

Formal Methods in the World (Draft)*

Various authors

June 23, 2008

Abstract

A reasoned argument is used to explain why things work or fail. A formal argument is one that follows a precise structure of axioms and inference rules. By encoding the inference steps, formal arguments can be mechanically checked for correctness using a computer. In many cases, the formal arguments can be produced mechanically as well. Automated reasoning techniques can be used to also generate examples of some mathematical constraints. The axiomatic framework has been a part of the scientific method since Plato. The use of formal, mechanized reasoning has been advocated since Ramon Llull and Leibniz. In computing, the use of formal methods to verify software has been a part of computing from the mid-20th century. Recently, there have been a number of significant advances in both the theory and technology of formal methods. With these advances, we can contemplate a number of far-reaching applications for this technology. In particular, we can apply both the ideas and tools in supporting a rigorous, coherent, skills-driven curriculum for school-level mathematics and computing. Formal methods for verifying hardware, algorithms, and protocols and modeling and designing complex software systems can be inserted into courses in other sub-disciplines of computing. In industry, formal methods can be applied in a variety of ways to capture requirements and business rules, generate test cases and scenarios, explore designs, verifying protocols, and producing assurance cases. In other science and engineering disciplines, formal methods can be used to support system-level modeling and analysis. Over the next thirty years, formal methods will find a much broader audience covering a wide range of disciplines.

Science is what we understand well enough to explain to a computer. Art is everything else we do.

Donald E. Knuth

1 Introduction

Formal methods have made rapid progress in recent years. The tools and ideas are increasingly being used in industry in a variety of ways. Many of the challenging problems of building secure software systems, programming multicore processors, and cyber-physical systems will require formal support for modeling and analysis. There

*This draft roadmap is based on the input of a number of contributors. We welcome your comments. Funded by NSF CISE Grant Nos. 0646174 and 0627284.

are a growing number of applications both in computer science research as well as in other disciplines. Despite their growing strength and importance, formal methods are not an integral part of a mainstream computer science education. Even at the graduate level, few students pursue research careers in formal methods. We outline some of the obstacles to the wider use of formal methods in education and industry, and enumerate the opportunities for both traditional and non-traditional applications.

Information complexity is a major challenge for the twenty-first century. A modern automobile has about 70 to 80 processors handling a wide range of sophisticated functions ranging from engine control and automated braking, to cruise control, climate control, entertainment, and navigation. The World Wide Web contains several billion web pages. Software is often nearly half the development cost of complex systems such as planes, weapon systems, and automobiles. Software unreliability alone cost the United States economy over a hundred billion dollars. The United States Bureau of Labor Statistics (<http://www.bls.gov/oco/ocos042.htm>) estimates a 40% growth in computer science jobs during 2006–2016. At the same time, undergraduate enrollment dropped from 13,900 in 1998 to 7915 in 2007 (<http://www.insidehighered.com/news/2008/03/05/compsci>). Between 1984 and 2004, the share of CS bachelor's degrees awarded to women fell from 37% to 25% (<http://www.cra.org>). There is clearly a lot of work to be done in packaging Computer Science as a dynamic and intellectually rich discipline with growing links to other scientific and engineering disciplines.

Computer science, like mathematics, is the construction and study of pure abstraction. Though computers do exist in physical reality, computing is really about the representation and manipulation of information. Whereas mathematics deals with the laws of abstract entities such as numbers, sets, surfaces, volumes, relations, algebras, and homomorphisms, computing focuses on manipulating information represented using abstractions such as bits, bit-vectors, arrays, tables, stacks, queues, and trees. Mathematics is used to represent and analyze physical reality, and computing can also be used for modeling and simulation. Where mathematics deals with operations with concise descriptions, computer hardware and software systems are willy-nilly complex constructions. These systems are nearly always required to perform precisely and reliably even in the face of unreliable data and failure-prone physical components.

A computer scientist must have a facility for dealing with abstraction in order to formalize requirements, construct effective and expressive data representations, design correct and efficient algorithms that work on all possible inputs, and cast these designs into reliable code. Abstraction is essential for formalizing the environment in which the software operates; framing the requirements; designing and implementing the system architecture and the components; and debugging and maintaining the system over time. A computer scientist is constantly solving problems in order to invent data representations, algorithms, and protocols, designing new programming and description languages, and resolving synchronization conflicts. Over the last sixty years, computer science has been successful in developing a body of abstract knowledge for solving such problems.

Computing systems confront complexity at every level from hardware platforms, protocols, and data representations to cyber-physical systems and sophisticated security threats. Formal methods represents one approach for managing this complexity

through the use of symbolic representations of abstractions drawn from mathematics and computing in the form of logical formulas. These formulas can be manipulated in various ways to construct proofs. Formulas can be viewed as constraints to be solved. Logic can be used to show that one set of constraints entails another. Computing deals with dynamic phenomena so that automata, state machines, and temporal logic are expressive formalisms for capturing computational behavior and properties. Over nearly fifty years of active development, since the pioneering work of John McCarthy, formal methods have introduced such ideas as functional programming, logic programming, denotational semantics, operational semantics, axiomatic semantics, type systems, type theory, invariance, termination, safety, liveness, modal logics, temporal logics, description logics, specification languages, process algebras, abstract interpretation, runtime verification, and a host of other influential conceptual innovations. Nearly a quarter of the Turing awards given during 1966-2007 have been in recognition of work that has a significant formal methods component.

Now after fifty years, the field of formal methods is at a crossroad. On the positive side, the technology is well developed and there are a large number of interesting applications and many significant industrial users. Within Microsoft Research alone, there are about a hundred researchers working on developing and using formal methods. However, on the negative side, it is no longer considered a part of mainstream computing. There are very few university professors working in the area, sparse course offerings at the undergraduate or graduate level, hardly any flagship projects, and only a small number of doctoral graduates in the pipeline. The capacity of the students for dealing with abstraction, both at the undergraduate and graduate levels, appears to have declined over the years, and the computer science curriculum has itself become increasingly math-phobic [TKB01].

The subject matter of formal methods is both exciting and increasingly relevant. With the dramatic increase in computing power over the last decade, we are now in a position to lift computing to the semantic realm of models and inference. This shift enables a number of novel applications in education and industry. As examples, we can now model both digital and analog hardware, mathematical theories, static and dynamic physical systems, biological networks, complex software systems, and even human psychology and social interactions.

Education at all levels can benefit from formal modeling and analysis, and the deeper use of computational metaphors. Currently, computing is taught in terms of low-level information processing tasks whereas the real insights are obtained from the use of computational models and computation. As a simple example, finite automata can serve as a useful metaphor for a number of tasks such as the working of a vending machine, language parsers, or a video game. Interactive games themselves constitute another computational metaphor that is underused in education. Dynamical systems can be used for modeling continuous flows such as moving vehicles, projectiles, fluid levels, speed governors, analog circuits, and pendulums. Hybrid systems combine discrete and continuous state changes and can be used to model thermostats, cruise control systems, and biological and economic systems. As it happens, these computational models can actually be formally analyzed for properties and counterexample scenarios. Computational models, metaphors, and technology can enrich education at all levels by supporting instruction, skills development, collaboration, and communication.

The industrial uptake of formal methods can also be broadened. Currently, a few industries are both producers and consumers of formal tools. For example, Intel and AMD use such tools for checking circuit equivalence, generating test sequences, model checking finite-state controllers, and proving theorems about floating-point arithmetic algorithms. Microsoft has a number of projects that employ software model checking and automated reasoning to analyze sequential code for termination, assertional correctness, and absence of races and deadlocks. Rockwell-Collins is another leading user of automated reasoning in software verification. Many of the electronic design automation companies like Synopsys, Cadence, and Mentor Graphics sell tools for equivalence and model checking. Matlab recently announced a Simulink design verifier for test case generation and property verification. However, these are still relatively modest efforts compared to the systematic use of formal tools across the entire process. We are not suggesting that the technology is ready for such wider use, but that we now have a framework for contemplating more ambitious and non-traditional applications.

In summary, we need bold innovations that exploit the power of modern formal tools and techniques to radically transform education, both in computing and in other disciplines that value computational metaphors. We also have an opportunity to leverage formal methods in the computer-aided modeling and design of complex systems. The rest of this report outlines some concrete avenues for expanding the audience for formal methods.

2 A Brief Survey of Formal Methods

The idea of placing mathematics and science on an axiomatic foundation goes back to Pythagoras, Plato, Aristotle, and Euclid. The use of a formal language supported by mechanized reasoning was first advocated by Ramon Llull in the thirteenth century, and later by Gottfried Leibniz in the seventeenth century. Vannevar Bush's influential essay, *As We May Think* on the use of computing technology for managing large volumes of information, suggested that

Logic can become enormously difficult, and it would undoubtedly be well to produce more assurance in its use. . . . We may some day click off arguments on a machine with the same assurance that we now enter sales on a cash register.

Now, with the dramatic increases in computing power and the development of powerful software tools for formal reasoning, we are significantly closer to realizing the goals of these visionary thinkers.

Consider the operation that computes the absolute value $|x|$ of an integer x to return $-x$ if $x < 0$, and x , otherwise. We would like to know of this function that the absolute value is always non-negative, and that neither x and $-x$ is larger than $|x|$, but $|x|$ is not larger than both x and $-x$. Checking these requirements involves discharging proof obligations such as $x < 0$ implies $-x \geq 0$, $x \not< 0$ implies $x \geq 0$, $x < 0$ implies $x \geq -x$ and $-x \geq -x$, and $x \not< 0$ implies $x \geq x$ and $-x \geq x$. Such theorems can be proved from the laws of arithmetic which in turn can be proved from the axioms. Such

proof obligations can also be discharged automatically using an arithmetic decision procedure.

As a more interesting example, consider a bag containing a n black and white balls, with an unlimited supply of black balls and white balls outside the bag. As long as there are at least two balls left in the bag, in each step, we pick two balls out of the bag and insert one ball into the box. If the two balls that are picked have the same color, we insert a black ball, and otherwise, we insert a white ball. Can we be sure that this procedure will eventually terminate? What can we say about the color of the ball remaining in the bag with respect to the initial contents of the bag? This problem illustrates that two key issues in the verification of sequential code, namely, termination and invariance.

As a third example, consider the problem of simply counting votes in order to determine if some candidate has a majority of the votes polled. One simple algorithm due to Boyer and Moore [BM91] works by eliminating pairs of votes for distinct candidates, until the votes that remain are for a single candidate. The remaining candidate must be the winner, if anyone has a majority. It is possible that no candidate has polled more than half the votes. Why does this algorithm work? The explanation for this algorithm lies in its proof. Roughly, if a candidate V has a majority, then no matter how V 's votes are paired and eliminated with those of other candidates, there will be votes for V that are left unpaired. This intuition is formalized in both the algorithm and the proof, but in both cases, the details of the formalization are nontrivial. Such formalized proofs are useful in discovering, designing, and implementing correct algorithms as well as in demonstrating their correctness in a *post hoc* manner.

The need for formalization is especially acute in protocols that involve concurrency and cryptography. Without the rigor that is afforded by formalization is extremely difficult to achieve a serious degree of certainty that these protocols achieve their properties without deadlock, livelock, or race conditions.

Formal semantics explicates the mathematical interpretation of a piece of program text or a property, and postulates sound techniques for reasoning about programs and their properties. It is a prerequisite for the successful application of formal methods. Compositionality is a key challenge in formal semantics since we would like to develop properties of programs in terms of those of their constituent subprograms. Recent advances in program semantics have made it possible to formalize the behavior of programs with dynamic storage, object-oriented programs, and concurrent programs.

Automation plays a significant role in the successful application of formal methods. For verifying the behavior of finite-state programs and hardware, model checking can be used to annotate the states with their behavioral properties with a bounded amount of effort. For example, for a given property, this can be done in linear time in the size of the state space of the system. Of course, this state space can be enormous since even a system with n bits of state has a 2^n possible states. The *state explosion* problem in model checking has been tackled in various ways through the use of symbolic representations, abstractions, and partial order reductions. Model checking has also been extended to infinite-state systems such as parametric systems, real-time systems, and linear hybrid systems.

Boolean satisfiability has also been used to represent and reason about finite-state systems through bounded model checking (BMC). In recent years, there have been

significant algorithmic improvements in such satisfiability procedures. These algorithmic advances have also been exploited by solvers for constraints drawn from a mix of theories such as those for arithmetic, arrays, bit vectors, and recursive datatypes. Satisfiability procedures can be used to check that a large formula is unsatisfiable, or to generate counterexamples, test cases, or schedules, when the formula is indeed satisfiable.

3 Challenges for Formal Methods

There is not very much debate these days that formal methods should be more widely used in industry. Many of the problems of complexity that arise in modern software engineering, such as those involving multicore processors, multiple virtual machines, heightened security requirements, and globally distributed systems, are not handled effectively by existing methods which are heavy on testing. We can also make a strong argument that education in computer science could benefit from the introduction of formal tools and techniques. However, there are many challenges that must be addressed before we can make a convincing case for the benefits of formal methods.

1. The programming languages used in large-scale software development have complex formal semantics. Some of this complexity is needless and unhelpful, but formal tools will have to cope with the complexity of other features such as information flow, interrupts, threads, and concurrency.
2. Even with powerful automation, the successful application of formal methods requires a great deal of human skill and ingenuity. Potential users might be unwilling to invest a great deal of time and money on a technology without a clear idea of the benefits.
3. The overhead of applying formal methods in a software development project can be quite large. We do not know the extent to which the application of formal methods can reduce testing costs.
4. Formal methods tools are not sufficiently user-friendly for wider use. In particular, there is not a lot of useful feedback when a verification effort fails. Also, maintaining a formal development is a daunting task comparable to the maintenance of software.
5. There are many powerful formal tools, but the difficult problems often require some combination of these tools. Interoperability is a challenge for these tools since there are subtle semantic differences that must be bridged.

Some of these challenges are being addressed in ongoing work. Quite a bit of the technology can be woven into functionality whose working is transparent to the user [TRS03]. The overhead of applying formal methods will drop as the tools become more powerful and more integrated. Moore's law is already providing a powerful acceleration in the power of formal tools. As the technology stabilizes, the development of expressive user interfaces will become a priority.

4 Opportunities for Formal Methods in Education

Formal methods must be fully integrated into the education of a computer scientist. Subjects such as Theory of Computation, Algorithms, Databases, Compilers, Operating Systems, Computer Architecture, Programming Languages, Artificial Intelligence, and Numerical Analysis can be taught more effectively through the use of formal modelling and analysis methods and tools. Currently, these courses are lacking in formality and computer science curricula are increasingly math-phobic. This means that computer science graduates are not significantly better qualified for careers in computing than those with degrees in other disciplines. There are some preliminary efforts in this direction. The use of the Alloy tool for teaching formal software engineering is showing some success. The Coq proof tool is being used in the University of Pennsylvania to teach programming language metatheory. Wing's detailed proposal [Win00] can serve as a template for incorporating formal methods and verification tools within other computer science courses. The TeachScheme! project is currently experimenting with the ACL2 theorem prover to teach programming to college freshmen [EVF07].

Other disciplines can also benefit from the use of formal methods. Thanks to the success of hardware verification, electrical engineering students are now increasingly exposed to the use of Boolean reasoning and temporal logics. Many engineering courses are now taught using the MATLAB framework for modelling and simulation. These models are formalizable and many pedagogical benefits can be realized from integrating formal reasoning with simulation. *Mathematica* is a popular tool for computer algebra and an excellent medium for developing courseware [Sco91, Kal97]. *Mathematica* and other computer algebra systems can be enhanced with support for proof development, as has been done with REDLOG [DS97]. The combination of modelling, proof construction, and computer algebra tools can replace textbooks by *livebooks* that combine information with software to support experimentation, visualization, and proof. The abstraction tools that are mastered during a course can then become part of the student's toolkit.

5 Opportunities for Formal Methods in Science

Formal modelling and analysis techniques are already being used in some sciences to achieve a system-level understanding. For example, gene regulatory networks can be modelled as Boolean networks, hybrid systems, or stochastic systems and analyzed for properties using deduction and model checking. We propose a **Systems X** initiative for taking a systems level view of different processes, both natural and artificial. These systems level models capture some abstraction of the actual target system using state machines and constraints. Formal techniques can then be used to deduce system-level properties from these models, and this analysis can be used to test theories, diagnose problems, and make predictions.

6 Opportunities for Formal Methods in Industry

As we have already mentioned, there are about a hundred researchers at Microsoft working on Formal Methods. Many other technology companies like Intel, AMD, Rockwell-Collins, and MathWorks are actively developing and using formal tools. Electronic design automation vendors like Cadence, Mentor Graphics, and Synopsys have significant formal verification products in their tool suite. In Microsoft, there are about ten major projects developing formal software engineering tools for specification, annotation, testing and test generation, model checking, and security checking. Many of these projects are centered around powerful engines for deduction and symbolic model exploration. There is clearly a need for such tools in domains with stringent safety and security requirements. However, the more powerfully automated tools can be inserted into integrated development environments (IDEs) and compilers. One particular challenge for formal methods is the development of ultra-large systems that integrate software and hardware components from diverse sources. With the increasing awareness of the capabilities of formal tools and the strengthened requirements of reliability and assurance, we can expect a substantial uptake in the deployment of formal methods in industry. We can also expect industrial research laboratories to contribute significant research results as well as prototypes.

7 Conclusions

The three decades from 1975 to 2005 can be regarded as the golden age of research into formal methods. Work during these decades yielded a number of significant advances in program logics, program semantics, specification and programming languages, design methodologies, static analysis techniques, automated reasoning and model checking tools, and interactive proof assistants. During the next three decades, the focus of formal methods will be on expanding the scope and audience for these methods. On the one hand, these techniques should be made sufficiently powerful so that they can be used without any overt intellectual effort. On the other hand, the complementarity between human insight and judgment and computer-driven calculation and inference can be exploited as a key tool in managing information complexity. Over these next three decades, we can expect formal methods to not only enrich computer science, but many other disciplines that can benefit from formalization and inference. It is our hope that at the end of these four decades, we will have substantially realized Leibniz's dream that instead of resorting to technical arguments, we will be able to sit at a computer and say "*Let us calculate.*"

References

- [BM91] R. S. Boyer and J. S. Moore. MJRTY – a fast majority vote algorithm. In R. S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 105–117. Kluwer, Dordrecht, The Netherlands, 1991.

- [DS97] A. Dolzmann and T. Sturm. REDLOG: Computer algebra meets computer logic. *SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation)*, 31(2):2–9, 1997.
- [EVF07] Carl Eastlund, Dale Vaillancourt, and Matthias Felleisen. Acl2 for freshmen: First experiences. In *Seventh International Workshop on the ACL2 Theorem Prover and its Applications*. 2007.
- [Kal97] Erich Kaltofen. Teaching computational abstract algebra. *J. Symb. Comput*, 23(5/6):503–515, 1997.
- [Sco91] D. S. Scott. Exploration with Mathematica. In Richard F. Rashid, editor, *CMU Computer Science: A 25th Anniversary Commemorative*, pages 505–519. Addison-Wesley, pub-AW:adr, 1991.
- [TKB01] Allen B. Tucker, Charles F. Kelemen, and Kim B. Bruce. Our curriculum has become Math-Phobic! In *Proceeding of the Thirty-second SIGCSE Technical Symposium on Computer Science Education (SIGCSE-01)*, volume 33.1 of *ACM Sigcse Bulletin*, pages 243–247, New York, February 21–25 2001. ACM Press.
- [TRS03] Ashish Tiwari, John Rushby, and Natarajan Shankar. Invisible formal methods for embedded control systems. *Proceedings of the IEEE*, 91(1):29–39, January 2003.
- [Win00] Jeannette M. Wing. Weaving formal methods into the undergraduate computer science curriculum. In Teodor Rus, editor, *Algebraic Methodology and Software Technology. 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000, Proceedings*, volume 1816 of *Lecture Notes in Computer Science*, pages 2–9. Springer, 2000.