Next]_A!vars
<1>3. QED
  BY <1>1, <1>2, Invariance, PTL DEF Spec, A!Spec
```

Figure 8: High-level implementation proof of the two-phase handshake.

A `USE` or `HIDE` step is local to the current subproof and obeys the obvious scoping rules.

The identifier `Invariance` in the `BY` proof of step `<1>3` tells TLAPS to use the theorem with that name.

For most examples, the proof of step `<1>1` is straightforward. For this simple example, you and TLAPS should be able prove it without having to decompose the proof. For more complicated examples, you can prove each conjunction of `A!Init` as a separate step. This should be done with the Toolbox's Decompose Proof command.[5]

The interesting part of the proof is the proof of step `<1>2`. The same sort of decomposition we used in Section 6.2.2 to decompose the proof of step `<1>2` of the invariance proof leads in this case to the decomposition of Figure 9.

The easiest step is `<2>7`, so let's look at it first. A stuttering step of algorithm `TwoPhase` should implement a stuttering step of the `Alternation` spec. Therefore, `UNCHANGED vars` should imply `UNCHANGED A!vars`, which implies `[A!Next]_A!vars`. (The latter implication is trivial, since $[A]_v$ equals $A \vee (\text{UNCHANGED } v)$ for any A and v.) The `INSTANCE` statement defines `A!vars` to equal `<<pcBar, s>>`, so it's easy to see that leaving the three variables in the triple `vars` unchanged leaves both components of the pair `A!vars` unchanged. This is obvious, and TLAPS should prove `<2>7` with the proof

```
    BY <2>7 DEF vars, A!vars
```

[5] A bug in versions through 1.5.7 of the Toolbox causes the command to produce garbage when the formula being proved contains a defined symbol such as `A!Init` that is imported with an `INSTANCE` statement containing implicit instantiation of module parameters. The workaround is to make the instantiation of all module parameters explicit—in this case by changing the `INSTANCE` statement that imports module `Alternation` to

```
    A == INSTANCE Alternation WITH pc <- pcBar, s <- s, s0 <- s0
```

This bug has been fixed in version 1.5.8 which, as I write this, has not yet been released.

```
<2> SUFFICES ASSUME Inv,
                    Inv',
                    [Next]_vars
             PROVE  [A!Next]_A!vars
  OBVIOUS
<2>1. CASE a1
<2>2. CASE a2
<2>3. CASE a3
<2>4. CASE b1
<2>5. CASE b2
<2>6. CASE b3
<2>7. CASE UNCHANGED vars
<2>8. QED
  BY <2>1, <2>2, <2>3, <2>4, <2>5, <2>6, <2>7 DEF con, Next, pro
```

Figure 9: The decomposition of step `<1>2`.

The `<2>7` in this `BY` proof tells TLAPS to use the step's case assumption. As we've seen, TLAPS does not use named facts by default. The `CASE` step `<2>7` is an abbreviation for

```
<2>7. ASSUME UNCHANGED vars
      PROVE  [A!Next]_A!vars
```

and within a named `ASSUME`/`PROVE` step's proof, the step's name names its assumptions.

Let's now look at the proofs of steps `<2>1`–`<2>6`. TLAPS can probably prove each of them with a single `BY` proof, but let's suppose we want to prove them by ourselves. We'll start with `<2>1`.

Since an `a1` step leaves `s` unchanged, we expect `a1` to imply that `A!vars` is unchanged. This leads to two possible ways to decompose the proof. The first is

```
<3>1. UNCHANGED A!vars
<3>2. QED
  BY <3>1
```

The second is:

```
<3> SUFFICES UNCHANGED A!vars
  OBVIOUS
...
```

The second is better if we need to further decompose the proof, since the decomposition is done with level `<3>` steps while the first way uses a level `<4>` proof. If this were a complex algorithm, we might want to check that the step is actually true and `a1` does imply `UNCHANGED A!vars`. We can do this by having TLC check

```
THEOREM  Inv /\ [][a1]_vars  =>  [][FALSE]_A!vars
```

This is done in essentially the same way we used TLC to check step `<1>2`. Since `A!vars` equals `<<pcBar, s>>`, we can decompose the proof of `<2>1` as

```
<3> SUFFICES UNCHANGED A!vars
  OBVIOUS
<3>1. pcBar' = pcBar
<3>2. s' = s
<3>3. QED
  BY <3>1, <3>2 DEF a1, A!vars
```

Step `<3>2` follows immediately from the `<2>1` case assumption and the definition of `a1`. Step `<3>1` is more subtle. Here is a very detailed hand proof.

```
<4>1. /\ pc["a"] = "a1"
      /\ p = c
  (**********************************************************)
  (* By a1, which is the <2>1 case assumption.              *)
  (**********************************************************)
<4>2. pc'["a"] = "a2"
  (**********************************************************)
  (* By a1 and TypeOK, which is implied by the level <2>    *)
  (* SUFFICES assumption Inv.                               *)
  (**********************************************************)
<4>3. pc["b"] = "b1"
  (**********************************************************)
  (* By Inv and p=c, which is asserted by <4>1.             *)
  (**********************************************************)
<4>4. QED
  <5>1. pcBar = "a"
    (********************************************************)
    (* By <4>1, <4>3, and the definition of pcBar.          *)
    (********************************************************)
  <5>2. pcBar' = "a"
    (********************************************************)
    (* By <4>2 and the definition of pcBar.                 *)
    (********************************************************)
```

```
      <5>3. QED
        BY <5>1, <5>2
```

Note that the type-correctness hypothesis TypeOK is needed to prove `<4>2` because we can deduce `pc'["a"]="a2"` from

```
pc' = [pc EXCEPT !["a"] = "a2"]
```

only if `pc` is a function whose domain contains `"a"`.

This finishes the proof sketch for step `<2>1`. Like action `a1`, the actions `a3`, `b1` and `b3` imply `UNCHANGED A!vars`. The proofs of steps `<2>3` `<2>4`, and `<2>6` are therefore similar to that of `<2>1`. Let's examine the proof of `<2>2`.

Action `a2` of the algorithm `TwoPhase` implements action `a` of the `Alternation` spec. We can therefore decompose the proof of `<2>2` as:

```
<3>1. A!a
<3>2. QED
  BY <3>1 DEF A!Next
```

I have written it this way instead of with a `SUFFICES` step because I want to decompose the proof of `A!a`, and the Decompose Proof command doesn't decompose a goal introduced with `SUFFICES`. Applying that command to step `<3>1` yields this proof:

```
<4>1. pcBar = "a"
<4>2. s' = P(s)
<4>3. pcBar' = "b"
<4>4. QED
  BY <4>1, <4>2, <4>3 DEF A!a
```

Steps `<4>1` and `<4>2` follow directly from the case assumption `a2`. Here's the sketch of a proof of `<4>3`:

```
<5>1. pc'["a"] = "a3"
  (**********************************************************)
  (* By a2 (the <2>2 case assumption), which implies        *)
  (*                                                        *)
  (*    pc' = [pc EXCEPT !["a"] = "a3"]                     *)
  (*                                                        *)
  (* and TypeOK (which is implied by Inv, a level <2>       *)
  (* SUFFICES assumption).                                  *)
  (**********************************************************)
```

```
    <5>2. pc'["b"] = "b1"
      (***********************************************************)
      (* <5>1 and Inv' (which holds by the level <2> SUFFICES     *)
      (* assumption) imply p' = c', which by Inv' implies <5>2.   *)
      (***********************************************************)
    <5>3. QED
      BY <5>1, <5>2 DEF pcBar
```

This finishes the proof of step `<2>2`.

Action `b2` implies `A!b`, and the proof of step `<2>5` is similar to that of step `<2>2`. This completes our sketch of the proof of step `<1>2`.

12 Conclusion

The proofs in Sections 6 and 11 are excruciatingly detailed. There's no reason for a human to check such proofs because TLAPS can do it. For algorithms not as trivial as our examples, proving some steps requires human thought. Before you can prove such a step, either by hand or with TLAPS, you need to understand why it's true. And that requires writing a proof—one that's detailed enough and rigorous enough to convince a person.

The kind of proof written by most mathematicians isn't good enough. The paper *How to Write a 21st Century Proof* describes why mathematicians should write structured proofs and how they can do it. Proofs of properties of TLA$^+$ specifications differ from mathematicians' proofs because the statement of what has to be proved, as well as the steps in the proof, can be written as precise mathematical formulas. This permits the degree of rigor needed to avoid the tiny mistakes in a proof that can mask serious bugs in an algorithm.

You have now seen how to write such proofs for trivial examples. With practice, it will feel like the natural way to prove correctness of real algorithms. Later, you may want to try writing completely formal proofs checked by TLAPS.

References

[1] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[2] R. W. Floyd. Assigning meanings to programs. In *Proceedings of the Symposium on Applied Math., Vol. 19*, pages 19–32. American Mathematical Society, 1967.

[3] Kourosh Gharachorloo, Madhu Sharma, Simon Steely, and Stephen Van Doren. Architecture and design of AlphaServer GS320. In Anoop Gupta, editor, *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 13–24, November 2000.