TLA$^+$

Leslie Lamport

20 November 2015
minor revision 29 January 2016

Introduction

TLA$^+$ is a high-level specification language that is especially useful for checking designs of distributed and concurrent systems. I developed it at DEC in the 90s, and a model checker for it was written around 2000. TLA$^+$ differs from most specification languages because it's based on simple, ordinary math rather than programming constructs. Around 2005, I designed PlusCal, a language that looks like a simple programming language and is compiled into TLA$^+$. While quite powerful, its resemblance to a programming language keeps PlusCal from being as expressive as TLA$^+$.

The first industrial users of TLA$^+$ were hardware designers at DEC who moved to Intel, where it was still being used in 2014 when my last informant left Intel. Adoption of formal specification by software engineers has been slow. This seems to be changing, and TLA$^+$ use has been slowly spreading in recent years. But I still know of only a handful of users at Microsoft.

I won't try to convince you that TLA$^+$ should be used more widely at Microsoft. I'll leave that to others, starting with Amazon engineers. The following section is a highly abridged version of the paper *How Amazon Web Services Uses Formal Methods* by six Amazon engineers, published in the April 2015 issue of *Communications of the ACM*. Every word in blue is a direct quote from the paper, though much text has been omitted. I have added in black some connecting prose to substitute for deleted material, as well as a few comments marked by "LL". The next section contains abridged quotes from some other TLA$^+$ users. Again, their words are in blue. The concluding section explains my not very hidden agenda.

The Amazon Paper

Since 2011, engineers at Amazon Web Services (AWS) have used formal specification and model checking to help solve difficult design problems in critical systems.

AWS builds systems that are inherently complex. Errors in the core of our system could cause loss or corruption of data. So, we need to reach extremely high confidence that it is correct. We have found the standard verification techniques in industry are necessary but not sufficient. We routinely use deep design reviews, code reviews, static code analysis, stress testing, fault-injection testing, and many other techniques, but we still find that subtle bugs can hide in complex concurrent fault-tolerant systems.

Some of the more subtle, dangerous bugs turn out to be errors in design. We have found that testing the code is inadequate as a method for finding these errors, as the number of reachable states of the code is astronomical.

Precise Designs

In order to find subtle bugs in a system design, it is necessary to have a precise description of that design. The final executable code is unambiguous but contains an overwhelming amount of detail. We needed to be able to capture the essence of a design in a few hundred lines of precise description. As our designs are unavoidably complex, we needed a highly expressive language, far above the level of code, but with precise semantics. And, as we wish to build services quickly, we wanted a language that is simple to learn and apply, avoiding esoteric concepts. We also wanted an existing ecosystem of tools. We were looking for an off-the-shelf method with high return on investment.

We found what we were looking for in TLA$^+$, a formal specification language based on simple discrete math, or basic set theory and predicates, with which all engineers are familiar. We found it helpful that the same language is used to describe both the desired correctness properties of the system (the "what") and the design of the system (the "how"). In TLA$^+$, correctness properties and system designs are just steps on a ladder of abstraction, with correctness properties occupying higher levels, systems designs and algorithms in the middle, and executable code and hardware at the lower levels. TLA$^+$ is intended to make it as easy as possible to show that a system design correctly implements the desired correctness properties, either through conventional mathematical reasoning or more easily and quickly through tools like the TLC model checker that exhaustively checks the desired correctness

properties across all possible execution traces. The ladder of abstraction also helps designers manage the complexity of real-world systems. The freedom to choose and adjust levels of abstraction makes TLA$^+$ extremely flexible.

TLA$^+$ is accompanied by a second language called PlusCal that is closer to a C-style programming language but is much more expressive, as it uses TLA$^+$ for expressions and values. Several engineers at Amazon have found they are more productive using PlusCal than using TLA$^+$. In other cases, the additional flexibility of plain TLA$^+$ has been very useful.

Formal Methods for Real-World Systems

In industry, formal methods have a reputation for requiring a huge amount of training and effort to verify a tiny piece of relatively straightforward code, so the return on investment is justified only in safety-critical domains. Our experience with TLA$^+$ shows that perception to be wrong. At the time of this writing, Amazon engineers have used TLA$^+$ on 10 large complex real-world systems. LL: By last spring, the number had grown to 14. In each, TLA$^+$ has added significant value, either finding subtle bugs that we are sure we would not have found by other means, or giving us enough understanding and confidence to make aggressive performance optimizations without sacrificing correctness. Amazon now has seven teams using TLA$^+$, with encouragement from senior management and technical leadership. Engineers from entry level to principal have been able to learn TLA$^+$ from scratch and get useful results in two to three weeks, in some cases in their personal time on weekends and evenings, without further help or training.

Side Benefit

TLA$^+$ changes the conventional approach of thinking about what can go wrong to specifying what must go right for the system to achieve its goals. We have found this rigorous "what needs to go right" approach to system design to be significantly less error prone than the ad hoc "what might go wrong" approach.

More Side Benefits

Writing a formal specification pays dividends over the lifetime of the system. All production services at Amazon are under constant development. Our first priority is always to avoid causing bugs in a production system, so we often have to answer the question: "Is this change safe?" A major benefit of having a precise, testable model of the core system is that we

can quickly verify that even deep changes are safe or learn that they are unsafe without doing harm. In several cases, we have prevented subtle but serious bugs from reaching production. In other cases we have been able to make innovative performance optimizations that we would not have dared to do without having model-checked those changes. LL: Users at Intel also reported that model checking enabled optimizations they would not otherwise have made. We regularly have to bring new people up to speed on systems. To avoid creating subtle bugs, we need all engineers to have the same mental model of the system and for that shared model to be accurate, precise, and complete. A formal specification is precise, short, and can be explored and experimented upon with tools.

First Steps to Formal Methods

With hindsight, Amazon's path to formal methods seems straightforward; we had an engineering problem and we found a solution. Reality was somewhat different.

The paper here describes what led author Chris Newcombe to TLA$^+$ after trying a different method that was not up to the task. Newcombe then tried to persuade colleagues at Amazon to adopt TLA$^+$. However, engineers have almost no spare time for such things, unless compelled by need. Fortunately, a need was about to arise.

First Big Success at Amazon

In January 2012, Amazon launched DynamoDB. The replication and fault-tolerance mechanisms in DynamoDB were created by author T.R. He did extensive testing and wrote detailed informal correctness proofs. To achieve the highest level of confidence in the design, T.R. chose to apply TLA$^+$.

T.R. learned TLA$^+$ and wrote a detailed specification of these components in a couple of weeks. The model checker verified that a small, complicated part of the algorithm worked as expected. T.R. then checked the broader fault-tolerant algorithm. This time the model checker found a bug that could lead to losing data if a particular sequence of failures and recovery steps was interleaved with other processing. This was a very subtle bug; the shortest error trace exhibiting the bug contained 35 high-level steps. The improbability of such compound events is not a defense against such bugs; historically, AWS engineers have observed many combinations of events at least as complicated as those that could trigger this bug. The bug had passed unnoticed through extensive design reviews, code reviews, and testing, and

T.R. is convinced that we would not have found it by doing more work in those conventional areas. The model checker later found two bugs in other algorithms, both serious and subtle.

T.R. says that, had he known about TLA$^+$ before starting work on DynamoDB he would have used it from the start. Using TLA$^+$ would likely have improved time to market, in addition to achieving greater confidence in correctness of the system.

Persuading More Engineers

The article describes the further spread of TLA$^+$. It observes that: TLA$^+$ can be taught by engineers who are still new to it themselves; this is important for quickly scaling adoption in an organization as large as Amazon.

Most Frequently Asked Question

On learning about TLA$^+$, engineers usually ask, "How do we know that the executable code correctly implements the verified design?" The answer is we do not know. Despite this, formal methods help in multiple ways:

Get design right. Formal methods help engineers get the design right, which is a necessary first step toward getting the code right. If the design is broken, then the code is almost certainly broken. Engineers are unlikely to realize the design is incorrect while focused on coding;

Gain better understanding. Formal methods help engineers gain a better understanding of the design. Improved understanding can only increase the chances they will get the code right; and

Write better code. Formal methods can help engineers write better "self-diagnosing code" in the form of assertions. Formal methods can help improve assertions that help improve the quality of code.

Conclusion

Formal methods are a big success at AWS, helping us prevent subtle but serious bugs from reaching production, bugs we would not have found via any other technique. They have helped us devise aggressive optimizations to complex algorithms without sacrificing quality. At the time of this writing, seven Amazon teams have used TLA$^+$, all finding value in doing so. More Amazon teams are starting to use TLA$^+$. We believe that use of TLA$^+$ will improve both time-to-market and quality of our systems. Executive management is now actively encouraging teams to write TLA$^+$ specs for new

features and other significant design changes. In annual planning, managers now allocate engineering time to TLA$^+$.

Comments from Other Users

The Rosetta Spacecraft

TLA$^+$ was used in designing a real-time operating system now flying on the Rosetta spacecraft orbiting comet 67P/Churyumov-Gerasimenko. Eric Verhulst, the lead developer of that system wrote:

The TLA$^+$ abstraction helped a lot in coming to a much cleaner architecture (we witnessed first hand the brain washing done by years of C programming). One of the results was that the code size is about 10x less than in an earlier version.

Intel

When asked why Intel used TLA$^+$, Brannon Batson, a former Intel engineer, answered:

1. It saves a lot of effort to use a high-level language which easily models operations on complex data structures. TLA$^+$ provides a powerful set of operators which can be used to densely encode complex statements in a readable fashion, without hiding important information.

2. In order to tackle our 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.

Xbox 360

Chuck Thacker reports:

During the development of the Xbox 360, I was working with the engineering group on the memory coherence protocol. Working with an intern, we were developing a TLA$^+$ model for the protocol. In the course of this, we discovered a very subtle bug. We reported it to IBM, and they told us it couldn't happen. A couple of weeks later, they relented and told us that not only was the bug real, but that their regression tests wouldn't have found it. Had they not fixed it, they would have shipped us chips that would have deadlocked after about four hours of use.

Had this not been done, the schedule for a Christmas launch would almost certainly have been missed. The console business is cyclic, with most

consoles sold in the three months around Christmas. Sony had similar problems two years in a row, allowing Microsoft to dominate this market. How much money was involved? It is difficult to say, but without TLA$^+$/TLC, we would have found out.

Azure

Cheng Huang and Huseyin Simitci report the following use of TLA$^+$ to fix a serious Azure bug that could have resulted in customer data unavailability:

We extracted the replication and erasure coding logic from the production system and represented it as a TLA$^+$ specification. We checked the specification using the TLC model checker, which reproduced the problem in minutes. Confirming the problem gave us confidence that our specification was faithfully representing the production system. We augmented our specification with the fix, and TLC checked it in several days on a cluster of 10 powerful servers. This gave us high confidence that the proposed fix did solve the original problem without introducing any new ones. We felt confident in implementing and deploying the fix in production.

Conclusion

Hardware engineers care about correctness; they are used to writing specifications and applying tools to check them. Modern processor chips are complex distributed message-passing systems, and the tools they were using can't find design errors in them. TLA$^+$ allowed Intel engineers to write formal specifications of designs that they had been able to specify only with prose and pseudo-code.

Software engineers have cared little about errors, which they considered mere "bugs" that can be found by testing and fixed with simple patches. They have been slow to realize that this isn't true for concurrent and distributed systems. Testing is ineffective at catching concurrency errors; patches seldom fix all instances of an error and are likely to introduce new errors. Such errors must be eliminated at the design level using high-level specifications and tools—the most useful tool being a model checker.

There are a number of specification languages that address this problem. Some are good and some are mostly harmful. TLA$^+$ is different from most of them because writing a TLA$^+$ spec requires learning to use math and to think mathematically. This improves your thinking. Engineers start using TLA$^+$ so they can debug their designs with the model checker. Only later do they realize that TLA$^+$ does more than that. Brannon Batson, a former

Intel engineer, writes:

> The hard part of learning to write TLA$^+$ specs is learning to think abstractly about the system. With experience, engineers learn how to do it. Being able to think abstractly improves their design process.

Dave Langworthy, a Microsoft software engineer, puts it more simply:

> TLA$^+$ taught me how to think.

Teaching engineers how to think will do more to improve the quality of their software than any tool.