TLA$^+$ Proofs*

Denis Cousineau[1], Damien Doligez[2], Leslie Lamport[3], Stephan Merz[4], Daniel Ricketts[5], and Hernán Vanzetto[4]

[1] Inria - Université Paris Sud, Orsay, France. ***
[2] Inria, Paris, France
[3] Microsoft Research, Mountain View, CA, U.S.A.
[4] Inria Nancy & LORIA, Villers-lès-Nancy, France
[5] Department of Computer Science, University of California, San Diego, U.S.A.

Abstract. TLA$^+$ is a specification language based on standard set theory and temporal logic that has constructs for hierarchical proofs. We describe how to write TLA$^+$ proofs and check them with TLAPS, the TLA$^+$ Proof System. We use Peterson's mutual exclusion algorithm as a simple example to describe the features of TLAPS and show how it and the Toolbox (an IDE for TLA$^+$) help users to manage large, complex proofs.

1 Introduction

TLA$^+$ [10] is a specification language originally designed for specifying concurrent and distributed systems and their properties. Specifications and properties are written as formulas of TLA, a linear-time temporal logic. TLA$^+$ is based on TLA and Zermelo-Fraenkel set theory with the axiom of choice; it also adds a module system for structuring specifications. More recently, constructs for writing proofs have been added to TLA$^+$; these are derived from a hierarchical presentation of natural-deduction proofs proposed for writing rigorous hand proofs [14].

In this paper, we present the main ideas that guided the design of the proof language and our experience with using the TLA$^+$ tools for verifying safety properties of TLA$^+$ specifications. The TLA$^+$ Toolbox is an integrated development environment (IDE) based on Eclipse for writing TLA$^+$ specifications and running the TLA$^+$ tools on them, including the TLC model checker and TLAPS, the TLA$^+$ proof system [5, 19]. In particular, it provides commands to hide and unhide parts of a proof, allowing a user to focus on a given proof step and its context. It is also invaluable to be able to run the model checker on the same formulas that one reasons about.

The TLA$^+$ proof language and TLAPS have been designed to be independent of any particular theorem prover. All interaction takes place at the level of TLA$^+$, letting the user focus on the specification of the algorithm being developed. We

*** This work was partially funded by INRIA-Microsoft Research Joint Centre, France.

do not expect users to have precise knowledge of the inner workings of the back-end provers that TLAPS uses, although with experience users learn about the strengths and weaknesses of the different provers—for example, that SMT solvers excel at arithmetic.

TLAPS has a *Proof Manager* (PM) that transforms a proof into individual proof obligations that it sends to back-end provers. Currently, the main back-end provers are Isabelle/TLA$^+$, an encoding of TLA$^+$ as an object logic in Isabelle [22], Zenon [4], a tableau prover for classical first-order logic with equality, and a back-end for SMT solvers. Isabelle serves as the most trusted back-end prover, and when possible, we expect back-end provers to produce a detailed proof that is checked by Isabelle. This is currently implemented for the Zenon back-end, which can export its proofs as Isar scripts that Isabelle can certify.

We explain how to write and check TLA$^+$ proofs, using a tiny well-known example: a proof that Peterson's algorithm [18] implements mutual exclusion. We start by writing the algorithm in PlusCal [11], an algorithm language that is based on the expression language of TLA$^+$. The PlusCal code is translated to a TLA$^+$ specification, which is what we reason about. Section 3 introduces the salient features of the proof language and of TLAPS with the proof of mutual exclusion. Liveness of Peterson's algorithm (processes eventually enter their critical section) can also be asserted and proved with TLA$^+$. However, liveness reasoning makes full use of temporal logic, and TLAPS cannot yet check temporal logic proofs. We therefore discuss only mutual exclusion.

Section 4 describes how TLA$^+$, TLAPS, and the Toolbox scale to realistic examples. Their relation to other proof systems is discussed in Section 5. A concluding section summarizes what we have done and our plans for future work.

2 Modeling Peterson's Algorithm In TLA$^+$

Peterson's algorithm is a classic, very simple two-process mutual exclusion algorithm. We specify the algorithm in TLA$^+$ and prove that it satisfies mutual exclusion, meaning that no two processes are in their critical sections at the same time.[6]

2.1 From PlusCal To TLA$^+$

We will write Peterson's algorithm in the PlusCal algorithm language. To do so, we have the Toolbox create an empty TLA$^+$ module. We name the two processes 0 and 1, and we define an operator Not so that $Not(0) = 1$ and $Not(1) = 0$:

$$Not(i) \triangleq \text{IF } i = 0 \text{ THEN } 1 \text{ ELSE } 0$$

[6] The TLA$^+$ module containing the specification and proof is accessible at the TLAPS Web page [19].

```
--algorithm Peterson {
  variables flag = [i ∈ {0, 1} ↦ FALSE], turn = 0;
  process (proc ∈ {0, 1}) {
    a0: while (TRUE) {
    a1:   flag[self] := TRUE;
    a2:   turn := Not(self);
    a3a:  if (flag[Not(self)]) {goto a3b} else {goto cs} ;
    a3b:  if (turn = Not(self)) {goto a3a} else {goto cs} ;
    cs:   skip; \* critical section
    a4:   flag[self] := FALSE;
        } \* end while
  } \* end process
} \* end algorithm
```

Fig. 1. Peterson's algorithm in PlusCal.

The PlusCal code for Peterson's algorithm is shown in Figure 1; it appears in a comment in the TLA$^+$ module.[7] The **variables** statement declares the variables and their initial values. For example, the initial value of *flag* is an array such that $flag[0] = flag[1] = \text{FALSE}$. (Mathematically, an array is a function; the TLA$^+$ notation $[x \in S \mapsto e]$ for writing functions is similar to a lambda expression.) To specify a multiprocess algorithm, it is necessary to specify what its atomic actions are. In PlusCal, an atomic action consists of the execution from one label to the next. With this brief explanation, the reader should be able to figure out what the code means.

The PlusCal translator, accessible through a Toolbox menu, generates a TLA$^+$ specification from the PlusCal code of the algorithm. Figure 2 gives the generated TLA$^+$ translation.[8] The PlusCal compiler adds a variable *pc*, which explicitly records the control state of each process. For example, control in process i is at *cs* iff $pc[i]$ equals the string "`cs`".

The heart of the TLA$^+$ specification consists of the initial predicate *Init*, which describes the initial state, and the next-state relation *Next*, which describes how the state can change. Given the PlusCal code, the meaning of formula *Init* in the figure is straightforward. The formula *Next* is a predicate on old-state/new-state pairs. Unprimed variables refer to the old state and primed variables to the new state. Formula *Next* is the disjunction of the two formulas $proc(0)$ and $proc(1)$, and each $proc(self)$ is the disjunction of seven formulas—one for each label in the body of the **process**. The formula $a0(self)$ specifies the state change performed by process *self* executing an atomic action starting

[7] The figure shows the pretty-printed version of PlusCal code and TLA$^+$ formulas. As an example of how they are typed, here is the ASCII version of the **variables** declaration: `variables flag = [i \in {0, 1} |-> FALSE], turn = 0;`

[8] For clarity of presentation, we have simplified the translation slightly by "in-lining" a definition. The proof we develop works for the unmodified translation if we add a global declaration that causes the definition to be expanded throughout the proof.

VARIABLES $flag$, $turn$, pc

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

$$
\begin{aligned}
Init \triangleq\ & \land flag = [i \in \{0, 1\} \mapsto \text{FALSE}] \\
& \land turn = 0 \\
& \land pc = [self \in \{0, 1\} \mapsto \text{“a0”}]
\end{aligned}
$$

$$
\begin{aligned}
a0(self) \triangleq\ & \land pc[self] = \text{“a0”} \\
& \land pc' = [pc \text{ EXCEPT } ![self] = \text{“a1”}] \\
& \land \text{UNCHANGED } \langle flag, turn \rangle
\end{aligned}
$$

$$
\begin{aligned}
a1(self) \triangleq\ & \land pc[self] = \text{“a1”} \\
& \land flag' = [flag \text{ EXCEPT } ![self] = \text{TRUE}] \\
& \land pc' = [pc \text{ EXCEPT } ![self] = \text{“a2”}] \\
& \land turn' = turn
\end{aligned}
$$

$$
\begin{aligned}
a2(self) \triangleq\ & \land pc[self] = \text{“a2”} \\
& \land turn' = Not(self) \\
& \land pc' = [pc \text{ EXCEPT } ![self] = \text{“a3a”}] \\
& \land flag' = flag
\end{aligned}
$$

$$
\begin{aligned}
a3a(self) \triangleq\ & \land pc[self] = \text{“a3a”} \\
& \land \text{IF } flag[Not(self)] \\
& \quad \text{THEN } pc' = [pc \text{ EXCEPT } ![self] = \text{“a3b”}] \\
& \quad \text{ELSE } pc' = [pc \text{ EXCEPT } ![self] = \text{“cs”}] \\
& \land \text{UNCHANGED } \langle flag, turn \rangle
\end{aligned}
$$

$$
\begin{aligned}
a3b(self) \triangleq\ & \land pc[self] = \text{“a3b”} \\
& \land \text{IF } turn = Not(self) \\
& \quad \text{THEN } pc' = [pc \text{ EXCEPT } ![self] = \text{“a3a”}] \\
& \quad \text{ELSE } pc' = [pc \text{ EXCEPT } ![self] = \text{“cs”}] \\
& \land \text{UNCHANGED } \langle flag, turn \rangle
\end{aligned}
$$

$$
\begin{aligned}
cs(self) \triangleq\ & \land pc[self] = \text{“cs”} \\
& \land pc' = [pc \text{ EXCEPT } ![self] = \text{“a4”}] \\
& \land \text{UNCHANGED } \langle flag, turn \rangle
\end{aligned}
$$

$$
\begin{aligned}
a4(self) \triangleq\ & \land pc[self] = \text{“a4”} \\
& \land flag' = [flag \text{ EXCEPT } ![self] = \text{FALSE}] \\
& \land pc' = [pc \text{ EXCEPT } ![self] = \text{“a0”}] \\
& \land turn' = turn
\end{aligned}
$$

$$
\begin{aligned}
proc(self) \triangleq\ & a0(self) \lor a1(self) \lor a2(self) \lor a3a(self) \lor a3b(self) \\
& \lor cs(self) \lor a4(self)
\end{aligned}
$$

$Next \triangleq \exists\, self \in \{0, 1\} : proc(self)$

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

Fig. 2. A pretty-printed version of the TLA$^+$ translation, slightly simplified.

at label $a0$, and similarly for the other six labels. (If f is a function, the TLA$^+$ notation $[f \text{ EXCEPT } ![arg] = exp]$ denotes the function that is equal to f except that it maps arg to exp.) The reader should be able to figure out the meaning of the TLA$^+$ notation and of formula *Next* by comparing these seven definitions with the corresponding PlusCal code.

The temporal formula *Spec* is the complete specification. It is satisfied by a behavior (i.e., an ω-sequence of states) iff the behavior starts in a state satisfying *Init* and each of its steps (pairs of successive states) either satisfies *Next* or else leaves the values of the three variables *flag*, *turn*, and *pc* unchanged.[9] The $\Box$ is the ordinary *always* operator of linear-time temporal logic, and $[Next]_{vars}$ is an abbreviation for $Next \lor \text{UNCHANGED } vars$, where UNCHANGED *vars* is an abbreviation for $vars' = vars$ and priming an expression means priming all the variables that occur in it.

2.2 Validation Through Model Checking

Before trying to prove that the algorithm is correct, we use TLC, the TLA$^+$ model checker, to check it for errors. We first instruct the Toolbox to have TLC check for "execution errors".[10] What are type errors in typed languages are one source of execution errors in TLA$^+$.

The Toolbox runs TLC on a model of a TLA$^+$ specification. A model usually assigns particular values to specification constants, such as the number N of processes. It can also restrict the set of states explored, which is useful if the specification allows an infinite number of reachable states. For this trivial example, there are no constants to specify and only 58 reachable states. TLC finds no execution errors.

We next check if the algorithm actually satisfies mutual exclusion. Since we made execution of the critical section an atomic action, mutual exclusion means that the two processes never both have control at label *cs*. Mutual exclusion therefore holds iff the following predicate *MutualExclusion* is an invariant of the algorithm—meaning that it is true in all reachable states:

$$MutualExclusion \triangleq (pc[0] \neq \text{“cs”}) \lor (pc[1] \neq \text{“cs”})$$

TLC reports that the algorithm indeed satisfies this invariant. Peterson's algorithm is so simple that TLC has checked that all possible executions satisfy mutual exclusion. For more interesting algorithms that have an infinite set of reachable states, TLC is no longer able to exhaustively verify all executions, and correctness must be proved deductively. Still, TLC is invaluable for catching errors in the algorithm or its formal model: the effort required for running TLC is incomparably lower than that for writing a formal proof.

[9] "Stuttering steps" that leave all variables unchanged are allowed in order to make refinement simple [9].

[10] The translation is a temporal logic formula, so there is no obvious definition of an execution error. An execution error occurs in a behavior if whether or not the behavior satisfies the formula is not specified by the semantics of TLA$^+$—for example, because the semantics do not specify whether or not 0 equals FALSE.

THEOREM $Spec \Rightarrow \Box MutualExclusion$
$\langle 1 \rangle 1$. $Init \Rightarrow Inv$
$\langle 1 \rangle 2$. $Inv \wedge [Next]_{vars} \Rightarrow Inv'$
$\langle 1 \rangle 3$. $Inv \Rightarrow MutualExclusion$
$\langle 1 \rangle 4$. QED

Fig. 3. The high-level proof.

3 Proving Mutual Exclusion For Peterson's Algorithm

We now describe a deductive correctness proof of Peterson's algorithm in TLA$^+$. Proofs of more interesting algorithms follow the same basic structure, but they are longer. Section 4 describes how TLA$^+$ proofs scale to larger algorithms.

3.1 The High-Level Proof

The assertion that Peterson's algorithm implements mutual exclusion is formalized in TLA$^+$ as:

THEOREM $Spec \Rightarrow \Box MutualExclusion$

The standard method of proving this invariance property is to find an inductive invariant *Inv* that implies *MutualExclusion*. An inductive invariant is one that is true in the initial state and whose truth is preserved by the next-state relation. TLA$^+$ proofs are hierarchically structured and are generally written top-down. The top level of this invariance proof is shown in Figure 3. Step $\langle 1 \rangle 2$ asserts that the truth of *Inv* is preserved by the next-state relation.

Each proof in the hierarchy ends with a QED step that asserts the goal of that proof, the QED step for the top level asserting the statement of the theorem. We usually write the QED step's proof first. This QED step follows easily from $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, and $\langle 1 \rangle 3$ by propositional logic and the following two temporal-logic proof rules:

$$\frac{I \wedge [N]_v \Rightarrow I'}{I \wedge \Box[N]_v \Rightarrow \Box I} \qquad \frac{P \Rightarrow Q}{\Box P \Rightarrow \Box Q}$$

However, TLAPS does not yet handle temporal reasoning, so we omit the proof of the QED step. When temporal reasoning is added to TLAPS, we expect it easily to check such a trivial proof.

To continue the proof, we must define the inductive invariant *Inv*. (A definition must precede its use, so the definition of *Inv* appears in the module before the proof.) Figure 4 defines *Inv* to be the conjunction of two formulas. The first, *TypeOK*, is a "type-correctness" invariant, asserting that the values of all variables are elements of the expected sets. (The expression $[S \to T]$ is the set of all functions whose domain is S and whose range is a subset of T.) In an untyped logic like that of TLA$^+$, almost any inductive invariant must assert type correctness. The second conjunct, I, is the interesting one that explains why Peterson's algorithm implements mutual exclusion.

$$
\begin{array}{l}
\mathit{TypeOK} \triangleq \land\ \mathit{pc} \in [\,\{0,1\} \to \{\text{“a0”}, \text{“a1”}, \text{“a2”}, \text{“a3a”}, \text{“a3b”}, \text{“cs”}, \text{“a4”}\}\,] \\
\qquad\qquad \land\ \mathit{turn} \in \{0,1\} \\
\qquad\qquad \land\ \mathit{flag} \in [\,\{0,1\} \to \text{BOOLEAN}\,] \\
I \triangleq \forall i \in \{0,1\} : \\
\qquad \land\ \mathit{pc}[i] \in \{\text{“a2”}, \text{“a3a”}, \text{“a3b”}, \text{“cs”}, \text{“a4”}\} \Rightarrow \mathit{flag}[i] \\
\qquad \land\ \mathit{pc}[i] \in \{\text{“cs”}, \text{“a4”}\} \Rightarrow \land\ \mathit{pc}[\mathit{Not}(i)] \notin \{\text{“cs”}, \text{“a4”}\} \\
\qquad\qquad\qquad\qquad\qquad\qquad \land\ \mathit{pc}[\mathit{Not}(i)] \in \{\text{“a3a”}, \text{“a3b”}\} \Rightarrow \mathit{turn} = i \\
\mathit{Inv} \triangleq \mathit{TypeOK} \land I
\end{array}
$$

Fig. 4. The inductive invariant.

There is no point trying to prove that a formula is an inductive invariant if TLC can show that it's not even an invariant. So, we first run TLC to test if *Inv* is an invariant. In the simple case of Peterson's algorithm, TLC can check not only that it is an invariant, but that it is an inductive invariant. We check that *Inv* is an inductive invariant of *Spec* by checking that it is an (ordinary) invariant of the specification $\mathit{Inv} \land \Box[\mathit{Next}]_{\mathit{vars}}$, obtained from *Spec* by replacing the initial condition by *Inv*. In most real examples, TLC can at best check an inductive invariant on a tiny model—one that is too small to gain any confidence that it really is an inductive invariant. However, TLC can still often find simple errors in an inductive invariant.

3.2 Leaf Proofs for Steps ⟨1⟩1–⟨1⟩3

We now prove steps ⟨1⟩1–⟨1⟩3. We can prove them in any order; let us start with ⟨1⟩1. We expect this step to follow easily from the definitions of *Init* and *Inv* and simple properties of sets and functions. TLAPS knows about sets and functions, but it does not expand definitions unless directed to do so. (In complex proofs, automatically expanding definitions often leads to formulas that are too big for provers to handle.) We assert that the step follows from simple math and the definitions of *Init* and *Inv* by writing the following leaf proof immediately after the step:

BY DEF *Init*, *Inv*

We then tell the Toolbox to run TLAPS to check this proof. It does so and reports that the prover failed to prove the following obligation:

```
ASSUME NEW VARIABLE flag,
       NEW VARIABLE turn,
       NEW VARIABLE pc
PROVE  (/\ flag  =  [i \in {0, 1} |-> FALSE]
        /\ turn  =  0
        /\ pc  =  [self \in {0, 1} |-> "a0"])
        =>  TypeOK  /\  I
```

This obligation is exactly what TLAPS's back-end provers are trying to prove. They are given no other facts. In particular, the provers know nothing about

```
⟨1⟩2. Inv ∧ [Next]_vars ⇒ Inv′
  ⟨2⟩1. SUFFICES ASSUME Inv, Next
                 PROVE Inv′
  ⟨2⟩2. TypeOK′
  ⟨2⟩3. I′
  ⟨2⟩4. QED
```

Fig. 5. The top-level proof of ⟨1⟩2.

TypeOK and I, so they obviously can't prove the obligation. We have to tell TLAPS also to use the definitions of *TypeOK* and I. We do that by making the obvious change to the BY proof, after which TLAPS easily proves the step. Forgetting to expand some definitions is a common mistake, and looking at the formula displayed by the Toolbox usually reveals which definitions need to be invoked.

Step ⟨1⟩3 is proved the same way, by simply expanding the definitions of *MutualExclusion*, *Inv*, I, and *Not*. We next try the same technique on ⟨1⟩2. A little thought shows that we have to tell TLAPS to expand all the definitions in the module up to and including the definition of *Next*, except for the definition of *Init*. However, when we direct TLAPS to prove the step, it fails to do so, reporting a 65-line proof obligation.

TLAPS uses Zenon and Isabelle as its default back-end provers, first trying Zenon and then trying Isabelle if Zenon fails to find a proof. However, TLAPS also includes an SMT solver back-end [16] that is capable of handling larger "shallow" proof obligations—in particular, ones that do not contain significant quantifier reasoning. We instruct TLAPS to use the SMT back-end when proving the current step by writing

```
BY SMT DEF ...
```

The SMT back-end translates the proof obligation to SMT-LIB [3], the standard input language for different SMT solvers, and calls an SMT solver (CVC3 by default) to try to prove the resulting formula. CVC3 proves step ⟨1⟩2 in a few seconds. Variants of the SMT back-end translate to the native input languages of Yices and Z3, which sometimes perform better than does CVC3 using the standard SMT-LIB translation.

3.3 A Hierarchical Proof of Step ⟨1⟩2

For sufficiently complicated examples, an SMT solver will not be able to prove inductive invariance as a single obligation. The proof will have to be hierarchically decomposed. To illustrate how this is done, we now write a proof of ⟨1⟩2 that can be checked using only the Zenon and Isabelle back-end provers.

Step ⟨1⟩2 and its top-level proof appear in Figure 5. The first step in the proof of an implication like this would normally be:

```
⟨2⟩1. SUFFICES ASSUME Inv, [Next]_vars
               PROVE  Inv′
```

This step asserts that to prove the current goal, which is step $\langle 1\rangle 2$, it suffices to assume that *Inv* and $[Next]_{vars}$ are true and prove Inv'. The step also changes the goal of the rest of the level-2 proof to Inv' and allows the assumptions *Inv* and $[Next]_{vars}$ to be used in the rest of the proof. This step's assertion is obviously true, and TLAPS will check the one-word leaf proof OBVIOUS. However, the proof of Figure 5 does something a little different.

Since the assumption $[Next]_{vars}$ equals $Next \lor$ UNCHANGED *vars* , it leaves two cases to be proved: (i) *Next* is true and (ii) all variables are unchanged, so their primed values equal their unprimed values. The proof in the second case is trivial, and TLAPS should have no trouble checking it. In Figure 5, the assumption in the SUFFICES statement is *Next* rather than $[Next]_{vars}$, so the remainder of the proof only has to consider case (i). To show that it suffices to prove Inv' under this stronger assumption, the proof of that SUFFICES step has to prove case (ii).

The remainder of the level-2 proof is straightforward. Since *Inv* equals $TypeOK \land I$, the goal Inv' is the conjunction of the two formulas $TypeOK'$ and I'. We therefore decompose the proof by proving each conjunct separately. The proof of the QED step is simply

BY $\langle 2\rangle 2$, $\langle 2\rangle 3$ DEF *Inv*

Observe that we have to tell TLAPS exactly what facts to use as well as what definitions to expand.

We next prove $\langle 2\rangle 1$–$\langle 2\rangle 3$. Zenon proves $\langle 2\rangle 1$ when the definitions of *vars*, *Inv*, *TypeOK*, and *I* are expanded. Note that the definition of *Next* is not needed. To prove $\langle 2\rangle 2$ and $\langle 2\rangle 3$, we need to use the definition of *Next*—that is, with all definitions expanded down to TLA$^+$ primitives—as well as the definition of *Inv*. We also have to use the assumption that *Inv* and *Next* are true, introduced by step $\langle 2\rangle 1$. This leads us to try the following proof for $\langle 2\rangle 2$.

BY $\langle 2\rangle 1$ DEFS *Inv*, *TypeOK*, *Next*, *proc*, $a0$, $a1$, $a2$, $a3a$, $a3b$, *cs*, $a4$, *Not*

Instead of the reference to step $\langle 2\rangle 1$ in the BY clause, we could also name the required facts directly and write

BY *Inv*, *Next* DEFS ...

The proof manager checks that *Inv* and *Next* indeed follow from the currently available assumptions.

Zenon fails on this proof, but Isabelle succeeds. However, both Zenon and Isabelle fail on the corresponding proof of $\langle 2\rangle 3$ (which requires also using the definition of *I*). To prove it (with only Zenon and Isabelle), we need one more level of proof. That level appears in Figure 6, which contains the complete proof of the theorem.

Since priming a formula means priming all variables in it, the goal I' has the form $\forall i \in \{0, 1\} : exp(i)'$. A standard way to prove this formula is by $\forall$-introduction: we introduce a new variable, say j, we assume $j \in \{0, 1\}$, and we prove $exp(j)'$. TLA$^+$ provides a notation for naming subexpressions of a

```
THEOREM Spec ⇒ □MutualExclusion
⟨1⟩1. Init ⇒ Inv
  BY DEF Init, Inv, TypeOK, I
⟨1⟩2. Inv ∧ [Next]_vars ⇒ Inv′
  ⟨2⟩1. SUFFICES ASSUME Inv, Next
                 PROVE Inv′
    BY DEF vars, Inv, TypeOK, I
  ⟨2⟩2. TypeOK′
    BY ⟨2⟩1 DEF Inv, TypeOK, Next, proc, a0, a1, a2, a3a, a3b, cs, a4, Not
  ⟨2⟩3. I′
    ⟨3⟩1. SUFFICES ASSUME NEW j ∈ {0, 1}
                   PROVE  I!(j)′
      BY DEF I
    ⟨3⟩2. PICK i ∈ {0, 1} : proc(i)
      BY ⟨2⟩1 DEF Next
    ⟨3⟩3. CASE i = j
      BY ⟨2⟩1, ⟨3⟩2, ⟨3⟩3 DEF Inv, I, TypeOK, proc, a0, a1, a2, a3a, a3b,
                               cs, a4, Not
    ⟨3⟩4. CASE i ≠ j
      BY ⟨2⟩1, ⟨3⟩2, ⟨3⟩4 DEF Inv, I, TypeOK, proc, a0, a1, a2, a3a, a3b,
                               cs, a4, Not
    ⟨3⟩5. QED
      BY ⟨3⟩3, ⟨3⟩4
  ⟨2⟩4. QED
    BY ⟨2⟩2, ⟨2⟩3 DEF Inv
⟨1⟩3. Inv ⇒ MutualExclusion
  BY DEF MutualExclusion, Inv, I, Not
⟨1⟩4. QED
  PROOF OMITTED
```

Fig. 6. The complete proof.

definition. With that notation, the expression $exp(j)$ is written $I!(j)$. This leads us to begin the proof of $\langle 2\rangle 3$ with the SUFFICES step $\langle 3\rangle 1$ of Figure 6 and its simple proof.

The assumption *Next* (introduced by $\langle 2\rangle 1$) equals $\exists self \in \{0, 1\} : proc(self)$. A standard way to use such an assumption is by $\exists$-elimination: we pick some value of *self* such that $proc(self)$ is true. That is what step $\langle 3\rangle 2$ does, naming the value i.

We simplified our task to proving $I!(j)'$ instead of I', using $proc(i)$ instead of *Next*, which eliminates two quantifiers. However, Zenon and Isabelle still cannot prove the goal in a single step. The usual way to decompose the proof that process i preserves an invariant is to show that each separate atomic action of process i preserves the invariant. In mathematical terms, $proc(i)$ is the disjunction of the seven formulas $a0(i), \ldots, a4(i)$, each describing one of the process's atomic action. We can decompose the proof by considering each of the seven formulas as a separate case.

While this is the usual procedure, Peterson's algorithm is simple enough that it is not necessary. Instead, we just have to help the back-end provers by splitting the proof into the two cases of $i = j$ and $i \neq j$. The reader can see how this is done in Figure 6. Observe that in the proof of CASE statement $\langle 3 \rangle 3$, the name $\langle 3 \rangle 3$ refers to the case assumption $i = j$. There is no explicit use of $\langle 3 \rangle 1$ because a NEW assumption in an ASSUME is used by default in all proofs in the assumption's scope. The same is true of the formula $i \in \{0, 1\}$ asserted by the PICK step. (This is a pragmatic choice in the design of TLAPS, based on the observation that such facts are used so often.)

4 Writing Real Proofs

We have described how one writes and checks a TLA$^+$ proof of a tiny example. Several larger case studies have been carried out using the system. These include verifications of Byzantine Paxos [13], the Memoir security architecture [17], and the lookup and join protocols of the Pastry algorithm for maintaining a distributed hashtable over a peer-to-peer network [15]. TLA$^+$ and TLAPS, with its Toolbox interface, provide a number of features that help manage the complexity of large proofs.

4.1 Hierarchical Proofs And The Proof Manager

The most important aid in writing large proofs is TLA$^+$'s hierarchical and declarative proof language, where intermediate proof obligations are stated explicitly. While declarative proofs are more verbose than standard tactic scripts, they are easier to understand and maintain because the information on what is currently being proved is available at each point. Hierarchical proofs enable a user to keep decomposing a complex proof into smaller steps until the steps become provable by one of the back-end provers.

In logical terms, proof steps correspond to natural-deduction sequents whose validity must be established in the current context (containing constant and variable symbols, assumptions, and already-established facts). The Proof Manager tracks the context, which is modified by non-leaf proof steps. For leaf proof steps, it sends the corresponding sequent to the back-end provers, and records the status of the step's proof (succeeded, failed, canceled by the user, or omitted).

Because proof obligations are independent of one another, users can develop proofs in any order and work on the proof of a step independently of the state of the proof of other steps. This permits them to concentrate on the part of a planned proof that is most likely to be wrong and require changes to other parts. The Toolbox makes it easy to instruct TLAPS to check the proof of everything in a file, of any single theorem, or of any single step. It displays every obligation whose proof fails or is taking too long; in the latter case the user can cancel the proof. Clicking on the obligation shows the part of the proof that generated it.

A linear presentation, as in Figure 6, is unsuitable for reading or writing large proofs. The Toolbox's editor helps reading and writing large TLA$^+$ proofs,

providing commands that show or hide particular subproofs. Commands to hide a proof or view just its top level aid in reading a proof. A command that is particularly useful when writing a subproof is one that hides all preceding steps that cannot be used in that subproof because of their positions in the hierarchy.

TLA$^+$'s hierarchical proofs provide a much more powerful mechanism for structuring complex proofs than the conventional approach using lemmas. In a TLA$^+$ proof, each step with a non-leaf proof is effectively a lemma. One typical 1100-line invariance proof [13] contains 100 such steps. A conventional linear proof with 100 lemmas would be impossible to read.

4.2 Fingerprinting: Tracking The Status Of Proof Obligations

During proof development, a user repeatedly modifies the proof structure or changes details of the specification. Rerunning the back-end provers on a sizable proof takes time. By default, TLAPS does not re-prove an obligation that it has already proved—even if the proof has been reorganized and the step that generated it has been moved, or if the step was removed from the proof and reinserted in a later version. It can also show the user the impact of a change by indicating which parts of the existing proof must be re-proved.

The Proof Manager computes a *fingerprint* of every obligation, which it stores, along with the obligation's status, in a separate file. Technically, a proof obligation is canonically represented as a lambda term, with bound variables replaced by de Bruijn indices [7] such that their actual names in the TLA$^+$ proof are irrelevant. The context is minimized by erasing symbols and hypotheses that are not used in the step. The fingerprint is a compact representation of the resulting term, which is therefore insensitive to structural modifications of the proof context that do not affect the obligation's logical validity.

The Toolbox displays the proof status of each step, indicating by color whether the step has been proved or some obligation in its proof has failed or has been omitted. Looking up an obligation's status takes little time, so the user can tell TLAPS to re-prove a step or a theorem even if only a small part of the proof has changed; TLAPS will recognize any obligation that has not changed and will not attempt to prove it anew. There is also a check-status command that displays the proof status without actually launching any proofs.

An incident that occurred in the Byzantine Paxos proof reveals the advantages of our method of writing proofs. The third author wrote the safety proof primarily as a way of debugging TLAPS, spending a total of several weeks over several months on it. Later, when writing a paper about the algorithm, he discovered that it did not satisfy the desired liveness property, so it had to be modified. He changed the algorithm, fixed minor bugs found by TLC, and reproved the safety property—all in a day and a half, with about 12 hours of actual work. He was able to do it that fast because of the hierarchical proof structure, TLAPS's fingerprinting mechanism (about 3/4 of the proof obligations in the new proof had already been proved), and the Toolbox's aid in managing the proof.

5 Related Work

We have designed the TLA$^+$ proof system as a platform for interactively verifying concurrent and distributed algorithms. Unlike most interactive proof assistants [23], TLAPS has been designed around a declarative proof language that is independent of any specific proof back-end. TLA$^+$ proofs indicate what facts are needed to prove a certain result, but they do not specify precisely how the back-end provers should use these facts. Although this lack of fine control can frustrate users who are intimately familiar with the inner workings of a particular prover, declarative proofs are less dependent on specific back-end provers and less sensitive to changes in their implementation.

We write complex proofs by hierarchically structuring their logic. The graphical user interface provides commands that support hierarchical proofs by allowing a user to zoom in on the current context and by supporting non-linear proof development. Although some other interactive proof systems such as Mizar [20] and Isabelle/Isar [21] also offer hierarchical proofs, to the best of our knowledge these systems do not provide the Toolbox's abilities to use that structure to aid in reading and writing proofs and to prove individual steps in any order—facilities that we find crucial in developing and managing large proofs. The only other proof assistant that we know to offer a mechanism comparable to our fingerprinting facility is the KIV system [2].

The Rodin toolset supporting the Event-B formal method [1] shares several aspects with TLAPS: Event-B and TLA$^+$ are both based on set theory, both emphasize refinement as a way to structure formal developments, and Rodin and TLAPS mechanize proofs of safety properties with the help of different back-end provers. Unlike with Event-B models, the structure of TLA$^+$ specifications is not fixed: any TLA$^+$ formula can be considered as a system specification or a property, and TLAPS does not impose a structure on invariant or refinement proofs.

Provers designed for program verification such as VCC [6] or Why [8] target low-level source code rather than high-level specifications of algorithms. They are based on generators of verification conditions corresponing to programming constructs, that are discharged by invoking powerful automatic provers. User interaction is essentially restricted to the choice of suitable program annotations.

6 Conclusion

Using the example of Peterson's algorithm, we have presented the main constructs of the TLA$^+$ proof language and, by extension, the ideas underlying the language design. That algorithm was chosen because it is well known and simple—so simple that we had to eschew the use of the SMT solver back-end so we could write a nontrivial proof. We explained in Section 4 why TLA$^+$ proofs scale to more complex algorithms and specifications that we do not expect any prover to handle automatically. The hierarchical structure of the proof language is essential for giving users flexibility in designing their proof structure, and it

ensures that individual proof steps are independent of one another. The fingerprinting mechanism of TLAPS makes use of this independence by storing previously proved results and retrieving them, even when they appear in a different context.

While not illustrating the entire proof language [12], Peterson's algorithm does show its main features. Steps correspond to natural-deduction sequents. Leaf proofs immediately prove a step, citing the necessary definitions, facts, and assumptions. Non-leaf proofs consist of another level of proof steps that end with a QED step. This basic structure is oriented towards forward-style proofs, but the judicious use of backward chaining (SUFFICES steps) can make proofs more readable. Some features of the proof language that do not appear in the proof of Figure 6 are constructs for providing a witness to prove an existentially quantified formula, introducing local definitions, and specifying facts that can be used by the back-end provers even when they are not explicitly mentioned.

Different proof techniques, such as resolution, tableau methods, rewriting, and SMT solving offer complementary strengths. Future versions of TLAPS will probably add new back-end provers. Adding a new back-end mainly involves writing a translation from TLA$^+$ to the input language of the prover. Such translations can be complex, and there is a legitimate concern about their soundness as well as about the soundness of the back-ends themselves. For back-ends that can produce proof traces, TLAPS provides the option to certify the traces within Isabelle. Proof trace certification has been implemented for Zenon, and we plan to implement it for other back-end provers including SMT solvers. Still, it is much more likely that a proof is meaningless because of an error in the formula we are proving than because of an error in a back-end. Soundness also depends on parts of the proof manager. Users who do not trust its fingerprinting mechanism can disable it and reprove the entire proof or any part of it. The proof manager also carries out some critical transformations, such as replacing $(a + b)'$ by $(a' + b')$.

We cannot overstate how important it is that TLAPS is integrated with the other TLA$^+$ tools—especially the TLC model checker. Checking putative invariants and assertions with TLC on finite instances of a specification is much more productive than discovering errors during the proof. Users check the exact same specifications that appear in their proofs. Less obvious is how useful it is that TLC can evaluate TLA$^+$ formulas. When verifying a system, we don't want to prove well-known mathematical facts; we want to assume them. However, it is easy to make a mistake in formalizing even simple mathematics, and assuming the truth of an incorrect formula can lead to an incorrect proof. TLC can usually check the exact TLA$^+$ formulas assumed in a proof for a large enough instance to make us confident that our formalization of a correct mathematical result is indeed correct.

We are actively developing TLAPS. The current version supports reasoning about non-temporal formulas, which is enough for proving safety properties, including invariants and step simulation. Non-trivial temporal reasoning is required for proving liveness properties, and our main short-term objective is to

support temporal reasoning in TLAPS. It is not obvious how best to extend natural deduction to temporal logic. We have designed an approach involving two forms of sequents, expressed with two forms of the ASSUME/PROVE statement having different semantics, that we think will work well. We also plan to improve support for standard TLA$^+$ data structures such as sequences.

References

1. J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.
2. M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *FASE*, volume 1783 of *LNCS*, pages 363–366, Berlin, Germany, 2000. Springer.
3. C. Barrett, L. de Moura, S. Ranise, A. Stump, and C. Tinelli. The SMT-LIB initiative and the rise of SMT. In S. Barner, I. Harris, D. Kroening, and O. Raz, editors, *Hardware and Software: Verification and Testing*, volume 6504 of *LNCS*, pages 3–3. Springer, 2011.
4. R. Bonichon, D. Delahaye, and D. Doligez. Zenon : An extensible automated theorem prover producing checkable proofs. In N. Dershowitz and A. Voronkov, editors, *LPAR*, volume 4790 of *LNCS*, pages 151–165, Yerevan, Armenia, 2007. Springer.
5. K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. Verifying safety properties with the TLA$^+$ proof system. In J. Giesl and R. Hähnle, editors, *IJCAR*, volume 6173 of *LNCS*, pages 142–148, Edinburgh, UK, 2010. Springer.
6. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying Concurrent C. In *22nd Intl. Conf. Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *LNCS*, pages 23–42, Munich, Germany, 2009. Springer.
7. N. G. de Bruijn. Lambda calculus notation with nameless dummies. A tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.
8. P. Herms, C. Marché, and B. Monate. A certified multi-prover verification condition generator. In R. Joshi, P. Müller, and A. Podelski, editors, *4th Intl. Conf. Verified Software: Theories, Tools, Experiments (VSTTE 2012)*, volume 7152 of *LNCS*, pages 2–17, Philadelphia, PA, 2012. Springer.
9. L. Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83*, pages 657–668, Paris, Sept. 1983. IFIP, North-Holland.
10. L. Lamport. *Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2003.
11. L. Lamport. The PlusCal algorithm language. In M. Leucker and C. Morgan, editors, *ICTAC*, volume 5684 of *LNCS*, pages 36–60, Kuala Lumpur, Malaysia, 2009. Springer.
12. L. Lamport. TLA^{+2}: A preliminary guide. Draft manuscript, Nov. 2011. `http://research.microsoft.com/users/lamport/tla/tla2-guide.pdf`.
13. L. Lamport. Byzantizing Paxos by refinement. Available at `http://research.microsoft.com/en-us/um/people/lamport/pubs/web-byzpaxos.pdf`, 2011.
14. L. Lamport. How to write a 21st century proof. *Journal of Fixed Point Theory and Applications*, Mar. 2012. DOI: 10.1007/s11784-012-0071-6.

15. T. Lu, S. Merz, and C. Weidenbach. Towards verification of the Pastry protocol using TLA$^+$. In R. Bruni and J. Dingel, editors, *FORTE*, volume 6722 of *LNCS*, pages 244–258, Reykjavik, Iceland, 2011. Springer.
16. S. Merz and H. Vanzetto. Automatic verification of TLA$^+$ proof obligations with SMT solvers. In N. Bjørner and A. Voronkov, editors, *LPAR*, volume 7180 of *LNCS*, pages 289–303, Mérida, Venezuela, 2012. Springer.
17. B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *Security and Privacy*, pages 379–394. IEEE, 2011.
18. G. L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.
19. The TLAPS Project. Web page. `http://msr-inria.inria.fr/~doligez/tlaps/`.
20. A. Trybulec. Mizar. In Wiedijk [23], pages 20–23.
21. M. Wenzel and L. C. Paulson. Isabelle/Isar. In Wiedijk [23], pages 41–49.
22. M. Wenzel, L. C. Paulson, and T. Nipkow. The Isabelle framework. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *TPHOLs*, volume 5170 of *LNCS*, pages 33–38, Montreal, Canada, 2008. Springer.
23. F. Wiedijk, editor. *The Seventeen Provers of the World*, volume 3600 of *LNCS*. Springer, 2006.