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 and show how TLAPS and the Toolbox (an IDE for TLA$^+$) help users to manage large, complex proofs.

1 Introduction

TLA$^+$ [5] is a specification language originally designed for specifying concurrent and distributed systems and their properties. It is based on Zermelo-Fraenkel set theory for modeling data structures and on the linear-time temporal logic TLA for specifying system executions and their properties. More recently, constructs for writing proofs have been added to TLA$^+$, following a proposal for presenting rigorous hand proofs in a hierarchical style [8].

In this paper, we present the main ideas that guided the design of the proof language and its implementation in TLAPS, the TLA$^+$ proof system [3, 13]. The proof language and TLAPS have been designed to be independent of any particular theorem prover. All interaction takes place at the level of TLA$^+$. Users need know only what sort of reasoning TLAPS's backend provers tend to be good at—for example, that SMT solvers excel at arithmetic. This knowledge is gained mostly by experience.

TLAPS has a *Proof Manager* (PM) that transforms a proof into individual proof obligations that it sends to backend provers. Currently, the main backend provers are Isabelle/TLA$^+$, an encoding of TLA$^+$ as an object logic in Isabelle [14], Zenon [2], a tableau prover for classical first-order logic with equality, and a backend for SMT solvers. Isabelle serves as the most trusted backend prover, and when possible, we expect backend provers to produce a detailed proof that is checked by Isabelle. This is currently implemented for the Zenon backend.

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

TLAPS has been integrated into the TLA$^+$ Toolbox, an IDE (Integrated Development Environment) based on Eclipse for writing TLA$^+$ specifications and running the TLA$^+$ tools on them, including the TLC model checker. The Toolbox 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.

We explain how to write and check TLA$^+$ proofs, using a tiny well-known example: a proof that Peterson's algorithm [12] implements mutual exclusion. We start by writing the algorithm in PlusCal [6], 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.

Section 4 indicates the features that make TLA$^+$, TLAPS, and the Toolbox scale to realistic examples. 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: no two processes are in their critical sections at the same time.[6]

A representation of Peterson's algorithm in the PlusCal algorithm language is shown on the left-hand side of Figure 1. The two processes are named 0 and 1; the PlusCal code is embedded in a TLA$^+$ module that defines an operator *Not* so that $Not(0) = 1$ and $Not(1) = 0$.

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.

A translator, normally called from the Toolbox, generates a TLA$^+$ specification from the PlusCal code. We illustrate the structure of the TLA$^+$ translation in the right-hand part of Figure 1. The heart of the TLA$^+$ specification consists of the predicates *Init* describing the initial state and *Next*, which represents the next-state relation.

The PlusCal translator adds a variable *pc* to record the control state of each process. The meaning of formula *Init* in the figure is straightforward. The formula *Next* is the disjunction of the two formulas $proc(0)$ and $proc(1)$, which

[6] The TLA$^+$ module containing the specification and proof as well as an extended version of this paper are accessible at the TLAPS Web page [13].

```
--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
```

```
VARIABLES flag, turn, pc
vars ≜ ⟨flag, turn, pc⟩
Init ≜ ∧ flag = [i ∈ {0,1} ↦ FALSE]
       ∧ turn = 0
       ∧ pc = [self ∈ {0,1} ↦ "a0"]
a3a(self) ≜
  ∧ pc[self] = "a3a"
  ∧ IF flag[Not(self)]
    THEN pc' = [pc EXCEPT ![self] = "a3b"]
    ELSE pc' = [pc EXCEPT ![self] = "cs"]
  ∧ UNCHANGED ⟨flag, turn⟩

\* remaining actions omitted

proc(self) ≜ a0(self) ∨ ... ∨ a4(self)
Next ≜ ∃ self ∈ {0,1} : proc(self)
Spec ≜ Init ∧ □[Next]_vars
```

Fig. 1. Peterson's algorithm in PlusCal (left) and in TLA$^+$ (excerpt, right).

are in turn defined as disjunctions of formulas corresponding to the atomic steps of the **process**. In these formulas, unprimed variables refer to the old state and primed variables to the new state. The temporal formula *Spec* is the complete specification. It characterizes behaviors (ω-sequences of states) that start in a state satisfying *Init* and where every pair of successive states either satisfies *Next* or else leaves the values of the tuple *vars* unchanged.[7]

Before trying to prove that the algorithm is correct, we use TLC, the TLA$^+$ model checker, to check it for errors. The Toolbox runs TLC on a model of a TLA$^+$ specification. A model usually assigns particular values to specification constants, such as the number of processes. It can also restrict the set of states explored, which is useful if the specification allows an infinite number of reachable states. TLC easily verifies that the two processes can never both be at label *cs* by checking that the following formula is an invariant (true in all reachable states):

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

Peterson's algorithm is so simple that TLC can check all possible executions. For more interesting algorithms that have parameters (such as the number of processes) and perhaps an infinite set of reachable states, TLC cannot exhaustively verify all executions, and correctness can only be proved deductively. Still, TLC is invaluable for catching errors, and it is much easier to run TLC than to write a formal proof.

[7] “Stuttering steps” are allowed in order to make refinement simple [4].

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. 2. The high-level proof.

3 Proving Mutual Exclusion For Peterson's Algorithm

The assertion that Peterson's algorithm implements mutual exclusion is formalized in TLA$^+$ as the theorem in Figure 2. The standard method of proving this invariance property is to find an inductive invariant *Inv* such that the steps $\langle 1 \rangle 1$–$\langle 1 \rangle 3$ of Figure 2 are provable.

TLA$^+$ proofs are hierarchically structured and are generally written top-down. Each proof in the hierarchy ends with a QED step that asserts the proof's goal. We usually write the QED step's proof before the proofs of the intermediate steps. The QED step follows easily from steps $\langle 1 \rangle 1$–$\langle 1 \rangle 3$ by standard proof rules of temporal logic. However, TLAPS does not yet handle temporal reasoning, so we omit that step's proof. When temporal reasoning is added to TLAPS, we expect it easily to check such a trivial proof.

Figure 3 defines the inductive invariant *Inv* as the conjunction of two formulas. (A definition must precede its use, so the definition of *Inv* appears in the module before the proof.) The first, *TypeOK*, asserts simply 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. We again use TLC to check that *Inv* is indeed an invariant. In our simple example, TLC can even check that *Inv* is inductive, by checking that it is an (ordinary) invariant of the specification $Inv \wedge \Box[Next]_{vars}$, obtained from *Spec* by replacing the initial condition by *Inv*.

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

Fig. 3. The inductive invariant.

⟨1⟩2. $Inv \wedge [Next]_{vars} \Rightarrow Inv'$
  ⟨2⟩1. SUFFICES ASSUME $Inv, Next$ PROVE Inv'
  ⟨2⟩2. $TypeOK'$
  ⟨2⟩3. I'
    ⟨3⟩1. SUFFICES ASSUME NEW $j \in \{0, 1\}$ PROVE $I!(j)'$
    ⟨3⟩2. PICK $i \in \{0, 1\} : proc(i)$
    ⟨3⟩3. CASE $i = j$
    ⟨3⟩4. CASE $i \neq j$
    ⟨3⟩5. QED
  ⟨2⟩4. QED

Fig. 4. Outline of a hierarchical proof of step ⟨1⟩2.

We now prove steps ⟨1⟩1–⟨1⟩3. We can prove them in any order; let us start with ⟨1⟩1. This step follows easily from the definitions, and the following leaf proof is accepted by TLAPS:

BY DEF *Init*, *Inv*, *TypeOK*, *I*

TLAPS will not expand definitions unless directed to so. In complex proofs, automatically expanding definitions often leads to formulas that are too big for provers to handle. Forgetting to expand some definition is a common mistake. If a proof does not succeed, the Toolbox displays the exact proof obligation that it passed to the prover. It is then usually easy to see 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*. Unfortunately, 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 backend provers. However, TLAPS also includes an SMT solver backend [10] 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 backend when proving the current step by writing

BY SMT DEF ...

The backend translates the proof obligation to the input language of SMT solvers. In this way, step ⟨1⟩2 is proved in a few seconds. For sufficiently complicated algorithms, an SMT solver will not be able to prove inductive invariance as a single obligation. Instead, the proof will have to be hierarchically decomposed. We illustrate how this is done by writing a proof of ⟨1⟩2 that can be checked using only the Zenon and Isabelle backend provers.

The outline of a hierarchical proof of step ⟨1⟩2 appears in Figure 4. The proof introduces more elements of the TLA$^+$ proof language that we now explain.

A SUFFICES step allows a user to introduce an auxiliary assertion, from which the current goal can be proved. For example, step $\langle 2\rangle 1$ reduces the proof of the implication asserted in step $\langle 1\rangle 2$ to assuming predicates *Inv* and *Next*, and proving Inv'. In particular, this step establishes that the invariant is preserved by stuttering steps that leave the tuple *vars* unchanged. Steps $\langle 2\rangle 2$ and $\langle 2\rangle 3$ establish the two conjuncts in the definition of *Inv*. Whereas $\langle 2\rangle 2$ can be proved directly by Isabelle, $\langle 2\rangle 3$ needs some more interaction.

Following the definition of predicate I as a universally quantified formula, we introduce in step $\langle 3\rangle 1$ a new variable j, assume that $j \in \{0, 1\}$, and prove $I!(j)'$, which denotes the body of the universally quantified formula, with j substituted for the bound variable, and with primed copies of all state variables. Similarly, step $\langle 3\rangle 2$ introduces variable i to denote the process that makes a transition, following the definition of *Next* (which is assumed in step $\langle 2\rangle 1$). Even after this elimination of two quantifiers, Isabelle and Zenon cannot prove the goal in a single step. The usual way of decomposing the proof is to reason separately about each atomic action $a0(i)$, ..., $a4(i)$. However, Peterson's algorithm is simple enough that we can just split the proof into the two cases $i = j$ and $i \neq j$ with steps $\langle 3\rangle 3$ and $\langle 3\rangle 4$. Isabelle and Zenon can now prove all the steps.

4 Writing Real Proofs

Peterson's algorithm is a tiny example. Some larger case studies have been carried out using the system [7, 9, 11]. Several features of TLAPS and its Toolbox interface help in coping with the complexity of large proofs.

4.1 Hierarchical Proofs And The Proof Manager

Hierarchical structure is the key to managing complexity. TLA$^+$'s hierarchical and declarative proof language enables a user to keep decomposing a complex proof into smaller steps until the steps become provable by one of the backend provers. In logical terms, proof steps correspond to natural-deduction sequents that must be proved in the current context. 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 backend provers, and records the result of the step's proof that they report.

Proof obligations are independent of one another, so users can develop proofs in any order and work on different proof steps independently. The Toolbox makes it easy to instruct TLAPS to check the proof of everything in a file, of a single theorem, or of any step in the proof hierarchy. Its editor helps reading and writing large proofs, providing commands that show or hide subproofs. Although some other interactive proof systems offer hierarchical proofs, we do not know of other systems that provide the Toolbox's abilities to use that structure to aid in reading and writing proofs and to prove steps in any order.

Hierarchical proofs are much better than conventional lemmas for handling complexity. In a TLA$^+$ proof, each step with a non-leaf proof is effectively a

lemma. One typical 1100-line invariance proof [7] contains 100 such steps. A conventional linear proof with 100 lemmas would be impossible to read.

Unlike most interactive proof assistants [15], TLAPS is independent of any specific backend prover. There is no way for a user to indicate how available facts should be used by backends. TLA$^+$ proofs are therefore less sensitive to changes in any prover's implementation.

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. By default, TLAPS does not re-prove an obligation that it has already proved—even if the proof has been reorganized. 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. The fingerprint is a compact canonical representation of the obligation and the relevant part of its context. 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 been omitted. The only other proof assistant that we know to offer a mechanism comparable to our fingerprinting facility is the KIV system [1].

5 Conclusion

The proof of Peterson's algorithm illustrates the main constructs of the hierarchical and declarative TLA$^+$ proof language. The algorithm is so simple that we had to eschew the use of the SMT solver backend so we could write a nontrivial proof. Section 4 explains why TLAPS, used with the TLA$^+$ Toolbox, can handle more complex algorithms and specifications.

A key feature of TLAPS is its use of multiple backend provers. Different proof techniques, such as resolution, tableau methods, rewriting, and SMT solving offer complementary strengths. Future versions of TLAPS will probably support additional backend provers. Because multiple backends raise concerns about soundness, TLAPS provides the option of having Isabelle certify proof traces produced by backend provers; and this has been implemented for Zenon. Still, it is much more likely that a proof is meaningless because of an error in the specification than that it is wrong because of an error in a backend. Soundness also depends on parts of the proof manager.

We cannot overstate the importance of having TLAPS integrated with the other TLA$^+$ tools—especially the TLC model checker. Finding errors by running TLC on finite instances of a specification is much faster and easier than discovering them when writing a proof. Also, verifying an algorithm or system may require standard mathematical results. For example, the correctness of a distributed algorithm might depend on known facts about graphs. Engineers want to assume such results, not prove them. However, it is easy to make a

mistake when formalizing mathematics. TLC can check the exact TLA$^+$ formulas assumed in a proof (on finite instances), greatly reducing the chance of introducing an unsound assumption.

We are actively developing TLAPS. Our main short-term objective is to add support for temporal reasoning. We have designed a smooth extension of the existing proof language to sequents containing temporal formulas. We also plan to improve support for standard TLA$^+$ data structures such as sequences.

References

1. 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.
2. 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.
3. 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.
4. 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.
5. L. Lamport. *Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2003.
6. 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.
7. L. Lamport. Byzantizing Paxos by refinement. Available at `http://research.microsoft.com/en-us/um/people/lamport/pubs/web-byzpaxos.pdf`, 2011.
8. 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.
9. 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.
10. 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.
11. 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.
12. G. L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.
13. The TLAPS Project. Web page. `http://msr-inria.inria.fr/~doligez/tlaps/`.
14. 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.
15. F. Wiedijk, editor. *The Seventeen Provers of the World*, volume 3600 of *LNCS*. Springer, 2006.