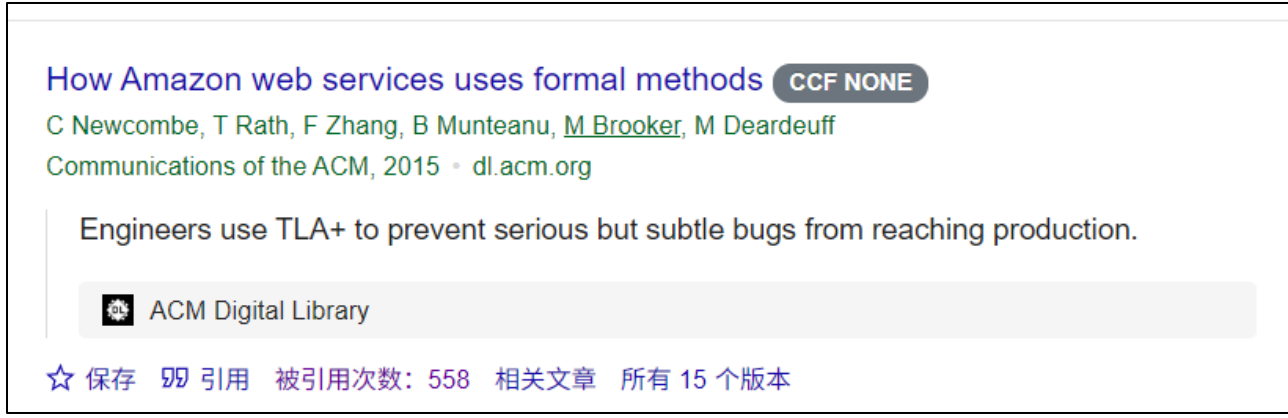

How Amazon Web Services Uses Formal Methods

Newcombe C, Rath T, Zhang F, et al. How Amazon web services uses formal methods[J]. Communications of the ACM, 2015, 58(4): 66-73.

key insights

Formal methods find bugs in system designs that cannot be found through any other technique we know of.

Formal methods are surprisingly feasible for mainstream software development and give good return on investment (投资回报).

At Amazon, formal methods are routinely applied to the design of complex real-world software, including public cloud services.

Simple Storage Service (S3) is just one of many Amazon Web Services (AWS) services that store and process data our customers have entrusted to us.

To safeguard that data, the core of each service relies on fault-tolerant distributed algorithms for replication, consistency, concurrency control, auto-scaling, load balancing, and other coordination tasks.

There are many such algorithms in the literature, but combining them into a cohesive system is a challenge, as the algorithms must usually be modified to interact properly in a real-world system.

In addition, we have found it necessary to invent algorithms of our own. We work hard to avoid unnecessary complexity, but the essential complexity of the task remains high.

Complexity increases the probability of human error in design, code, and operations. Errors in the core of the system could cause loss or corruption of data, or violate other interface contracts on which our customers depend.
So, before launching a service, we need to reach extremely high confidence that the core of the system 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, and fault-injection testing but still find that subtle bugs can hide in complex concurrent fault-tolerant systems.

One reason they do is that human intuition is poor at estimating the true probability of supposedly “extremely rare” combinations of events in systems operating at a scale of millions of requests per second.

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

At AWS, we strive to build services that are simple for customers to use. External simplicity is built on a hidden substrate of complex distributed systems. Such complex internals are required to achieve high availability while running on cost-efficient infrastructure and cope with relentless business growth.

As an example of this growth, in 2006, AWS launched S3. In the following six years, S3 grew to store one trillion objects. Less than a year later it had grown to two trillion objects and was regularly handling 1.1 million requests per second.

NASA's C. Michael Holloway says, "To a first approximation, we can say that accidents are almost always the result of incorrect estimates of the likelihood of one or more things."

Human fallibility means some of the more subtle, dangerous bugs turn out to be errors in design; the code faithfully implements the intended design, but the design fails to correctly handle a particular "rare" scenario.

We have found that testing the code is inadequate as a method for finding subtle errors in design, as the number of reachable states of the code is astronomical.

So we look for a better approach.

In order to find subtle bugs in a system design, it is necessary to have a precise description of that design. There are at least two major benefits to writing a precise design: the author is forced to think more clearly, and tools can be applied to check for errors in the design, even while it is being written.

As our designs are unavoidably complex, we needed a highly expressive language, far above the level of code, but with precise semantics.

That expressivity must cover real-world concurrency and fault tolerance. And, as we wish to build services quickly, we wanted a language that is simple to learn and apply, avoiding esoteric concepts. We also very much wanted an existing ecosystem of tools. We were thus 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.

A TLA+ specification describes the set of all possible legal behaviors, or execution traces, of a system. 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.

tlaplus/tlaplus: TLC is a model checker for specifications written in TLA+. The TLA+Toolbox is an IDE for TLA+.

TLA+ is intended to make it as easy as possible to show a system design correctly implements the desired correctness properties, through either conventional mathematical reasoning or tools like the TLC model checker that take a TLA+ specification and 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; designers may choose to describe the system at several “middle” levels of abstraction, with each lower level serving a different purpose (such as to understand the consequences of concurrency or more detailed behavior of a communication medium).

The designer can then verify that each level is correct with respect to a higher level. The freedom to choose and adjust levels of abstraction makes TLA+ extremely flexible.

At first, the syntax and idioms of TLA+ are somewhat unfamiliar to programmers.

Fortunately, TLA+ is accompanied by a second language called PlusCal that is closer to a C-style programming language but much more expressive, as it uses TLA+ for expressions and values.

PlusCal is intended to be a direct replacement for pseudo-code.

Several engineers at Amazon have found they are more productive using PlusCal than they are using TLA+.

However, in other cases, the additional flexibility of plain TLA+ has been very useful.

For many designs the choice is a matter of taste, as PlusCal is automatically translated to TLA+ with a single key press.

PlusCal users do have to be familiar with TLA+ in order to write rich expressions and because it is often helpful to read the TLA+ translation to understand the precise semantics of a piece of code.

Moreover, tools (such as the TLC model checker) work at the TLA+ level.

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 (such as medical systems and avionics).

Our experience with TLA+ shows this perception to be wrong. At the time of this writing, Amazon engineers have used TLA+ on 10 large complex real-world systems.

In each, TLA+ has added significant value, either finding subtle bugs 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.

TLA+ has been helping us shift to a better way of designing systems. Engineers naturally focus on designing the “happy case” for a system, or the processing path in which no errors occur.

This is understandable, as the happy case is by far the most common case.

That code path must solve the customer’s problem, perform well, make efficient use of resources, and scale with the business -- all significant challenges in their own right.

When the design for the happy case is done, the engineer then tries to think of “what could go wrong” based on personal experience and that of colleagues and reviewers.

The engineer then adds mitigations for these scenarios, prioritized by intuition and perhaps statistics on the probability of occurrence.

Almost always, the engineer stops well short of handling “extremely rare” combinations of events, as there are too many such scenarios to imagine.

In contrast, when using formal specification we begin by stating precisely “what needs to go right.”

We first specify what the system should do by defining correctness properties, which come in two varieties:

Safety. What the system is allowed to do. For example, at all times, all committed data is present and correct, or equivalently; at no time can the system have lost or corrupted any committed data;

Liveness. What the system must eventually do. For example, whenever the system receives a request, it must eventually respond to that request.

After defining correctness properties, we then precisely describe an abstract version of the design, along with an abstract version of its operating environment.

We express “what must go right” by explicitly specifying all properties of the environment on which the system relies. Examples of such properties might be “If a communication channel has not failed, then messages will be propagated along it”.

Next, with the goal of confirming our design correctly handles all dynamic events in the environment, we specify the effects of each of those possible events -- network errors and repairs, disk errors, process crashes and restarts, data-center failures and repairs, and actions by human operators.

We then use the model checker to verify that the specification of the system in its environment implements the chosen correctness properties, despite any combination or interleaving of events in the operating environment. We find this rigorous “what needs to go right” approach to be significantly less error prone than the ad hoc “what might go wrong” approach.

We also find that writing a formal specification pays dividends (回报) over the lifetime of the system.

All production services at Amazon are under constant development, even those released years ago; we add new features customers have requested, we redesign components to handle massive increases in scale, and we improve performance by removing bottlenecks.

Many of these changes are complex and must be made to the running system with no downtime.

Our first priority is always to avoid causing bugs in a production system, so we often have to answer “Is this change safe?”

We find 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 they are unsafe without doing harm.

In several cases, we have prevented subtle but serious bugs from reaching production.

In addition, a precise, testable, well commented description of a design is an excellent form of documentation, which is important, as AWS systems have unbounded lifetimes.

Over time, teams grow as the business grows, so we regularly have to bring new people up to speed on systems. This education must be effective.

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.

Engineers form mental models in various ways—talking to each other, reading design documents, reading code, and implementing bug fixes or small features.

But talk and design documents can be ambiguous or incomplete, and the executable code is much too large to absorb quickly and might not precisely reflect the intended design.

In contrast, a formal specification is precise, short, and can be explored and experimented on with tools.

Applying TLA+ to some of Amazon's more complex systems.

System	Components	Line Count (Excluding Comments)	Benefit
S3	Fault-tolerant, low-level network algorithm	804 PlusCal	Found two bugs, then others in proposed optimizations
	Background redistribution of data	645 PlusCal	Found one bug, then another in the first proposed fix
DynamoDB	Replication and group-membership system	939 TLA+	Found three bugs requiring traces of up to 35 steps
EBS	Volume management	102 PlusCal	Found three bugs
Internal distributed lock manager	Lock-free data structure	223 PlusCal	Improved confidence though failed to find a liveness bug, as liveness not checked
	Fault-tolerant replication-and-reconfiguration algorithm	318 TLA+	Found one bug and verified an aggressive optimization

We are concerned with two major classes of problems with large distributed systems: bugs and operator errors that cause a departure (违背) from the system's logical intent; and surprising "sustained emergent performance degradation" of complex systems that inevitably contain feedback loops.

We know how to use formal specification to find problems in the first class. However, problems in the second class can cripple a system even though no logic bug is involved.

A common example is when a momentary slowdown in a server (due, perhaps, to Java garbage collection) causes timeouts to be breached on clients, causing the clients to retry requests, thus adding load to the server, and further slowdown.

In such scenarios the system eventually makes progress; it is not stuck in a logical deadlock, livelock, or other cycle.

But from the customer’s perspective it is effectively unavailable due to sustained unacceptable response times.

TLA+ can be used to specify an upper bound on response time, as a real-time safety property. However, AWS systems are built on infrastructure—disks, operating systems, network—that does not support hard real-time scheduling or guarantees, so real-time safety properties would not be realistic.

We build soft real-time systems in which very short periods of slow responses are not considered errors. However, prolonged severe slowdowns are considered errors.

We do not yet know of a feasible way to model a real system that would enable tools to predict such emergent behavior. We use other techniques to mitigate these risks.

The fact that TLA+ was created by the designer of such a widely used algorithm gave us some confidence that TLA+ would work for real-world systems.

We became more confident when we learned a team of engineers at DEC/Compaq had used TLA+ to specify and verify some intricate cache-coherency protocols for the Alpha series of multicore CPUs.

We read one of the specifications and found they were sophisticated distributed algorithms involving rich message passing, fine-grain concurrency, and complex correctness properties.

We knew from Lamport’s Fast Paxos paper that TLA+ could model fault tolerance at a high level of abstraction and were further convinced when we found other papers showing TLA+ could model lower-level failures.

In January 2012, Amazon launched DynamoDB, a scalable high-performance “no SQL” data store that replicates customer data across multiple data centers while promising strong consistency.

This combination of requirements leads to a large, complex system. The replication and fault-tolerance mechanisms in DynamoDB were created by author T.R.

To verify correctness of the production code, T.R. performed extensive fault-injection testing using a simulated network layer to control message loss, duplication, and reordering.

The system was also stress tested for long periods on real hardware under many different workloads.

We know such testing is absolutely necessary but can still fail to uncover subtle flaws in design.

To verify the design of DynamoDB, T.R. wrote detailed informal proofs of correctness that did indeed find several bugs in early versions of the design.

However, we have also learned that conventional informal proofs can miss very subtle problems.

To achieve the highest level of confidence in the design, T.R. chose TLA+. T.R. learned TLA+ and wrote a detailed specification of these components in a couple of weeks.

To model check the specification, we used the distributed version of the TLC model checker running on a cluster of 10 cc1.4xlarge EC2 instances, each with eight cores plus hyperthreads and 23GB of RAM.

The model checker verified that a small, complicated part of the algorithm worked as expected for a sufficiently large instance of the system to give high confidence it is correct.

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 would be interleaved with other processing.

This was a very subtle bug; the shortest error trace exhibiting the bug included 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 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. fixed all these bugs, and the model checker verified the resulting algorithms to a very high degree of confidence.

T.R. says that, had he known about TLA+ before starting work on DynamoDB he would have used it from the start.

He believes the investment he made in writing and checking the formal TLA+ specifications was more reliable and less time consuming than the work he put into writing and checking his informal proofs.

Using TLA+ in place of traditional proof writing would thus likely have improved time to market, in addition to achieving greater confidence in the system's correctness.

Success with DynamoDB gave us enough evidence to present TLA+ to the broader engineering community at Amazon. This raised a challenge—how to convey the purpose and benefits of formal methods to an audience of software engineers.

Engineers think in terms of debugging rather than "verification," so we called the presentation "Debugging Designs."

Continuing the metaphor, we have found that software engineers more readily grasp the concept and practical value of TLA+ if we dub it "exhaustively testable pseudo-code."

We initially avoid the words "formal" "verification" and "proof" due to the widespread view that formal methods are impractical. We also initially avoid mentioning what TLA (Temporal Logic of Actions) stands for, as doing so would give an incorrect impression of complexity.

A team working on S3 asked for help using TLA+ to verify a new fault-tolerant network algorithm.

The documentation for the algorithm consisted of many large, complicated state-machine diagrams.

To check the state machine, the team had been considering writing a Java program to brute-force explore possible executions: essentially a hard-wired form of model checking.

They were able to avoid the effort by using TLA+ instead. Author F.Z. wrote two versions of the spec over a couple of weeks.

For this particular problem, F.Z. found that she was more productive in PlusCal than TLA+, and we have observed that engineers often find it easier to begin with PlusCal.

Model checking revealed two subtle bugs in the algorithm and allowed F.Z. to verify fixes for both.

F.Z. then used the spec to experiment with the design, adding new features and optimizations.

The model checker quickly revealed that some of these changes would have introduced bugs.

Engineers at Amazon continue to use TLA+, adopting the practice of first writing a conventional prose-design document, then incrementally refining parts of it into PlusCal or TLA+.

This method often yields important insight about the design, even without going as far as full specification or model checking.

In one case, C.N. refined a prose design of a fault-tolerant replication system that had been designed by another Amazon engineer.

C.N. wrote and model checked specifications at two levels of concurrency; these specifications helped him understand the design well enough to propose a major protocol optimization that radically reduced write-latency in the system.

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 still help in multiple ways:

1. 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, as mistakes during coding are extremely unlikely to compensate for mistakes in design.

Worse, engineers are likely to be deceived into believing the code is “correct” because it appears to correctly implement the (broken) design. Engineers are unlikely to realize the design is incorrect while focused on coding.

2. 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.

3. Write better code. Formal methods can help engineers write better “self-diagnosing code” in the form of assertions.

Independent evidence and our own experience suggest pervasive use of assertions is a good way to reduce errors in code. An assertion checks a small, local part of an overall system invariant.

A good system invariant captures the fundamental reason the system works; the system will not do anything wrong that could violate a safety property as long as it continuously maintains the system invariant.

The challenge is to find a good system invariant, one strong enough to ensure no safety properties are violated.

Formal methods help engineers find strong invariants, so formal methods can help improve assertions that help improve the quality of code.

While we would like to verify that executable code correctly implements the high-level specification or even generate the code from the specification, we are not aware of any such tools that can handle distributed systems as large and complex as those being built at Amazon.

We do routinely use conventional static analysis tools, but they are largely limited to finding "local" issues in the code, and are unable to verify compliance with a high-level specification.

We have seen research on using the TLC model checker to find "edge cases" in the design on which to test the code, an approach that seems promising.

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 through 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, and more Amazon teams are starting to use it.

Using TLA+ will improve both time-to-market and quality of our systems. Executive management actively encourages teams to write TLA+ specs for new features and other significant design changes. In annual planning, managers now allocate engineering time to TLA+.

While our results are encouraging, some important caveats remain.

Formal methods deal with models of systems, not the systems themselves.

The designer must ensure the model captures the significant aspects of the real system. Achieving it is a special skill, the acquisition of which requires thoughtful practice.

Also, we were solely concerned with obtaining practical benefits in our particular problem domain and have not attempted a comprehensive survey.

Therefore, mileage may vary with other tools or in other problem domains.