Leslie Lamport: The Specification Language TLA$^+$

This is an addendum to a chapter by Stephan Merz in the book *Logics of Specification Languages* by Dines Bjørner and Martin C. Henson (Springer, 2008). It appeared in that book as part of a "reviews" chapter.

Stephan Merz describes the TLA logic in great detail and provides about as good a description of TLA$^+$ and how it can be used as is possible in a single chapter. Here, I give a historical account of how I developed TLA and TLA$^+$ that explains some of the design choices, and I briefly discuss how TLA$^+$ is used in practice.

Whence TLA

The logic TLA adds three things to the very simple temporal logic introduced into computer science by Pnueli [4]:

- Invariance under stuttering.
- Temporal existential quantification.
- Taking as atomic formulas not just state predicates but also action formulas.

Here is what prompted these additions.

When Pnueli first introduced temporal logic to computer science in the 1970s, it was clear to me that it provided the right logic for expressing the simple liveness properties of concurrent algorithms that were being considered at the time and for formalizing their proofs. In the early 1980s, interest turned from ad hoc properties of systems to complete specifications. The idea of specifying a system as a conjunction of the temporal logic properties it should satisfy seemed quite attractive [5]. However, it soon became obvious that this approach does not work in practice. It is impossible to understand what a conjunction of individual properties actually specifies. The only practical way to specify non-trivial systems is to describe them as abstract state machines. So, I started writing specifications as state machines, where the meaning of a state machine was a temporal logic formula that described the set of all its possible executions.

There is a basic problem with using a state machine as a specification. Consider an hour clock—a clock that displays only the hour. Ignoring the actual time that elapses between ticks, an hour clock is trivially specified by a state machine that increments the hour with each step. This specification, or any similar one, does not forbid the clock from showing minutes (or temperature or the phase of the moon). The specification should therefore be satisfied by a

clock that shows both the hour and the minute. However, a naive state-machine specification of the hour clock asserts that the hour changes with every step, while an hour-minute clock changes the hour only on every 60th step. This problem is solved by requiring invariance under stuttering. The specification of the hour clock must allow any finite number of stuttering steps—ones that leave the hour unchanged—between successive changes to the hour. Steps of the hour-minute clock that change only the minute are then stuttering steps allowed by the hour clock's specification.

At the time, such state-machine specifications were criticized as being overly specific. The state-machine specification of a FIFO queue I would have written in those days would have been equivalent to the specification given by module *InternalFIFO* in Merz's Figure 4 (Section 3.5), though probably written in a pseudo-programming language. Critics pointed out that a specification should only mention the interface variables *in* and *out*. The variable q should not appear, since there is no reason why an implementation needs to implement the required behavior with an explicit queue. The only way to avoid all mention of q is to describe explicitly all the queue's possible behaviors. To see how difficult this is, I urge the reader to try to write an informal natural-language specification of a FIFO queue without mentioning the contents of the queue. However, the criticism remained valid: a specification of the queue should be in terms only of the variables *in* and *out*. The answer was to hide internal state variables. Hiding a variable is expressed in temporal logic by temporal existential quantification. The only (free) variables of the specification *Fifo* in module *FIFO* of Merz's Figure 4 are *in* and *out*.

The final step in the development of TLA came when I realized that taking action formulas as the atomic formulas in Pnueli's temporal logic made it easy to describe state machines with temporal logic formulas. There was no need to translate from a language for expressing state machines into temporal logic. The state machine could be written directly as a temporal logic formula.

TLA has allowed me to better understand and to formalize many concepts in concurrency. Merz discusses implementation as implication and composition as conjunction. The example that I find most compelling is reduction. Reduction is the process of proving properties of a concurrent algorithm by reasoning about a coarser-grained version. There are a number of theorems and folk theorems stating when this is possible. For example, one reduction folk theorem asserts that if shared variables are accessed only in mutually exclusive critical sections, then we can pretend that the execution of an entire critical section is a single atomic step. It was intuitively clear that these results were all variations on one basic idea, but it was only with the aid of TLA that I was able to understand reduction well enough to express that idea as a single theorem that encompasses those prior results [1].

Whence TLA$^+$

After deciding that TLA was the right way to describe and reason about concurrent systems, my next step was to develop a complete specification language

based on it. Merz makes the simple idea of taking predicate logic and (untyped) set theory as the logic of actions for TLA seem natural and almost inevitable. In fact, it took me years to discard the usual concepts of computer science to achieve the simplicity of TLA$^+$. Here are two examples.

Like most computer scientists, I thought that assignment statements were the natural way to describe state changes. I was skeptical when Jim Horning suggested that I write $x' = x + 1$ instead of $x := x + 1$. However, I tried it and found that it worked quite well. Unlike most computer scientists, I realized how much simpler $x' = x + 1$ is than $x := x + 1$. The assignment statement asserts that nothing but x changes—a concept that cannot be expressed mathematically in any simple way. (Since there are an infinite number of possible variables and a mathematical formula can mention only a finite number of them, a formula cannot assert that no variable other than x changes.) I was therefore happy to eliminate assignment statements. Upon seeing TLA$^+$, almost every computer scientist suggests getting get rid of the UNCHANGED conjuncts, essentially by introducing assignment. Initially, I replied that this would gain little, since removing the UNCHANGED conjuncts would reduce the size of most real specifications by less than 5%. I now point out that the explicit UNCHANGED conjuncts provide valuable redundancy, allowing the model checker to detect the common error of forgetting to specify the new value of a variable.

Like most computer scientists, I assumed that a language should be typed. When I realized that I could eliminate traditional types and let type correctness be an invariant, Martín Abadi encouraged me to do so. Only after I took his advice and started writing untyped specifications did I realize how complicated and constraining types are [2].

When I first started to think about a specification language for TLA, I assumed it would need the usual kinds of programming-language constructs favored by computer scientists. However, I didn't know which ones. I therefore decided to start with only TLA and simple mathematics, and to add other constructs as I needed them. Somewhat to my surprise, I found that all I needed were:

- A few constructs for writing mathematics formally, such as definitions and an IF/THEN/ELSE operator.
- Variable declarations and name scoping, which led to the TLA$^+$ module structure.

Using TLA$^+$

The initial motivation for TLA was to make completely formal, hierarchical correctness proofs of concurrent systems as simple as possible. The development of TLA$^+$ was motivated by the needs of engineers building large systems, for which complete formal development is out of the question. Thus, the TLC model checker was written about 6 years ago, while a project to develop a mechanical proof checker for TLA$^+$ is just starting. (This is in contrast to B,

which was developed for the complete mechanical verification of relatively simple programs.)

The industrial TLA$^+$ specifications I know of have mainly been high-level descriptions of concurrent algorithms or protocols. They have been written to debug the designs (with the aid of TLC) and to serve as documentation. TLA$^+$ specifications have also been used to improve testing of implementations. Randomly generated tests are notoriously inefficient at finding errors in concurrent systems. It is much more effective to guide testing with behaviors generated by TLC from the TLA$^+$ specification [6].

My fundamental objective is to improve the design of systems by getting engineers to think carefully about what they build. I have met with very limited success. Most engineers are looking for tools that can find bugs automatically without requiring any thought. Such tools are useful, but good systems are not built by removing the bugs from poorly designed ones. Thus far, hardware engineers have been the most eager users of TLA$^+$. They are very concerned about errors and are accustomed to using formal tools.

A couple of years ago, I asked Brannon Battson, then a hardware designer at Intel, why he used TLA$^+$. He replied:

> I get asked this question a lot. I randomly select between the following two answers:
>
> 1. It saves a lot of effort to use a high-level language which easily models operations on complex data structures—i.e., select the subset of elements in this set satisfying these conditions and apply this next state equation, etc. Most languages achieve readability of such operations through function encapsulation and other information hiding techniques. But information hiding is the last thing we want in a formal specification. TLA$^+$ provides a powerful set of operators (borrowed from mathematics) which can be used to densely encode complex statements in a readable fashion, without hiding information.
> 2. The next big frontier in computer engineering is algorithmic complexity. In order to tackle this increasingly complex world, we need tools and languages which augment human thought, not supplant it. TLA$^+$ is a language which connects engineers to the underlying mathematics of their design—providing insight which they otherwise wouldn't have.

For an idea of the problems that face designers of complex systems, I recommend trying to solve the Wildfire Challenge Problem [3].

References

[1] Ernie Cohen and Leslie Lamport. Reduction in TLA. In David Sangiorgi and Robert de Simone, editors, *CONCUR'98 Concurrency Theory*, volume

1466 of *Lecture Notes in Computer Science*, pages 317–331. Springer-Verlag, 1998.

[2] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems*, 21(3):502–526, May 1999.

[3] Leslie Lamport, Madhu Sharma, Mark Tuttle, and Yuan Yu. The wildfire verification challenge problem. At URL `http://research.microsoft.com/users/lamport/tla/wildfire-challenge.html` on the World Wide Web. It can also be found by searching the Web for the 24-letter string `wildfirechallengeproblem`.

[4] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, November 1977.

[5] Richard L. Schwartz and P. M. Melliar-Smith. Temporal logic specification of distributed systems. In *Proceedings of the 2nd International Conference on Distributed Computing Systems*, pages 446–454. IEEE Computer Society Press, April 1981.

[6] Serdar Tasiran, Yuan Yu, Brannon Batson, and Scott 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.