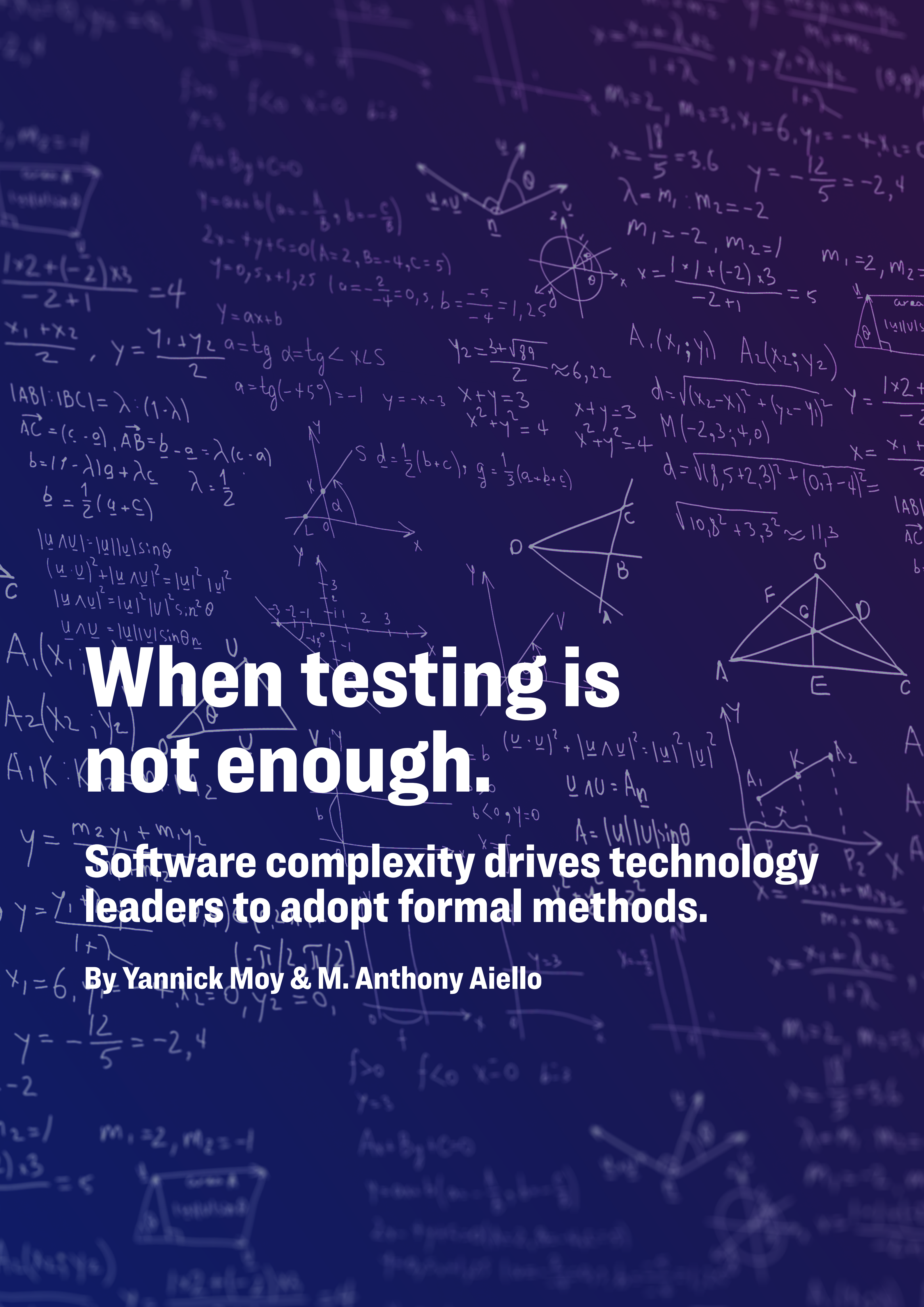


When testing is not enough.

Software complexity drives technology leaders to adopt formal methods.

By Yannick Moy & M. Anthony Aiello

Executive Summary

The size and complexity of software in embedded systems are growing at an astonishing rate. From aircraft and automobiles to medical devices, home appliances, and our homes themselves, products that were once hardware-only are now cyber-physical: they rely on software for much of their functionality. And we rely on that software for the dependability of those systems, especially their safety and security.

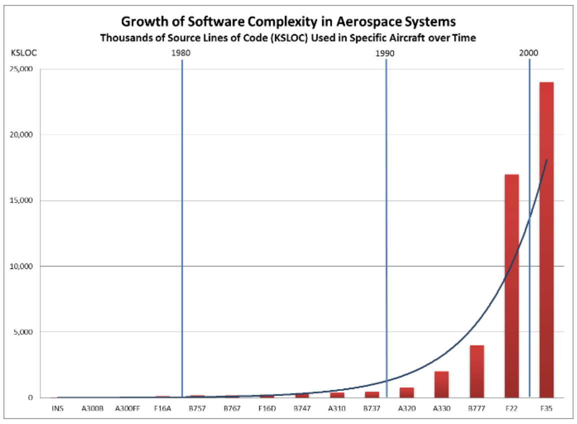
Verifying correct software behavior is an increasingly challenging problem, given this growth in software size and complexity. Conventional (or traditional) testing methods are insufficient; there is simply too much ground to cover. This is especially true for software that is safety- or security-critical: software for which undiscovered defects (bugs) can result in catastrophic failure, loss of life, vulnerability to theft, and/or severe financial damage. This software verification challenge is a problem we can no longer ignore.

Today, leading companies are finding relief from this problem through the use of *formal methods*. Long viewed as impractical or too expensive for commercial software development, formal methods have come of age thanks to advances in computing power and new tools that automate and simplify their application.

In this white paper, we examine the problems being created by software's increasing complexity. We look at why traditional verification methods are no longer adequate for highly dependable applications (and haven't been for some time). We explore how formal methods can help solve the problem of verifying that critical software is reliable, safe and secure, without increasing life cycle costs. We look at what formal methods are, what to look for when choosing them, why now is an excellent time to begin applying them, and who is benefiting from formal methods today.

The impact of complexity on software reliability

The complexity of software in embedded systems has grown at an exponential rate for years, as evidenced by the following graphs (Figures 1 and 2).



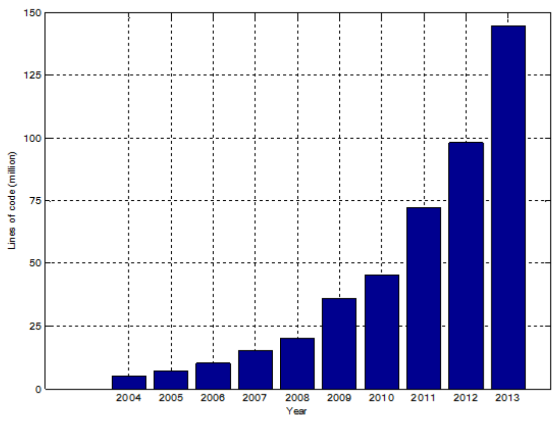


Figure 1: Growth software complexity in aerospace systems over time[i] Figure 2: Growth software complexity in automobiles over time[ii]

In software, size is an important measure of complexity. Larger software has more inputs, more states, and more variables. In other words: larger software has more things to test and more opportunities for things to go wrong. Over the last few decades, increasing software complexity has meant an ever-increasing challenge in the verification of critical systems.

Critical systems include those requiring a high degree of dependability, including safety and security, and are typical of numerous industries, such as aerospace and defense, automotive, medical, energy generation and distribution, hazardous material management, and cybersecurity. The required maximum probability of failure in the most critical elements of these systems may be on the order of 10^{-7} to under 10^{-9} failures per hour.

Cost and schedule impacts

Software complexity impacts the cost and schedule of system development. The National Research Council (NRC) found that large software projects show very high rates of delay, cost overrun, and cancellation.[iii] Both dependable and typical commercial software programs suffer similar low success rates. The NRC found this unsurprising, "because dependable applications are usually developed using methods that do not differ fundamentally from those used commercially. The developers of dependable systems carry out far more reviews, more documentation, and far more testing, but the underlying methods are the same."

Accidents and disruptions

As software has grown more complex, these underlying software-development methods have frequently failed to prevent disasters, even in systems developed to standards and extensively tested.

- On January 15, 1990, an undetected defect in a new version of switching software brought down ATT's global long-distance network for more than nine hours. ATT lost more than $60 million in unconnected calls and suffered a severe blow to its reputation.[iv]
- In January 2017, the US FDA and Dept. of Homeland Security issued warnings against at least 465,000 St. Jude's Medical RF-enabled cardiac devices. Software vulnerabilities in the devices could allow hackers to remotely access a patient's implanted device and disable therapeutic care, drain the battery, or even administer painful electric shocks. Short-selling firm Muddy Waters had revealed these flaws in August 2016, based on a report by the security firm MedSec, alleging negligence in St. Jude Medical's software development practices.[v]
- The WannaCry ransomware attack of May 2017 encrypted the data of more than 200,000 computers across 150 countries. WannaCry made use of EternalBlue; a sophisticated cyberattack exploit stolen from the U.S. National Security Agency (NSA). EternalBlue exploits a vulnerability in Microsoft's implementation of the Server Message Block (SMB) protocol in Windows and Windows Server. One month later, the NotPetya malware attack used EternalBlue to destroy data on computers across Europe, the U.S., and elsewhere. According to Kaspersky Labs, damage estimates from WannaCry range from $4 billion to $8 billion, while losses from NotPetya may total over $10 billion.[vi]

These are just a few examples. As software complexity continues to grow, so do the chances that current software verification methods will fail to find such faults. As the NRC concluded, "the evidence is clear: these methods cannot dependably deliver today's complex applications, let alone tomorrow's even more complex requirements."

Testing can find bugs… but cannot prove no bugs remain

Software testing is designed to reveal software defects. Software inputs are exercised in various sequences and combinations, and software outputs are monitored and compared with expected behavior. When the observed behavior differs from expected behavior, the source of the discrepancy is identified, and the software is corrected.

As software complexity has increased exponentially, there has been a corresponding increase in the number of test cases and the time and cost required to execute them. The question then becomes, when can we stop testing?

Testing can only establish the presence of defects, not their absence — unless all possible combinations of inputs and internal states are covered during testing. Thus, verification and validation by testing are—for all practical purposes—impossible. This has significant implications for critical software and has been known for quite a long time.

Exhaustive testing is practically impossible

To prove dependability, test cases must cover all possible combinations of inputs and internal states. However, even for software of moderate size, exhaustive testing is practically impossible. A simple piece of software with three 32-bit inputs and one 32-bit internal state could easily require 3.4×10^{38} test cases to test exhaustively. Executing one million test cases per second, the required time to complete exhaustive testing would be over 1×10^{25}—*ten trillion trillion*—years.

Life testing doesn't work on software

To address the impossibility of exhaustive testing, alternative testing strategies have been proposed. One traditional method of verifying reliability is *life testing*. Life testing involves testing to failure on a statistically significant number of test specimens.

Life testing works well on hardware. Physical failures occur when hardware breaks down due to a manufacturing or material defect, is eroded by adverse environmental conditions, or wears out from the repeated stress of normal operation.

Software, however, doesn't fail in the way hardware does. Software defects are introduced during development but are only revealed when the right combination of inputs and states is exercised. Otherwise, these defects lie dormant.

In 1991, Ricky Butler and George Finelli, from the Formal Methods Group at NASA Langley Research Center, proved that life testing is impractical for verifying high-reliability software. They showed that to demonstrate a fault probability of $<10^{-9}$, the duration of the required testing would range from hundreds of years (for prohibitively high numbers of test specimens) to *hundreds of thousands or even millions of years* for normal numbers of test specimens (see sidebar at right).[vii]

Testing critical software takes centuries!

In their paper, *The Infeasibility of Experimental Quantification of Life-Critical Software Reliability,* Ricky Butler and George Finelli published the following table of expected *minimum* test durations for life testing of software requiring a maximum fault probability 10^{-9} (DO-178 Level A, for example).

No. of Replicates	Expected Test Duration (D_t)
1	10^{10} hours = 1,141,550 years
10	10^{9} hours = 114,155 years
100	10^{8} hours = 11,415 years
10,000	10^{6} hours = 114 years

Table 1: Expected Life Testing Duration for r=1

The factor r is the number of observed failures after which the test is stopped. Butler and Finelli note that, “a value of r equal to 1 produces the shortest test time, but at the price of extremely high α and β errors (the probability of rejecting a good system and the probability of accepting a bad system, respectively). To get satisfactory statistical significance, larger values or r are needed and even more testing.”

Thus, we can see that even if one had 10,000 specimens (code instances and test environments) available simultaneously, it would still take over one hundred years of testing to assure a fault probability of $< 10^{-9}$.

With a more realistic number of test specimens, even a fault probability of $< 10^{-7}$ (DO-178 Level B) would take hundreds if not thousands of years to achieve.

Reliability growth models won’t work either

Another method, *reliability growth models*, uses a repetitive process of testing and repairing a program to predict the reliability of the latest repaired iteration. This method, too, is infeasible for critical software.

Reliability growth models require correction of software defects to improve the reliability prediction. At each subsequent iteration, however, the remaining defects become much harder to find and thus take much longer to remove. Butler and Finelli showed that to predict a fault probability below 10^{-9} the duration of the final test iteration—the time needed to remove *just the last bug*—would be on the order of tens of thousands to hundreds of thousands of years.

Why testing won’t prove dependability

That leaves us with the most common method: normal software testing. As previously described, software testing is valuable for *finding* defects. What testing doesn’t do, however, is *prove the absence of defects*, which is what is required to verify dependability.

And yet, testing is an essential part of standards in critical software domains such as avionics. Given the impossibility of exhaustive testing, these standards have defined coverage criteria to decide when enough testing has been performed. Those criteria are no guarantee of dependability, however.

For example, it would be a misunderstanding to attribute the safety of today’s avionics software to testing. It’s a common saying in that domain that verifying avionics software has become more a matter of achieving sufficient confidence in the developers and the development processes than of testing the software itself. Software quality is obtained mostly by a pervasive culture of safety encoded in the processes and assimilated by the people. Testing in that environment serves as one of the checks that the resulting software meets its requirements.

In general, as software complexity has risen, testing to coverage criteria has become more and more a matter of “test until you run out of time or budget.”

Why formal methods are needed to verify critical software

When civil engineers design a bridge, they don’t just build and test the bridge to discover if it will collapse under its own weight. Instead, they first analyze a model of the bridge, performing structural load calculations to prove the bridge won't collapse.

Likewise, when mechanical engineers design a new machine, they model and analyze the stresses the machine and its parts will have to withstand. Then, they specify parts that will handle those stresses.

In the software world, formal methods enable software developers to perform analyses that are analogous to those made by those civil and mechanical engineers. Formal methods are techniques that reason about mathematical models of software.

Using formal methods-based tools, software engineers can then rapidly gain assurance that their software behaves correctly by posing questions about software behavior, such as “can a race condition ever occur?” or “can a memory overflow ever occur?” or “does a certain critical variable ever exceed a safety threshold?” In short, formal-methods tools allow engineers to obtain answers to critical questions about their software.

Broadly, formal-methods analyses fall into three categories: (1) abstract interpretation, which models software semantics imprecisely but enables rapid identification of potential defects; (2) model checking, which attempts to verify that a property holds over bounded or exhaustive search of a model of software behavior; and (3) theorem proving, which attempts to apply deductive reasoning to verify that a property holds over all permitted inputs and internal states of the software. These analyses fall on a spectrum from more automatic but less precise (abstract interpretation) to less automatic but more precise (theorem proving) and are highly complementary.

Why formal methods are a better verification solution for critical software

Unlike testing, which samples software behavior in order to reveal defects, formal methods analyze the behavior of the software and establish the presence *or absence* of defects. Software engineers can identify specific conditions that must or must not occur and use formal methods to prove that those conditions always or never occur. Thus, formal methods can be used to guarantee the absence of buffer overruns, integer overflow conditions, and other defects that lead to unpredictable behavior.

The guarantees provided by formal methods deal with all possible inputs and all possible internal states, rather than only those inputs and states covered by executed test cases. Engineers can be confident a latent defect will not be revealed by an unforeseen, unusual sequence of inputs — as was the case in the examples we considered earlier.

Formal methods aren’t new

Formal methods are the foundation upon which computer science was based: even before practical, general-purpose computers appeared, models of computation were developed and analyzed. The introduction of high-order programming languages in the 1950s and

1960s provided a concrete representation for these concepts, and, in the late 1960s, C.A.R. "Tony" Hoare developed what became known as Hoare logic as a means to express a computation formally. It was nearly three decades; however, before advances in computing methods made the widespread application of formal methods truly practical.

Since the late 1990s, thanks in large part to the Pentium FDIV bug, formal methods have been used extensively in the electronics industry to verify integrated circuit designs.[viii, ix]

More recently, formal methods have entered the mainstream of software verification. This uptake of formal methods has been made possible by several primary factors:

- Advances in the state of the art in Satisfiability Modulo Theory (SMT) solvers, which provide the heavy lifting for most modern formal-methods analyses;
- Ever greater compute power in personal computers and in the cloud;
- The design of programming languages (in fact subsets of general-purpose languages) that are simple enough to be formally analyzable but expressive enough for coding real-world software applications; and
- Improvements in the overall usability of formal methods-based tools.

These innovations have enabled software companies to quickly apply and benefit from formal methods—even without prior formal-methods experience. Today, companies large and small are using formal methods on a day-to-day basis in software development.

Who will benefit from using formal methods?

Any organization that develops critical software to meet the highest dependability

Choosing a formal-methods solution

Different formal-methods solutions are geared toward different software development environments and organizational needs. Consider the following when selecting a formal method. We illustrate each topic using SPARK, a formally analyzable subset of Ada.

1. Expressivity versus Automation

Expressivity refers to how easily engineers can describe desired functionality in a given formal language. Automation refers to how easily formal-methods tools can complete analysis without human guidance. In general, greater expressivity results in lower automation.

SPARK provides a good balance between expressivity and automation. The language inherits Ada's strong typing and support for safety-critical development. SPARK excludes only those very few Ada features that limit automation or increase the likelihood of defects. The SPARK tools can complete most analyses with limited human guidance; additional guidance, when necessary, is provided using the SPARK language and is integrated into the software.

2. Soundness

Soundness means that analysis results are accurate and trustworthy: if a methodology or tool is claiming to verify a given program property (for example, that an index into an array is within the array bounds), then it must detect any and all violations of that property. This is a critical characteristic of formal methods. Many tools attempt to spot bugs automatically, but such tools are generally unsound; they *cannot prove the absence* of specific classes of bugs.

SPARK is sound. When SPARK proves absence of run-time exceptions, the proof is an ironclad guarantee that no such exceptions will occur in the part of the software written in SPARK.

3. Integration

Formal methods typically offer much smaller libraries of common programming tasks, as compared to traditional programming languages. Developers will, therefore, need to be able to integrate the part of the software written in the formal language with the part of the software that depends upon existing libraries.

SPARK makes this easy, by integrating with full Ada and with C and C++ as well. SPARK can also interface with other popular languages, such as Java and Python.

4. Incrementality

Formal methods are rarely applied in all-new software development. Instead, formal methods can be incrementally applied to existing software and within standards — i.e., software whose failure is unacceptable — will benefit from using formal methods in verification.

Safety-critical modules in avionics, medical devices, automotive vehicle control, nuclear power control and monitoring, and other hazmat management applications are prime candidates for formal verification. Critical infrastructure applications like energy distribution and telecom switching, connectivity, and other secure-data applications, and critical computing components like OS kernels will also benefit by having their reliability proven by formal methods.

Who is using formal methods and why

Today, formal methods are being used for software verification across many industries and applications. Practitioners include:

- Aerospace giants, including Lockheed Martin, Airbus, Rockwell Collins, Thales, and NASA
- Major automakers, including Toyota
- Computer industry leaders, including Microsoft, NVIDIA and Amazon
- Multinational conglomerates, including GE

These companies and many others have turned to formal verification to:

- Assure the dependability of critical software modules
- Eliminate vulnerabilities to malware attacks
- Reduce software verification schedules
- Eliminate defects early before they become more costly to remove

existing software-development efforts. SPARK enables incremental adoption within existing development efforts. SPARK brings immediate benefits, because the language omits a few Ada features that can challenge developer understanding and possibly increase the likelihood of defects.

Another way SPARK supports gradual adoption is by providing several levels of analysis. Each level requires more investment from developers but results in stronger guarantees.

5. High-Quality Tooling and Developer Support

Many formal methods are not professionally developed or supported, which may challenge their adoption in industrial software-development efforts.

SPARK provides high-quality tooling and is professionally supported. SPARK is integrated into AdaCore's professional IDE and provides detailed information about failed proofs—including counterexamples—which greatly assists developers in fixing errors so that the proofs can be completed.

6. Active User Community

Active user communities are as important for formal methods as they are for typical programming languages. User communities contribute to library development, offer a broader support network, and offer diverse opportunities for learning how to use formal methods.

SPARK has a diverse community that is active on several platforms, including GitHub, Stack Overflow, Reddit, LinkedIn, Twitter, Facebook, and AdaCore.com.

Case study: Amazon Web Services s2n

In June 2015, Amazon Web Services (AWS) introduced a new Open Source implementation of the SSL/TLS network encryption protocols. They've called this implementation Amazon s2n ("signal to noise"). Because of s2n's security-critical role, AWS decided to use it as a proving ground for new automated reasoning, testing, and assurance techniques that could be built upon for broader adoption.[x]

AWS hired Galois, a research and development firm that specializes in applied formal methods, to simplify this process and make it developer-friendly. Galois developed a tool chain that allows AWS to formally verify important aspects of s2n, which they integrated into the s2n build environment. Now, anyone with the prerequisites installed can run the same proofs on their own s2n code. Plus, they designed the reports generated from these automated proofs so that they're easy to understand—even by people with no formal methods training.[xi]

More broadly, AWS is proving that more and more of their code is correct using formal methods. For example, they have formally verified s2n's implementation of HMAC, an important algorithm used extensively within the TLS/SSL protocols and elsewhere. In addition, they are using automated formal methods tools to continuously enhance AWS security and to provide functionality to customers through the AWS services Con_g, Inspector, GuardDuty, Macie, Trusted Advisor, and the storage service S3.[xii]

"Proof is an accelerator for adoption," says Byron Cook, Director of Automated Reasoning at AWS. "People are moving orders of magnitude more workload (to AWS) because they're (saying) 'in my own data center I don't have proofs, but there (at AWS) they have proofs.' "[xiii]

Case study: seL4 open-source separation kernel

A separation kernel is a type of security kernel used to simulate a distributed environment. As such, the environment must appear as though each regime is a separate, isolated machine. "One of the properties we must prove of a separation kernel, therefore, is that there are no channels for information flow between regimes other than those explicitly provided," said John Rushby, who conceived the device.[xiv]

Without a formal guarantee of the absence of run-time errors in the separation kernel, however, you have no guarantee of security, even if other critical parts of the system have been formally verified.

Fortunately, separation kernels proven free of runtime errors are now becoming commercially available.

The seL4 is a third-generation microkernel developed by the NICTA group as a basis for highly secure and reliable systems. The world's first operating-system kernel with an end-to-end proof of implementation correctness and security enforcement—formal verification of its functional correctness was completed in 2009—seL4 became available as open source in July 2014.[xv]

The seL4 is no longer unique in that regard, however. The Muen kernel, a small separation kernel written primarily in SPARK, has also been proven error-free using formal methods.[xvi] These kernels are good examples of how formal verification can help establish a guaranteed error-free base upon which to build better, more reliable systems.

Case study: Tokeneer ID Station

To demonstrate that developing highly secure systems to the level of rigor required by the higher assurance levels of the Common Criteria is possible and practical, the United States National Security Agency (NSA) asked the UK consultancy firm Altran to undertake a research project: develop part of an existing secure system (the Tokeneer System) in accordance with Altran's formal methods-based *Correctness by Construction* development process.

The resulting Tokeneer ID Station[xvii] project demonstrated that formal methods can produce a high quality, low defect system, conformant to the Common Criteria's

Evaluation Assurance Level (EAL) 5 and above, in a cost-effective manner. The project effort was carefully monitored, and the resulting productivity was 38 lines of code per day overall and 203 lines of code per day during the coding phase while achieving the ultra-high reliability that was required.

Conclusions

Software testing has never been adequate for guaranteeing the dependability of critical software, as was proven mathematically in the 1990s. Moreover, as software has continued to grow exponentially, the inadequacy of software testing has manifested itself with increasing frequency in the form of costly system failures caused by latent software defects.

Now more than ever, developers of critical software need to join the leaders in their industry and embrace formal methods. Advances in technology and improvements to usability have made the application of formal methods feasible and practical for software verification. Current formal methods-based tools have made it easier than ever to get started.

Using formal methods, leading companies are now obtaining guarantees of correct functionality and dependability, while saving total life cycle cost. Far from a research-focused, esoteric application, formal methods have become a key differentiator in high-profile applications — a point underscored by Byron Cook, Director of Automated Reasoning at AWS. “AWS customers love this work!” says Cook, referring to Amazon’s use of formal verification. “Soundness is key. Customers don’t want to hear about how many more hours we’ve spent on testing. But when you talk about proof, the conversation changes.”

Additional Information

Besides software verification, formal methods are also being applied in related disciplines. Follow the links below to find more information on some of those domains, including:

System requirements

- www.ccs.neu.edu/home/pete/pub/re-2018.pdf
- http://loonwerks.com/tools/spear.html
- http://loonwerks.com/publications/wagner2017nfm_spear.html

Software architecture

- http://loonwerks.com/tools/agree.html

Model-based design

- https://www.mathworks.com/products/sldesignverifier.html
- https://ieeexplore.ieee.org/document/7423151

Software specification

- http://pvs.csl.sri.com/
- https://en.wikipedia.org/wiki/Z_notation

Notable formal methods centers include:

NASA LaRC: https://shemesh.larc.nasa.gov/fm/

SRI CSL: http://www.csl.sri.com/programs/formalmethods/

Collins Aerospace (http://loonwerks.com)

Galois (https://galois.com/)

About AdaCore

Founded in 1994, AdaCore supplies software development and verification tools for mission-critical, safety-critical and security-critical systems. Four flagship products highlight the company's offerings:

- The GNAT Pro development environment, a complete toolset for designing, implementing, and managing applications that demand high reliability and maintainability. GNAT Pro is available for Ada and also for C and C++.
- The CWE-Compatible CodePeer advanced static analysis tool, an automatic Ada code reviewer and validator that can detect and eliminate errors both during development and retrospectively on existing software. CodePeer can detect a number of the "Top 25 Most Dangerous Software Errors" in the MITRE Corporation's Common Weakness Enumeration (CWE).
- The SPARK Pro verification environment, a toolset providing full formal verification oriented toward high-assurance systems with stringent security requirements.
- The QGen model-based development tool suite for safety-critical control systems, providing a qualifiable and customizable code generator and static verifier for a safe subset of Simulink® and Stateflow® models, and a model-level debugger.

Over the years customers have used AdaCore products to field and maintain a wide range of critical applications in domains such as commercial and military avionics, automotive, railway, space, defense systems, air traffic management/control, medical devices, and financial services. AdaCore has an extensive and growing worldwide customer base; see www.adacore.com/industries/ for further information.

AdaCore products are open source and come with expert online support provided by the developers themselves. The company has North American headquarters in New York and European headquarters in Paris. www.adacore.com/.

For a more in-depth look at how your organization can get started on an incremental approach to formal verification, download our free guide, *Implementation Guidance for the Adoption of SPARK*.

If you are interested in pricing information on SPARK Pro, please fill out our online pricing request form.

References

[i] Motivation for Advancing the SAVI Program, Aerospace Vehicle Systems Institute, Texas A&M University (accessed: August 2019).

[ii] Khan, Z. H. and Khan, A.H., Perspectives in Automotive Embedded Systems: From manual to fully autonomous vehicles, SAME, November 2015.

[iii] Jackson, D., Thomas, M. and Millett, L., editors, Committee on Certifiably Dependable Software Systems, National Research Council, Software for Dependable Systems: Sufficient Evidence?, National Academy of Sciences, 2007.

[iv] Burke, D., All Circuits are Busy Now: The 1990 AT&T Long Distance Network Collapse, California Polytechnic State University, November 1995.

[v] 465,000 Abbott pacemakers vulnerable to hacking, need a firmware fix, CSO, September 2017.

[vi] Snow, J., Top 5 most notorious cyberattacks, Kaspersky, December 2018.

[vii] Butler, R. and Finelli, G., The Infeasibility of Experimental Quantification of Life-Critical Software Reliability, NASA, December 1991.

[viii] Van Eijk, C. A. J., Formal Methods for the Verification of Digital Circuits, Technische Universiteit Eindhoven, September 1997.

[ix] Marques-Silva, J. and Guerra e Silva, L., Solving Satisfiability in Combinational Circuits, IEEE, July 2003.

[x] MacCarthaigh, Colm, Automated Reasoning and Amazon s2n, Amazon Web Services, September 2016.

[xi] Tomb, A., Magill, S., et al, Proving Amazon's s2n correct, Galois, 2016.

[xii] Cook, B., Formal reasoning about the security of Amazon Web Services (paper), FLoC, July 2018.

[xiii] Cook, B., Formal reasoning about the security of Amazon Web Services (presentation), FLoC. July 2018.

[xiv] Rushby, J., The Design and Verification of Secure Systems, Eighth ACM Symposium on Operating System Principles, pp. 12-21, Asilomar, CA, December 1981. (ACM Operating Systems Review, Vol. 15, No. 5)

[xv] Potts, D., et al, Mathematically Verified Software Kernels: Raising the Bar for High Assurance Implementations, General Dynamics, July 2014.

[xvi] Buerki, R. and Rueegsegger, A., Muen - An x86/64 Separation Kernel for High Assurance, University of Applied Sciences Rapperswil, August 2013.

[xvii] Cooper, D. and Barnes, J., *Tokeneer ID Station EAL5 Demonstrator: Summary Report*, August 2008.
https://www.adacore.com/uploads/downloads/Tokeneer_Report.pdf

AdaCore