Use of Formal Methods at Amazon Web Services

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, Michael Deardeuff
Amazon.com

29th September, 2014

Since 2011, engineers at Amazon Web Services (AWS) have been using formal specification and model checking to help solve difficult design problems in critical systems. This paper describes our motivation and experience, what has worked well in our problem domain, and what has not. When discussing personal experiences we refer to authors by their initials.

At AWS we strive to build services that are simple for customers to use. That 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 also to cope with relentless rapid business growth. As an example of this growth; in 2006 we launched S3, our Simple Storage Service. In the 6 years after launch, S3 grew to store 1 trillion objects [1]. Less than a year later it had grown to 2 trillion objects, and was regularly handling 1.1 million requests per second [2].

S3 is just one of tens of AWS services that store and process data that 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 major challenge, as the algorithms must usually be modified in order 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.

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 such a service, we need to reach extremely high confidence that the core of the system is correct. We have found that the standard verification techniques in industry are necessary but not sufficient. We 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. One reason for this problem 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.

> "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." - C. Michael Holloway, NASA [3]

That human fallibility means that 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 to find subtle

Use of Formal Methods at Amazon Web Services

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, Michael Deardeuff
Amazon.com

29th September, 2014

Since 2011, engineers at Amazon Web Services (AWS) have been using formal specification and model checking to help solve difficult design problems in critical systems. This paper describes our motivation and experience, what has worked well in our problem domain, and what has not. When discussing personal experiences we refer to authors by their initials.

At AWS we strive to build services that are simple for customers to use. That 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 also to cope with relentless rapid business growth. As an example of this growth; in 2006 we launched S3, our Simple Storage Service. In the 6 years after launch, S3 grew to store 1 trillion objects [1]. Less than a year later it had grown to 2 trillion objects, and was regularly handling 1.1 million requests per second [2].

S3 is just one of tens of AWS services that store and process data that 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 major challenge, as the algorithms must usually be modified in order 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.

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 such a service, we need to reach extremely high confidence that the core of the system is correct. We have found that the standard verification techniques in industry are necessary but not sufficient. We 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. One reason for this problem 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.

> "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." - C. Michael Holloway, NASA [3]

That human fallibility means that 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 to find subtle

errors in design, as the number of reachable states of the code is astronomical. So we looked for a better approach.

Precise Designs

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, which helps eliminate 'plausible hand-waving', and tools can be applied to check for errors in the design, even while it is being written. In contrast, conventional design documents consist of prose, static diagrams, and perhaps pseudo-code in an ad hoc untestable language. Such descriptions are far from precise; they are often ambiguous, or omit critical aspects such as partial failure or the granularity of concurrency (i.e. which constructs are assumed to be atomic). At the other end of the spectrum, 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. 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. In summary, we were looking for an off-the-shelf method with high return on investment.

We found what we were looking for in TLA+ [4], a formal specification language. TLA+ is based on simple discrete math, i.e. basic set theory and predicates, with which all engineers are familiar. A TLA+ specification describes the set of all possible legal behaviors (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. TLA+ is intended to make it as easy as possible to show that a system design correctly implements the desired correctness properties, either via conventional mathematical reasoning, or more easily and quickly by using tools such as the TLC model checker [5], a tool which takes a TLA+ specification and exhaustively checks the desired correctness properties across all of the possible execution traces. The ladder of abstraction also helps designers to manage the complexity of real-world systems; the designer may choose to describe the system at several 'middle' levels of abstraction, with each lower level serving a different purpose, such as understanding the consequences of finer-grain 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 easily 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 which is closer to a C-style programming language, but much more expressive as it uses TLA+ for expressions and values. In fact, PlusCal is intended to be a direct replacement for pseudo-code. Several engineers at Amazon have found they are more productive in PlusCal than 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 need 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. Also, tools such as the TLC model checker work at the TLA+ level.

The Value of Formal Methods for 'Real-world Systems'

In industry, formal methods have a reputation of requiring a huge amount of training and effort to verify a tiny piece of relatively straightforward code, so the return on investment is only justified in safety-critical domains such as medical systems and avionics. Our experience with TLA+ has shown that perception to be quite wrong. So far we have used TLA+ on 10 large complex real-world systems. In every case 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. We now have 7 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 2 to 3 weeks, in some cases just in their personal time on weekends and evenings, and without help or training.

We lack space here to explain the TLA+ language or show any real-world specifications. We have not included any snippets of specifications because their unfamiliar syntax can be off-putting to potential new users. We've found that potential new users benefit from hearing about the value of formal methods in industry before tackling tutorials and examples. We refer readers to [4] for tutorials, and [12] for an example of a TLA+ specification from industry that is similar in size and complexity to some of the larger specifications at Amazon.

Applying TLA+ to some of our more complex systems

System	Components	Line count (excl. comments)	Benefit
S3	Fault-tolerant low-level network algorithm	804 PlusCal	Found 2 bugs. Found further bugs in proposed optimizations.
	Background redistribution of data	645 PlusCal	Found 1 bug, and found a bug in the first proposed fix.
DynamoDB	Replication & group-membership system	939 TLA+	Found 3 bugs, some requiring traces of 35 steps
EBS	Volume management	102 PlusCal	Found 3 bugs.
Internal distributed lock manager	Lock-free data structure	223 PlusCal	Improved confidence. Failed to find a liveness bug as we did not check liveness.
	Fault tolerant replication and reconfiguration algorithm	318 TLA+	Found 1 bug. Verified an aggressive optimization.

We have found TLA+ to be effective in our problem domain, but there are many other formal specification languages and tools. Some of those alternatives are described later.

Side Benefit: A Better Way to Design Systems

TLA+ has been helping us shift to a better way of designing systems. Engineers naturally focus on designing the 'happy case' for a system, i.e. 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; these are all significant challenges in their own right. Once the design for the happy case is done, the engineer then tries to think of "what might go wrong?", based on personal experience and that of colleagues and reviewers. The engineer then adds mitigations for these classes of scenarios, prioritized by intuition and perhaps some 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 precisely stating "what needs to go right?" First we state what the system should do, by defining correctness properties. These come in two varieties:

- Safety properties: "what the system is *allowed* to do"
  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 properties: "what the system *must eventually* do"
  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 of the 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.", and "If a process has not restarted, then it retains its local state, modulo any intentional modifications." Next, with the goal of confirming that our design correctly handles all of the dynamic events in the environment, we specify the effects of each of those possible events; e.g. 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 have found this rigorous "what needs to go right?" approach to be significantly less error prone than the ad hoc "what might go wrong?" approach.

More Side Benefits: Improved Understanding, Productivity and Innovation

We have found that writing a formal specification pays several dividends over the lifetime of the system. All production services at Amazon are under constant development, even those released years ago; we add new features that customers have requested, we re-design components to handle massive increases in scale, and we improve performance by removing bottlenecks. Many of these changes are complex, and they 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 need to answer the question, “is this change safe?” We have found that a major benefit of having a precise, testable model of the core system is that we can rapidly verify that even deep changes are safe, or learn that they are unsafe without doing any harm. In several cases we have prevented subtle, serious bugs from reaching production. In other cases we have been able to make innovative performance optimizations – e.g. removing or narrowing locks, or weakening constraints on message ordering – which we would not have dared to do without having model checked those changes. A precise, testable description of a system becomes a “what if …” tool for designs, analogous to how spread sheets are a “what if …” tool for financial models. We have found that using such a tool to explore the behavior of the system can give the designer an improved understanding of the system.

In addition, a precise, testable, well commented description of a design is an excellent form of documentation. Documentation is very important as our systems have unbounded lifetime. Over time, teams grow as the business grows, so we regularly have to bring new people up to speed on systems. We need this education to be effective. To avoid creating subtle bugs, we need all of the 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 a variety of 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 far 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 upon with tools.

What Formal Specification Is Not Good For

We are concerned with two major classes of problems with large distributed systems: 1) bugs and operator errors that cause a departure from the logical intent of the system, and 2) surprising ‘sustained emergent performance degradation’ of complex systems that inevitably contain feedback loops. We know how to use formal specification to find the first class of problems. However, problems in the second category can cripple a system even though no logic bug is involved. A common example is when a momentary slowdown in a server (perhaps due to Java garbage collection) causes timeouts to be breached on clients, which causes the clients to retry requests, which adds more load to the server, which causes further slowdown. In such scenarios the system will eventually make 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+ could be used to specify an upper bound on response time, as a real-time safety property. However, our systems are built on infrastructure (disks, operating systems, network) that do 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 don’t 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 those risks.

First Steps To Formal Methods

With hindsight, our path to formal methods seems clear and straightforward; we had an engineering problem and we found a solution. The reality was somewhat different. The effort began with author C.N.'s dissatisfaction with the quality of several distributed systems he had designed and reviewed, and with the development process and tools that had been used to construct those systems. The systems were considered very successful, and yet bugs and operational problems still remained. To mitigate those problems, the systems used well proven methods (pervasive contract assertions enabled in production) to detect symptoms of bugs, and mechanisms such as 'recovery-oriented computing' [6] to attempt to minimize the impact when bugs were triggered. However, reactive mechanisms cannot recover from the class of bugs that cause permanent damage to customer data; instead, we must prevent such bugs.

When looking for techniques to prevent bugs, C.N. did not initially consider formal methods, due to the strong pervasive view that they are only suitable for tiny problems and give very low return on investment. Overcoming the bias against formal methods required evidence that they work on real-world systems. This evidence was provided in a paper by Pamela Zave [7]. Zave used a language called Alloy to find serious bugs in the membership protocol of a distributed system called Chord. Chord was designed by a strong group at MIT and is certainly successful; it won a '10-year test of time' award at SIGCOMM 2011, and has influenced several systems in industry. Zave's success motivated C.N. to perform an evaluation of Alloy, by writing and model checking a moderately large Alloy specification of a non-trivial concurrent algorithm [8]. We liked many characteristics of the Alloy language; e.g. the emphasis on 'execution traces' of abstract system states composed of sets and relations. However, we found that Alloy is not expressive enough for many of our use cases. For instance, we could not find a practical way in Alloy to represent rich data structures such as dynamic sequences containing nested records with multiple fields.

Alloy's limited expressivity appears to be a consequence of the particular approach to analysis taken by the Alloy Analyzer tool. The limitations do not seem to be caused by Alloy's conceptual model ('execution traces' over system states). This hypothesis motivated C.N. to look for a language with a similar conceptual model but with richer constructs for describing system states. Eventually C.N. stumbled across a language with those properties when he found a TLA+ specification in the appendix of a paper on a canonical algorithm in our problem domain: the Paxos consensus algorithm [9].

The fact that TLA+ was created by the designer of such a widely used algorithm gave us some confidence that TLA+ worked for real-world systems. We became more confident when we learned that a team of engineers at DEC/Compaq had used TLA+ to specify and verify some intricate cache-coherency protocols for the Alpha series of multi-core CPUs [10][11]. We read one of the specifications [12] and found that these were sophisticated distributed algorithms, involving rich message passing, fine-grain concurrency, and complex correctness properties. That only left the question of whether TLA+ could handle 'real world' failure modes. (The Alpha cache-coherency algorithm does not consider failure.) We knew from the Fast Paxos paper [9] that TLA+ could model fault tolerance at a high level of abstraction, but we were further convinced when we found other papers that showed that TLA+ could model lower-level failures [13].

C.N. evaluated TLA+ by writing a specification of the same non-trivial concurrent algorithm that he had written in Alloy [8]. Both Alloy and TLA+ were able to handle that problem, but this comparison revealed that TLA+ is much more expressive than Alloy. This difference has been important in practice; several of the real-world specifications we have written in TLA+ would have been infeasible or impossible in Alloy. We initially had the opposite concern about TLA+; it is so expressive that no model checker can hope to evaluate everything that can be expressed in the language. But so far we have always been able to find a way to express our intent in a way that is clear, direct, and can be model checked.

After evaluating Alloy and TLA+, C.N. tried to persuade colleagues 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, a scalable high-performance 'No SQL' data store, which replicates customer data across multiple data centers while promising strong consistency [14]. 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 re-ordering. The system was also stress tested for long periods on real hardware under many different workloads. We know that such testing is absolutely necessary, but can still fail to uncover subtle flaws in design. To verify the design, T.R. wrote detailed informal proofs of correctness. Those proofs 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 [15]. 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. To model check the specification we used the distributed version of the TLC model checker, running on a cluster of ten cc1.4xlarge EC2 instances, each with 8 cores plus hyperthreads, and 23 GB 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 very high confidence that it is correct. T.R. then moved on to checking 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 has 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, again both serious and subtle. T.R. fixed all of 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 that the investment he made in writing and checking the formal TLA+

specifications was both more reliable, and also less time consuming than the work he put into writing and checking his informal proofs. Therefore, using TLA+ in place of traditional proof writing would likely have improved time-to-market in addition to achieving higher confidence on system correctness

After DynamoDB was launched, T.R. worked on a new feature to allow data to be migrated between data centers. As he already had the specification for the existing replication algorithm, T.R. was able to quickly incorporate this new feature into the specification. The model checker found that the initial design would have introduced a subtle bug, but this was easy to fix, and the model checker verified the resulting algorithm to the necessary level of confidence. T.R. continues to use TLA+ and model checking to verify changes to the design, both for optimizations and new features.

Persuading More Engineers Leads to Further Successes

The 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" [8]. Continuing that 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 the acronym 'TLA' stands for, as doing so would give an incorrect impression of complexity.

Immediately after seeing the presentation, a team working on S3 asked for help to use 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 that 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 in general 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 bugs. 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.

This success led to management advocating TLA+ to other teams working on S3. Engineers from those teams wrote specs for two additional critical algorithms, and one new feature. F.Z. helped teach the engineers how to write their first specs. We find it encouraging that TLA+ can be successfully taught by engineers who are still quite new to it themselves; this is important for quickly scaling adoption to an organization as large as Amazon.

Author B.M. was one of the engineers mentioned. B.M.'s first spec was for an algorithm that had a known subtle bug. The bug had passed unnoticed through multiple design reviews and code reviews, and had only surfaced after months of testing. B.M. spent two weeks learning TLA+ and writing the spec. Using this spec, the TLC model checker found the bug in a few seconds. The team had already designed and reviewed a fix for this bug, so B.M. changed the spec to include the proposed fix. The model checker found that the problem still occurred, but via a different execution trace. A stronger fix was proposed, and the model checker verified the second fix. Later, B.M. wrote another spec for a different algorithm. That spec did not uncover any bugs, but did uncover several important ambiguities in the documentation for the algorithm, which the spec helped to resolve.

Somewhat independently, after seeing internal presentations about TLA+, authors M.B and M.D. taught themselves PlusCal and TLA+ and started using them on their respective projects without further persuasion or assistance. M.B. used PlusCal to find three bugs, and also wrote a public blog about his personal experiments with TLA+ outside of Amazon [16]. M.D. used PlusCal to check a lock-free concurrent algorithm. He then used TLA+ to find a critical bug in one of our most important new algorithms. M.D. also came up with a fix for the bug, and verified the fix. C.N. independently wrote a spec for the same algorithm; C.N.'s spec was quite different in style to that written by M.D., but both found the same bug in the algorithm. This suggests that the benefits of using TLA+ are quite robust to variations between engineers. Both specs were later used to verify that a crucial optimization to the algorithm did not introduce any bugs.

We continue to use TLA+ in our work. We have adopted the practice of first writing a conventional prose design document, then incrementally refining parts of it into PlusCal or TLA+. Often this gives important insights without ever going as far as a 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, and these specifications helped him understand the design well enough to propose a major optimization that radically reduced write latency in that system. Most recently we discovered that TLA+ is an excellent tool for data modeling, e.g. designing the schema for a relational or 'No SQL' database. We used TLA+ to design a non-trivial schema, with semantic invariants over the data that were much richer than standard multiplicity constraints and foreign key constraints. We then added high-level specifications of some of the main operations on the data, which helped us to correct and refine the schema. This suggests that a data model might best be viewed as just another level of abstraction of the entire system. This work also suggests that TLA+ may help designers improve the scalability of a system; in order to remove scalability bottlenecks, designers often need to break atomic transactions into finer-grain operations chained together via asynchronous workflows, and TLA+ can help explore the consequences of such changes with respect to isolation and consistency.

The 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 that we don't. Despite this, formal methods help in multiple ways:

- Formal methods help engineers to 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 will probably be deceived into believing that the code is 'correct' because it appears to correctly implement the (broken) design. Engineers are unlikely to realize that the design is incorrect while they are focusing on coding.
- Formal methods help engineers gain a better understanding of the design. Improved understanding can only increase the chances that that the engineers will get the code right.
- Formal methods can help engineers write better "self-diagnosing code", in the form of assertions. Evidence [17] and our own experience suggest that 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 why the system works; the system won't 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 that is strong enough to ensure that no safety properties are violated. Formal methods help engineers to find strong invariants, so formal methods can help to improve assertions, which help improve the quality of code.

While we would like to verify that the 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 we are building. We do routinely use conventional static analysis tools, but these are largely limited to finding 'local' issues in the code, and cannot 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 [18]. This approach seems promising. However the cited paper is about hardware designs, and we have not yet tried to apply that method to software.

Alternatives to TLA+

There are many formal specification methods. We evaluated several, and published our findings in a paper [19]. That paper lists the requirements that we think are important for a formal method to be successful in our industry segment. When we found that TLA+ met those requirements, we stopped evaluating methods, as our goal was always practical engineering rather than an exhaustive survey.

Related Work

We have found relatively little published literature on using high-level formal specification to verify the design of complex distributed systems in industry. The Farsite project [20] is complex, but is somewhat different to the types of system we describe, and apparently was not launched commercially. Abrial [21] lists applications to commercial safety-critical control systems, but which seem less complex, and are quite far from our problem domain. Lu et al. [22] describe post-facto verification of a well-known algorithm for a fault-tolerant distributed hash table, and Zave [7] describes another such algorithm, but we don't know if these algorithms have been used in commercial products.

Conclusion

At AWS, formal methods have been a big success. They have helped us prevent subtle, serious bugs from reaching production, bugs that we would not have found via any other technique. They have helped us to make aggressive optimizations to complex algorithms without sacrificing quality. So far, seven teams have used TLA+, and all have found high value in doing so. At the time of writing, more teams are starting to use TLA+. We believe that use of TLA+ will accelerate both time-to-market and quality of these projects. Executive management is now proactively encouraging teams to write TLA+ specs for new features and other significant design changes. In annual planning, managers are now allocating engineering time to use TLA+.

While our results are very encouraging, some important caveats remain. Formal methods deal with models of systems, not the systems themselves, so the adage applies; "All models are wrong, some are useful." The designer must ensure that the model captures the significant aspects of the real system. Achieving this is a difficult 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.

References

[1] Barr, J. Amazon S3-The First Trillion Objects. *Amazon Web Services Blog*. June 2012; http://aws.typepad.com/aws/2012/06/amazon-s3-the-first-trillion-objects.html

[2] Barr, J. Amazon S3-Two Trillion Objects, 1.1 Million Requests per Second. *Amazon Web Services Blog*. March 2013; http://aws.typepad.com/aws/2013/04/amazon-s3-two-trillion-objects-11-million-requests-second.html

[3] Holloway, C. Michael. Why You Should Read Accident Reports. Presented at Software and Complex Electronic Hardware Standardization Conference, July 2005; http://klabs.org/richcontent/conferences/faa_nasa_2005/presentations/cmh-why-read-accident-reports.pdf

[4] Lamport, L. The TLA Home Page; http://research.microsoft.com/en-us/um/people/lamport/tla/tla.html

[5] Joshi, R., Lamport, L., et al. Checking Cache-Coherence Protocols With TLA+; http://research.microsoft.com/pubs/65162/fmsd.pdf

[6] Patterson, D., Fox, A., et al. The Berkeley/Stanford Recovery Oriented Computing Project; http://roc.cs.berkeley.edu/

[7] Zave, P. Using lightweight modeling to understand Chord; In *ACM SIGCOMM Computer Communication Review* volume 42 number 2, April 2012; http://www2.research.att.com/~pamela/chord.html

[8] Newcombe, C. Debugging Designs. Presented at the 14th International Workshop on High Performance Transaction Systems, Monterey 2011; http://hpts.ws/papers/2011/sessions_2011/Debugging.pdf and associated specifications; http://hpts.ws/papers/2011/sessions_2011/amazonbundle.tar.gz

[9] Lamport, L. Fast Paxos. Distributed Computing, Volume 19 Issue 2, October 2006, 79-103; http://research.microsoft.com/pubs/64624/tr-2005-112.pdf

[10] Lamport, L., Sharma, M., Tuttle, M., Yu, Y. The Wildfire Challenge Problem, Compaq 2001; http://research.microsoft.com/en-us/um/people/lamport/pubs/wildfire-challenge.pdf

[11] Batson, B., Lamport, L. High-Level Specifications: Lessons from Industry, March 2003; http://research.microsoft.com/en-us/um/people/lamport/pubs/high-level.pdf

[12] Lamport, L. The Wildfire Challenge Problem ; http://research.microsoft.com/en-us/um/people/lamport/tla/wildfire-challenge.html

[13] Lamport L., Merz, S. Specifying and Verifying Fault-Tolerant Systems. In *Proceedings of the Third International Symposium on Formal Techniques in Real Time and Fault Tolerant Systems*, September 1994; http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-ftrtft94.pdf

[14] Supported Operations in DynamoDB: Strongly Consistent Reads; http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/APISummary.html

[15] Lamport, L. Checking a Multithreaded Algorithm with +CAL; In Proceedings of Distributed Computing: 20th International Symposium, DISC 2006; http://research.microsoft.com/en-us/um/people/lamport/pubs/dcas.pdf

[16] Brooker, M. Exploring TLA+ With Two-Phase Commit; http://brooker.co.za/blog/2013/01/20/two-phase.html

[17] Kudrjavets, G., Nagappan, N., Ball, T. Assessing the Relationship between Software Assertions and Code Quality: An Empirical Investigation; http://research.microsoft.com/pubs/70290/tr-2006-54.pdf
[18] Tasiran, S., Yu, Y., Batson, B., Kreider, S. Using formal specifications to monitor and guide simulation. In *Proceedings of the 3rd IEEE International Workshop on Microprocessor Test and Verification,* 2002; http://research.microsoft.com/apps/pubs/default.aspx?id=65169
[19] Newcombe, C., Why Amazon Chose TLA+, in *Abstract State Machines, Alloy, B, TLA, VDM, and Z Lecture Notes in Computer Science Volume 8477, 2014, pp 25-39*; http://link.springer.com/chapter/10.1007%2F978-3-662-43652-3_3
[20] Bolosky, W., Douceur, J., Howell, J. The Farsite project: a retrospective; http://research.microsoft.com/apps/pubs/default.aspx?id=74211
[21] Abrial, J. Formal Methods in Industry: Achievements, Problems, Future; http://www.irisa.fr/lande/lande/icse-proceedings/icse/p761.pdf
[22] Lu, T., Merz, S., Weidenbach, C. Towards Verification of the Pastry Protocol Using TLA+; http://www.loria.fr/~merz/papers/forte2011pastry.html