Specifying and Verifying Systems With TLA$^+$

Leslie Lamport
Microsoft Research

John Matthews
HP Labs
Cambridge Research Lab
Cambridge, MA

Mark Tuttle
HP Labs
Cambridge Research Lab
Cambridge, MA

Yuan Yu
Microsoft Research

To appear in the Proceedings of the Tenth ACM SIGOPS European Workshop, to be held 22–25 September 2002 in Saint-Emilion, France.

Contents

Specifying and Verifying Systems With TLA$^+$

Leslie Lamport
Microsoft Research

John Matthews
HP Labs
Cambridge Research Lab
Cambridge, MA

Mark Tuttle
HP Labs
Cambridge Research Lab
Cambridge, MA

Yuan Yu
Microsoft Research

Abstract

TLA$^+$ is a high-level specification language that has been used to specify and check the correctness of several hardware protocols. We expect that it can also be used to specify and check concurrent algorithms and protocols for software systems.

1. Introduction

Correct code is an important component of software reliability. Modern operating systems make extensive use of concurrent and distributed algorithms. These algorithms are subtle and easy to get wrong, leading to an incorrect high-level design. We are concerned with formally describing high-level system designs and checking their correctness. We also address the question of checking that code implements the high-level design.

Systems should be described in a formal specification language to produce unambiguous descriptions that can be checked with tools. Conventional programming languages are ill-suited to this task because: (i) they describe one way of doing something, while a high-level description should allow many implementations; (ii) the need to generate efficient code makes them complicated; and (iii) they do not provide the mathematical abstractions required for simple, high-level specifications. We use the language TLA$^+$. We believe that its simplicity, elegance, and expressiveness make it ideal for writing high-level descriptions of concurrent and distributed systems.

We have had considerable experience applying TLA$^+$ to hardware protocols. Our experience suggests that it should be good for software as well. Section 2 describes TLA$^+$ and its associated tools and techniques; section 3 describes our experience using it; and section 4 discusses its potential application to software systems.

2. TLA$^+$ and Friends

2.1. TLA$^+$

TLA$^+$ is a formal specification language based on (untyped) ZF set theory, first-order logic, and TLA (the Temporal Logic of Actions) [4, 7]. TLA is a temporal logic developed for describing and reasoning about concurrent and distributed systems [5]. TLA$^+$ includes modules and ways of combining them to form larger specifications.

Although TLA$^+$ permits a wide variety of specification styles, a typical specification has the form $Init \wedge \Box Next \wedge Liveness$, where:[1]

$Init$ is the initial-state predicate—a formula describing all legal initial states.

$Next$ is the next-state relation, which specifies all possible steps (pairs of successive states) in a behavior of the system. It is a disjunction of actions that describe the different system operations. An action is a mathematical formula in which unprimed variables refer to the first state of a step and primed variables refer to its second state.

$Liveness$ is a temporal formula that specifies the liveness (progress) properties of the system as the conjunction of fairness conditions on actions. Although the specifications we wrote included liveness properties, these properties were not checked during the verification, so we largely ignore liveness here.

Such a specification essentially describes a state machine. However, unlike specifications in methods based on abstract machines or programming languages, the state predicates and actions in TLA$^+$ specifications are arbitrary formulas written in a high-level language with the full power of set theory and predicate logic, making TLA$^+$ very expressive. A TLA$^+$ specification consists of a single mathematical formula.

2.2. TLC

TLC [11] is an on-the-fly model checker for debugging TLA$^+$ specifications. This distinguishes it from almost all other model checkers, which require specifications to be written in primitive, low-level languages. No model checker can handle all the specifications that can be written in a language as expressive as TLA$^+$. However, TLC can handle a subclass of TLA$^+$ specifications that seems to include the ones that arise in describing actual systems.

Explicit-state model checkers like TLC must generate all reachable states. Real system specifications are usually not finite state—for example, they may contain unspecified sets of processors and

[1] The formula $\Box Next$ should actually be $\Box[Next]_v$, where v is the tuple of all variables; we drop such subscripts for simplicity. We also ignore the hiding of internal variables, expressed with temporal existential quantification.

unbounded message queues. We run TLC on the actual specification, using a separate configuration file that specifies a finite-state instance. TLC can also be used to generate finite-length random simulations of even infinite-state specifications.

TLC can be used to check safety and liveness properties of a specification written in the form $Init \wedge \Box Next \wedge Liveness$. For safety properties, TLC explores all reachable states in the model, looking for one in which an invariant is not satisfied or deadlock occurs (there is no possible next state). For liveness properties, TLC uses the standard tableau method described in [9]. TLC avoids the tableau construction of the temporal formula if it contains only temporal subformulas of the form $\Box\Diamond A$ or $\Diamond\Box A$, where A is a state or action predicate. This allows it to handle SF (strong fairness) and WF (weak fairness) properties more efficiently. Because the state graph constructed for liveness checking can easily be too large for TLC to handle, TLC periodically checks the liveness properties on the partial state graph constructed so far. When TLC detects an error, a state trace that exhibits the error is printed as part of the error report. For a violation of a safety property, the error trace is guaranteed to be minimal-length.

There are a number of interesting features that makes TLC a useful tool.

- TLC keeps all of its internal data structures for invariant checking on disk, using main memory as a cache to provide efficient implementation of those data structures. This allows us to explore large state spaces—eliminating the common way in which explicit-state model checkers run out of memory. (The largest state space TLC has checked so far contains about 900 million distinct reachable states.)

- TLC can run in multi-threaded mode to make use of multiprocessors, and in distributed mode to make use of multiple machines. It obtains a speedup that is nearly linear in the number of processors. TLC users routinely check their specifications using large multiprocessor systems and/or multiple machines.

- TLC periodically generates checkpoints during its runs. A checkpoint can be used to resume an old run after a system crash, or even after correcting a minor error found by TLC. Because its checkpointing is very fast, TLC can afford to take checkpoints fairly frequently during a long run. TLC users have found this feature quite useful.

- TLC allows a user to specify symmetries in a specification by providing a group of symmetry-preserving permutations. It then applies the standard symmetry-reduction techniques to explore only the quotient state space modulo the permutation group.

Because TLA$^+$ is such a high-level language, we expect TLC to be slower than comparable model checkers—perhaps by a factor of ten in the single-threaded mode, although we have not made any real effort to compare it with other systems. But TLC can make use of multiprocessors and multiple machines when better performance is needed. Symbolic model checkers outperform explicit-state ones on many problems. A new TLA$^+$ model checker based on satisfiability solving is therefore being written.

2.3. Hierarchical Proofs

In many cases, a model checker cannot handle a large enough instance of a specification to provide enough confidence in its correctness. When model checking does not suffice, we must use proof. Because TLA$^+$ specifications are mathematical formulas, correctness properties are expressed directly as mathematical theorems. The advantage is that there is no need to generate verification conditions; what you see is what you prove.

To prove that a state predicate I is an invariant of a specification S, we must prove $S \Rightarrow \Box I$, where $\Box I$ is the temporal formula asserting that I is always true. When S has the usual form $Init \wedge \Box Next$, this is proved by finding a formula Inv, called an invariant of $Next$, that satisfies $Init \Rightarrow Inv$, $Inv \wedge Next \Rightarrow Inv'$, and $Inv \Rightarrow I$, where Inv' is the formula obtained from Inv by priming all variables.

To prove that a specification S_1 implements another specification S_2, we must express the variables of S_2 as functions of the variables of S_1 and prove $S_1 \Rightarrow \overline{S_2}$, where $\overline{S_2}$ is formula S_2 with its variables replaced by the corresponding functions of the variables of S_1. The proof rules of TLA reduce all proofs to reasoning about individual states or pairs of states, using ordinary, nontemporal mathematical reasoning. Over the years, such state-based reasoning has been found to be more reliable than behavioral reasoning, in which one reasons directly about the entire execution.

The theorems that express correctness may be hundreds or thousands of lines long, and their proofs are long and complex. We have developed a hierarchical proof method in which the structure of the formula to be proved largely determines the structure of the proof [2, 6]. For example, the proof of $A \Rightarrow B \wedge C$ consists of the two lower-level steps $A \Rightarrow B$ and $A \Rightarrow C$, and the proof of $A \vee B \Rightarrow C$ consists of the steps $A \Rightarrow C$ and $B \Rightarrow C$.

We have only minimal tool support, in the form of Emacs macros, to help with hand proofs. One interesting application of TLC is to debug these theorems before proving them, and to debug the proofs as they are written. We have not explored the use of mechanical proof systems for real system specifications.

3. Experience

3.1. Wildfire

In the fall of 1996, three of us began a project to verify the cache-coherence protocol of an EV6-based multiprocessor [3]. (EV6 and EV7 are the internal names for the Alpha 21264 and 21364 processors.) We spent about three months writing a TLA$^+$ specification of the protocol, starting with a stack of detailed, incomplete, and inconsistent design documents. Interacting frequently with the system's designers by email and telephone, we produced a 1900 line specification. We also wrote a high-level TLA$^+$ specification of the Alpha memory model, which the protocol was supposed to implement. The memory model and a simplified version of the protocol specification are available on the web [8].

TLC was not yet written, so our only reasonable option for verifying the protocol was to write a hand proof. We did not have the resources to write a complete proof, so we decided to find an invariant of the protocol and prove its invariance. We wrote about 1000 lines of informal state predicates that formed the major part of a complete (inductive) invariant. (Although it would have been straightforward, we decided not to spend the time writing the invariant in TLA$^+$.) We selected two conjuncts, each about 150 lines long, as the part of the invariant most likely to reveal an error. We completed the proof for one of the conjuncts; it was about 2000 lines long and 13 levels deep. The proof of the second conjunct would have been about twice as long, but we stopped about halfway through because we decided that the likelihood of its discovering an error was too small to justify further effort. We spent about seven months on these two proofs.

We also wrote an informal higher-level proof of one crucial aspect of the protocol. It was about 550 lines long and had a maximum depth of 10 levels.

We found two ways in which the protocol did not implement the Alpha memory model. One was deemed to be an error in the memory model, which was subsequently changed in [1]. The other was a genuine bug in the protocol—an easily-fixed error in one entry of one table. The simplest scenario displaying the bug required four processors, two memory locations, and over 15 messages. We believe that this error could have been found only by writing a proof; an instance of the protocol exhibiting the error would have too many states to be checked exhaustively by a model checker.

Perhaps the major achievement of the project was to subject the protocol to a level of rigorous analysis that significantly increased the designer's confidence in its correctness. The designers were quite happy with our work.

3.2. EV7

Influenced by our effort on the EV6 protocol, engineers decided to use TLA$^+$ to verify the cache-coherence protocol of the EV7. This project began in the spring of 1998 and is not yet finished. This time, the specification was written by an engineer who received a few hours instruction on TLA$^+$. (At the time, there was no language manual.) His specification was about 1800 lines long.

Meanwhile, the TLC model checker was being written. By the fall of 1998, it was ready to be applied to the TLA$^+$ specification. TLC was able to handle the specification; no modifications were required. The largest instances it was feasible to check with TLC had one cache line, two data values, and three processors. These instances had about 12 million distinct reachable states and originally took several days to run. Improvements to TLC have since reduced that time to a few hours.

We had planned to use TLC to check the RTL implementation by translating runs of the RTL simulator into behaviors at the level of the TLA$^+$ specification, and using TLC to check those behaviors. That idea was put aside, and is only now being implemented, because the engineers discovered another way to use TLC for RTL-level testing. The error traces that TLC produced in the course of debugging the specification often exhibited corner cases not considered by the designers. So, the verification team translated those traces into input stimuli for the verification team's RTL simulator. The translation was then automated and TLC was used to generate randomly chosen traces. The RTL input from such traces is better than purely random input because it satisfies the TLA$^+$ specification.

Work is continuing on combining model checking and simulation more systematically [10]. A translator from output of the RTL simulator to behaviors at the level of the TLA$^+$ specification is being written. TLC will then be used to check that the RTL code implements the TLA$^+$ specification during simulations. In the process of checking simulation steps, TLC also collects information about the TLA$^+$ specification states visited. TLC can then generate traces to interesting but unvisited specification states. These traces can be used to direct the RTL simulation towards coverage gaps.

An invariance proof was also written for the specification. TLC was used extensively to debug the invariant.

As soon as we started using TLC, we found many errors in the TLA$^+$ specification. Not counting simple mistakes that were easily corrected, we found about 70 errors. About 90% of them were discovered by TLC; the rest were found by a human reading the specification. Most of the errors were introduced when translating from the informal specification; they demonstrate the ambiguity inherent in such specifications. Five design/implementation errors were discovered—one directly by TLC, the other four by using TLC error traces to generate simulator input.

The engineers were quite happy with TLA$^+$. The verification group preferred the TLA$^+$ specification to the informal English one. The EV8 designers were planning to use a TLA$^+$ specification rather than an English one as the "official" cache-coherence protocol specification. However, the EV8 was cancelled. The engineers from that project, who are now at Intel, are continuing to use TLA$^+$. They report that TLA$^+$ and TLC are starting to become widely used within the former Alpha group at Intel, and they are generating significant interest in other groups.

3.3. Itanium

We have also applied TLA$^+$ to the cache-coherence protocols for multiprocessors based on the Intel Itanium processor. These systems have components that were designed elsewhere and therefore had to be modeled very abstractly. Simply writing the specifications uncovered many ambiguities in the English descriptions and suggested two small design changes.

TLC could not check the TLA$^+$ specification of these protocols on large enough instances to give us adequate confidence in the design. This was because (i) the highly abstract models of components designed elsewhere allowed many more interleavings than any actual implementation would, and (ii) the design appeared to require at least four processors to produce interesting scenarios. However, we were able to run meaningful tests by having TLC generate random simulations of large instances of the specification. We are currently exploring ways of model checking larger instances.

As part of this effort, we wrote a TLA$^+$ specification of the Itanium memory model, which we hope to release in the near future. In addition to serving as a correctness condition for hardware implementations, this specification could be used for analyzing the correctness of software systems intended to run on Itanium-based systems.

In this project, we were separated from the design team by several thousand miles. The fact that an engineer could make so much progress with TLA$^+$ and TLC in relative isolation re-enforced our confidence in the ability of engineers to use these tools.

3.4. Other Projects

We have also applied TLA$^+$ in a number of smaller projects. Disk Paxos [2] was developed for use in a project; to explain the algorithm in its full generality, we described it to the engineers with a TLA$^+$ specification. We used TLA$^+$ and TLC to find bugs in proposals submitted to the working group for the PCI-X bus protocol. For a database system project, we used TLC to check database recovery and cache-management protocols. We now routinely use TLC to check the concurrent algorithms we write in the course of our research.

4. Software

The major industrial applications of TLA$^+$ have been in the realm of hardware. Hardware engineers routinely use tools based on formal methods to check lower-level designs. They are not accustomed to using tools to check their high-level designs. However, they do write careful informal specifications of those designs. Engineers are aware that, with the increasing complexity of hardware, errors in their high-level designs are becoming an important concern. We have found them receptive to the idea of using TLA$^+$ at the protocol level.

At the highest level, there is no fundamental difference between hardware and software. The EV6 and EV7 cache-coherence protocols, which send messages that can be reordered in flight, look very much like distributed algorithms in which separate computers communicate over a local area network. So TLA$^+$, which has proven useful for hardware, should be just as useful for software. However, there does seem to be a cultural difference between hardware and software engineers. Software engineers do not have the same tradition of relying on specifications that hardware engineers do.

An important lesson we have learned is that moving formal methods from the research community to the engineering community requires patience and perseverance. Engineers are under severe constraints when designing a system. Speed is of the essence, and they never have as many people as they can use. Engineers must be convinced that formal methods will help before they will risk using them.

We plan to explore the use of TLA$^+$ for specifying and checking the high-level design of concurrent software systems. We believe that TLA$^+$ is ideal for specifying these systems. We expect that the methods of generating tests and of checking an implementation against the higher-level specification, developed for the EV7, can be used for software as well. We hope to work with engineers to discover how best to use TLA$^+$ in the software development process.

Acknowledgments

We were assisted by many colleagues in this work. On the EV6 project: Madhumitra Sharma helped us to understand the protocol and write its high-level proof; and Paul Harter helped write the low-level proof. On the EV7 project: Joshua Scheid wrote the specification; Homayoon Akhiani and Jonathan Nall implemented the TLA$^+$ to RTL translator; Damien Doligez wrote the invariance proof; and Serdar Tasiran has been implementing the RTL to TLA$^+$ translation. On the Itanium project: Jae Yang helped specify the cache-coherence protocol and Gil Neiger helped us specify the Itanium memory model. On other projects: Thomas Rodeheffer worked on the PCI-X bus protocol; and David Lomet worked on the database protocols. Rajeev Joshi helped us to build a satisfiability-based model checker for TLA$^+$.

References

[1] Alpha Architecture Committee. *Alpha Architecture Reference Manual*. Digital Press, Boston, third edition, 1998.

[2] E. Gafni and L. Lamport. Disk paxos. Technical Report 163, Compaq Systems Research Center, July 2000. To appear in *Distributed Computing*. Currently available on the web at `http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr% -163.html`.

[3] K. Gharachorloo, M. Sharma, S. Steely, and S. V. Doren. Architecture and design of AlphaServer GS320. In A. Gupta, editor, *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 13–24, Nov. 2000.

[4] L. Lamport. TLA—temporal logic of actions. A web page, a link to which can be found at URL `http://lamport.org`. The page can also be found by searching the Web for the 21-letter string formed by concatenating `uid` and `lamporttlahomepage`.

[5] L. Lamport. The temporal logic of actions. *ACM Trans. Prog. Lang. Syst.*, 16(3):872–923, May 1994.

[6] L. Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, August-September 1995.

[7] L. Lamport. *Specifying Systems*. Addison-Wesley, Boston, 2002.

[8] L. Lamport, M. Sharma, M. Tuttle, and Y. Yu. The wildfire verification challenge problem. At URL `http://www.research.compaq.com/SRC/tla/wildfire-challenge.html` on the World Wide Web. It can also be found by searching the Web for the 24-letter string `wildfirechallengeproblem`.

[9] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, 1991.

[10] S. Tasiran, Y. Yu, B. Batson, and S. Kreider. Using formal specifications to monitor and guide simulation: Verifying the cache coherence engine of the Alpha 21364 microprocessor. In *In Proceedings of the 3rd IEEE Workshop on Microprocessor Test and Verification, Common Challenges and Solutions*. IEEE Computer Society, 2002. To appear as an HP technical report.

[11] Y. Yu, P. Manolios, and L. Lamport. Model checking TLA$^+$ specifications. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66, Berlin, Heidelberg, New York, Sept. 1999. Springer-Verlag. 10th IFIP wg 10.5 Advanced Research Working Conference, CHARME '99.