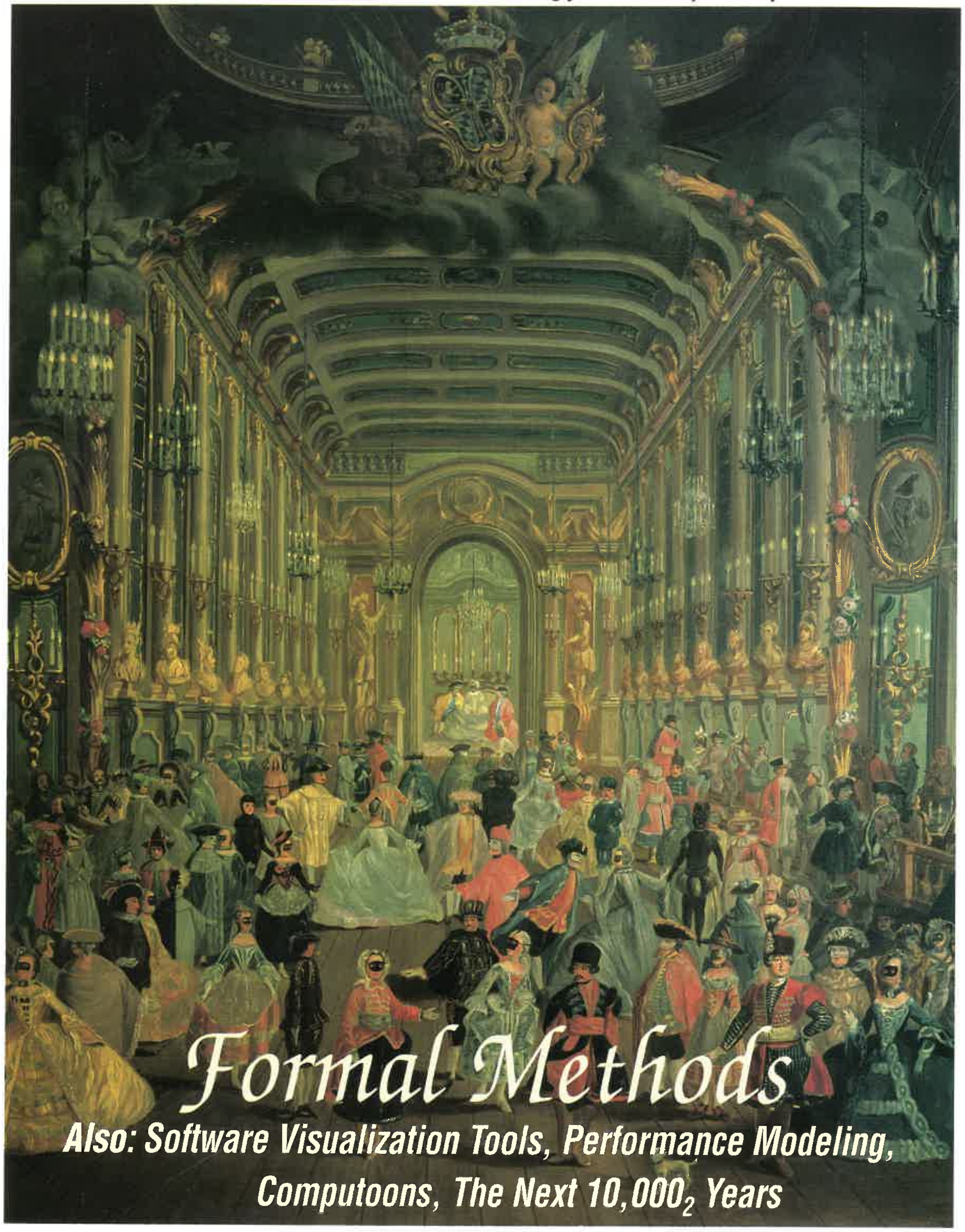
APRIL 1996

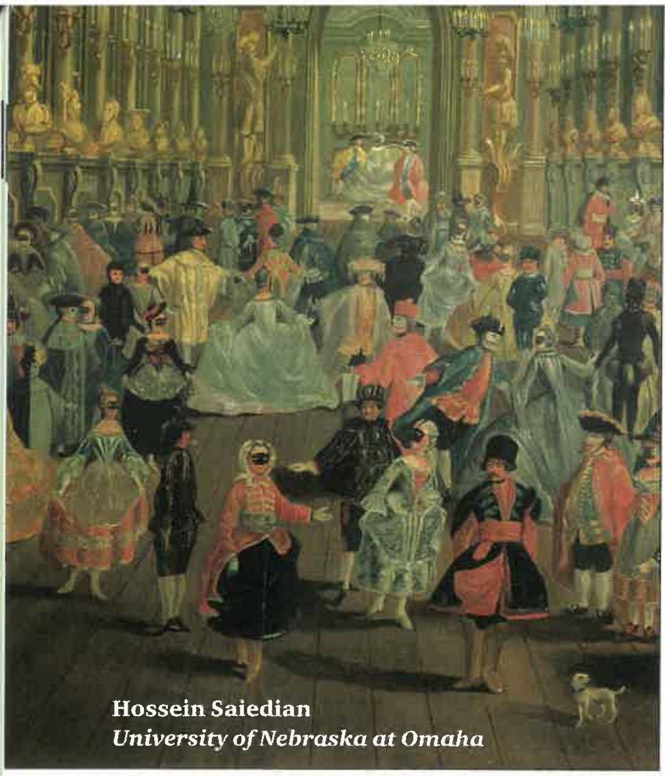
COMPUTER

Innovative technology for computer professionals


Formal Methods
Also: Software Visualization Tools, Performance Modeling, Computoons, The Next $10{,}000_2$ Years

IEEE COMPUTER SOCIETY
50 YEARS OF SERVICE • 1946-1996

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.



Hossein Saiedian
University of Nebraska at Omaha

An Invitation to Formal Methods

Featuring:

Jonathan P. Bowen, ***University of Reading***

Ricky W. Butler, ***NASA Langley Research Center***

David L. Dill, ***Stanford University***

Robert L. Glass, ***The Software Practitioner***

David Gries, ***Cornell University***

Anthony Hall, ***Praxis***

Michael G. Hinchey, ***New Jersey Institute of Technology***

C. Michael Holloway, ***NASA Langley Research Center***

Daniel Jackson, ***Carnegie Mellon University***

Cliff B. Jones, ***University of Manchester***

Michael J. Lutz, ***Rochester Institute of Technology***

David Lorge Parnas, ***McMaster University***

John Rushby, ***SRI International***

Jeannette Wing, ***Carnegie Mellon University***

Pamela Zave, ***AT&T Research***

One of the most challenging tasks in software system design is to assure reliability, especially as these systems are increasingly used in sensitive and often life-critical environments such as medical systems, air traffic control, and space applications. It is therefore essential for developers to employ those methods that offer a high degree of assurance that the system's requirements accurately capture the users' critical requirements and that an implementation in software (or hardware) is an accurate realization of the design.

Software reliability, although the primary concern of the computing industry, is not the only one. Another major concern has been the increased cost of developing and maintaining software. Precise, up-to-date figures representing the actual cost of software in industry are neither available nor easy to establish. Boehm suggested that the worldwide cost of software in 1985 was roughly $140 billion ($70 billion in the US alone).[1] Even if we assume a modest growth rate of 10 percent per year (Boehm suggested 12), the software cost in the year 2000 will be close to $600 billion ($300 billion in the US). Much of software's cost stems from testing and maintenance. The absence of rigorous practices that eliminate residual specification and design errors (caused by imprecision, ambiguity, and sometimes, plain mistakes) has translated into a significant portion of this cost.

Many claim that formal methods not only provide added reliability but also have the potential to reduce costs. They provide a notation for the formal specification of a system whereby the desired properties are described in a way that can be reasoned about, either formally in mathematics or informally but rigorously. In essence, formal methods offer the same advantages for software (or even hardware) design that many other engineering disciplines have exploited—namely, mathematical analysis using a mathematically based model. Such models allow the designer to predicate the behavior and validate the accuracy of a system instead of having to rely entirely on nonassuring exhaustive testing.

The clear advantages of a more mathematical approach to software design has certainly been well documented; the literature contains many excellent examples of applications of formal methods for large, critical, or even business transaction systems. Despite the evidence, however, a large percentage of practitioners see formal methods as irrelevant to their daily work.

To be sure, practitioners can't simply be blamed for their skepticism. What is hindering the adoption of formal methods by industry? Is the notation too complex? Or are existing formal methods unable to contribute to, benefit from, or be integrated with traditional methodologies? Is there still a perception of high complexity and costs? Has the barrier been the lack of easy-to-use support tools? Or would an increase of formalism in computer science or software engineering education translate

0018-9162/96/$5.00 © 1996 IEEE

into a more formal approach in industry by the next generation of software engineers?

In seeking answers to these questions, I contacted some of the best minds in academia and industry, asking them to share their insights.

ROUNDTABLE CONTRIBUTIONS

We open with a point/counterpoint on the viability of formal methods in industrial practice. Two long-time advocates of formal methods, Jonathan Bowen and Michael Hinchey, identify four areas they believe are important in the industrial adoption of formal methods: standards, tools, training, and correcting the misconceptions. Robert Glass speaks for practitioners who perceive a chasm between researchers and practitioners. He argues that it will be impossible to advance formal methods in industry until the chasm is effectively bridged.

Formal methods light

Formal methods can be applied to all or selected components of a system in varying degrees, depending on the criticality and nature of that system. Of course, increasing the level of formality may imply increased costs (for example, in formal analysis and verification). Is it possible to maintain acceptable levels of rigor without complete formalization, or to use partial formalism during modeling and analysis? Commentaries by Cliff Jones and by Daniel Jackson and Jeannette Wing address these issues. While emphasizing rigor and the importance of a formal basis, Jones suggests using a less-than-completely formal approach in most development cases. Jackson and Wing, on the other hand, advocate a lightweight approach to formal methods in which cost-effectiveness is achieved by developing partial specifications (with a focused application) that can be analyzed mechanically. The authors discuss partiality in terms of language, modeling, analysis, and composition.

Formal methods in practice

The experts from industry examine formal methods from the vantage point of their direct experiences. Anthony Hall elaborates on their economic benefits. His industrial experiences in using formal methods for relatively large projects (see his recent article in the March issue of *IEEE Software*) suggest that such methods may indeed make software development cheaper if used as part of an overall engineering approach. David Dill and John Rushby discuss the successful application of formal verification in hardware design. Formal methods have substantially more practical impact on hardware design, they claim, not only because hardware is easier to verify, but also because the problem has been approached with an eye to maximizing ROI. Here, formalists target areas of high payoff, use highly automated techniques, and apply formal verification more to debugging than assurance. Michael Holloway and Ricky Butler discuss three primary impediments to the wide-scale industrial use ~~of formal~~ methods: inadequate tools, inadequate examples, and the "build it and they will come" expectations. Finally, Pamela Zave discusses the conceptual gaps between the mathematical models offered by formalists and actual application domains.

Resources

For a comprehensive list of formal methods resources, try this URL: http:// www.comlab.ox.ac.uk/archive/formalmethods.html.

Engineering mathematics

If software engineering is in fact an engineering discipline, then application of formal methods should parallel the use of mathematics in other engineering disciplines. Michael Lutz believes that the gap between researchers and industrial practitioners is largely due to a misunderstanding of the differences between research mathematics and engineering mathematics. He outlines these differences and suggests some approaches that may accelerate the use of formal methods in industry. David Parnas, who was the first winner of the Norbert Wiener Award for Professional and Social Responsibility, also believes that formal methods should be more like the mathematics used by other engineering disciplines. While emphasizing the importance of routinely using mathematics in software engineering, Parnas expresses concerns about the notation purveyed by most formal methods researchers.

Education

The term *formal* in formal methods derives from formal logic, a powerful tool intended for reasoning and certifying certain properties. But has this topic been adequately presented as a useful tool? David Gries, who is known for significant contributions to computing education and who won the 1995 Karl V. Karlstrom Outstanding Educator Award, reflects on the role of logic as the foundation of most formal methods. He argues that until people are comfortable using formal logic, they won't be comfortable with most formal methods.

THIS ROUNDTABLE WILL, it is hoped, serve as a beginning for a more serious forum in which academics and practitioners can sit down together to discuss their needs, expectations, and goals in an effort to close the gap between them. I am open to suggestions about how to proceed from here. ▌

Acknowledgment

I thank Scott Hamilton for helping organize and coordinate this roundtable.

Reference

1. B. Boehm, "Improving Software Productivity," *IEEE Computer*, Vol. 20, No. 10, Sept. 1987, pp. 43-58.

Hossein Saiedian *is an associate professor in the Department of Computer Science at the University of Nebraska at Omaha. His research interests include software engineering models, formal methods, and object-oriented computing. E-mail hossein@cs.unomaha.edu.*

Formal Methods: Point-Counterpoint

TO FORMALIZE OR NOT TO FORMALIZE?

Michael G. Hinchey, ***New Jersey Institute of Technology and University of Limerick***
Jonathan P. Bowen, ***University of Reading***

It may seem strange to have two academics contributing to such a roundtable. However, our recent work in editing a collection of essays on the industrial application of formal methods has brought us into contact with a wide range of industrial projects, giving us some useful insights.[1] Broadly speaking, we can identify four reasons for industry's reluctance to take formal methods to heart.

MISCONCEPTIONS OR MYTHS. Claims that formal methods can guarantee correct hardware and software, eliminate the need for testing, and so on have led some to believe that formal methods are something almost magical.[2] Similarly, and even more significantly, myths that formal methods are difficult to use, delay the development process and raise development costs have led many to believe that formal methods are not for them. We hope that the examples in our study will help to dispel some of these myths.

Formal methods are not a panacea; they are just one of several techniques that, when correctly applied, have resulted in systems of the highest integrity. Formality should be used appropriately and judiciously at the weakest links in the chain of development, in tandem with other techniques of proven benefit for improved computer-based systems.

STANDARDS. Formal methods should not be applied to satisfy a whim or to be in vogue. Realistically, the first thing that must be determined before a formal development is undertaken is that formal methods are needed—whether for increased confidence in a system, to conquer complexity, or, increasingly, to satisfy the standards set by procurers or various regulatory bodies.

Until recently, formal methods were not included in that latter category. Now, however, standards bodies are not only using formal methods in making their own standards less ambiguous but have strongly recommended, and in the future may mandate, the use of formal methods in certain classes of applications.[3]

Michael G. Hinchey *is a professor in the Department of Computer and Information Science at New Jersey Institute of Technology and in the Department of Computer Science & Information Systems at University of Limerick, Ireland. His interests include formal methods for system specification, Z, verification, concurrency, functional programming, and method integration. E-mail hinchey@cis.njit.edu.*

Jonathan P. Bowen *is a lecturer at the Department of Computer Science, University of Reading. His interests include formal specification, Z, provably correct systems, rapid prototyping using logic programming, decompilation, hardware compilation, safety-critical systems, and software/hardware codesign. E-mail J.P.Bowen@reading.ac.uk.*

TOOLS. Just as significant investment in compiler technology was required for the widespread take-up of high-level languages, a significant investment in formal methods tools is required for industrial application.

Most of the projects in our study necessitated a considerable emphasis on tool development initially, because the required tools were often simply not available off the shelf. There is certainly a need for, and trend toward, further tool support and tool integration. Thankfully, excellent tools are now commercially available, and more integrated toolkits are evolving.

We expect and hope that in the future more emphasis will be placed on IFDSEs (Integrated Formal Development Support Environments) to support formal development, just as CASE workbenches support system development using more traditional structured methods. Unfortunately, this requires significant financial investment, which to date the formal methods market has found difficult to sustain. This "chicken and egg" situation must be resolved if formal methods are to gain wider acceptance.

EDUCATION AND TRAINING. We strongly believe in the applicability of formal methods to industrial-scale problems. However, technology transfer from academic theory to industrial practice must be addressed.

One way to cost-effectively integrate formal methods is to investigate how formal methods can be combined with existing structured and other methods already in use. In addition, novice formal developers must ensure that they have access to expert advice. Most successful formal methods projects have benefited from the guidance of at least one outside expert. Until sufficient local expertise has been built up, it appears difficult to use formal methods successfully without such consultants.

In addition, the necessary grounding for the use of formal methods has either not been taught adequately in computer science courses or specialists in industry simply have not taken such courses. Thus, many software engineers (and even more managers) shy away from formal methods because they feel out of their depth. Fortunately, many undergraduate courses (at least in Europe, and particularly in the U.K.) now teach the requisite foundations for applying formal methods.

However, formal aspects of courses are often unintegrated with the rest of the syllabus. Coordinated course material in which the mathematical foundations are subsequently applied to practical problems will help produce more professional software engineers and formal developers.

THE "STICK" OF STANDARDS and the "carrot" of education, supported by industrial-strength tools, will make or break the significant industrial use of formal methods. The market sector with the greatest potential to combine these elements most effectively is probably safety-critical systems.[3] This is an increasingly important area for computer-based systems because of the flexibility that software provides. We hope that formal methods will prove their worth in helping to prevent the loss of human lives where such systems are involved.

References

1. M.G. Hinchey and J.P. Bowen, eds., *Applications of Formal Methods*, Prentice-Hall Int'l Series in Computer Science, Hemel Hempstead, UK, 1995.
2. J.A. Hall, "Seven Myths of Formal Methods," *IEEE Software*, Vol. 7, No. 5, Sept. 1990, pp. 11-19.
3. J.P. Bowen and M.G. Hinchey, "Formal Methods and Safety-Critical Standards," *Computer*, Vol. 27, No. 8, Aug. 1994, pp. 68-71.

FORMAL METHODS ARE A SURROGATE FOR A MORE SERIOUS SOFTWARE CONCERN

Robert L. Glass, *The Software Practitioner*

There is a serious problem with formal methods in the software profession. The problem goes well beyond the fact that academics engage in wishful thinking about their value, and practitioners engage in skepticism and resistance to their use. The problem is lodged in what many are calling a chasm that exists between software in academe and software in industry.

The essence of the chasm is that those two groups see software engineering as different concepts, with different needs, in different terms, and through different lenses. Formal methods are the epitome of, and an interesting surrogate for, that chasm. Academics see formal methods as inevitable in the future of the software profession (the charge to write this roundtable response said "It is clear that formal methods provide a high degree of assurance . . . that systems will operate as desired"). Practitioners see formal methods as irrelevant to what they do (my skeptical practitioner side responded to that charge with "Clear to whom?").

That chasm was articulated nicely in a prior roundtable, the one in which Ted Lewis collected opinions on "Where Is Software Headed?" (*Computer*, Aug. 1995) and noted that his most dramatic finding was that academic and practitioner responses were almost entirely unrelated to one another. Specifically, formal methods appeared on the academic list defining the future of software but did not appear on the industrial list.

In my view, it is pointless to try to address the advance of formal methods in industry until the broader problem of the chasm is addressed. After all, formal methods have been around now for at least 25 years (I recall a paper on proof of correctness being published in *Computing Surveys* in the late 1960s), well past the period of time Redwine and Riddle found to be typical for technology transfer of new concepts into practice. If a generation of brilliant software academics has failed to achieve the transfer of formal methods to practice in the past, why should we think the current crop will be able to do it?

What is most badly needed, I believe, is an ongoing forum in which academics and practitioners meet to discuss their views of software. In that forum there must be an absence of preaching and a great deal of listening. Religious zealots of any stripe, including unquestioning advocates of formal methods (and any other technique or concept), should be banned. The goal of this forum would be to seek truth, not to coerce converts. Because I have walked on both sides of this chasm, I am convinced that such a forum, properly conducted, would be a dramatic learning experience for practitioners and academics alike.

However, the point of this roundtable is to present the viewpoints of its participants. Let me provide my own view of what such a forum might conclude regarding formal methods.

First of all, I believe this forum would see that formal methods are underdefined. Some see formal methods as "any rigorous technique" for software development, while others see it as being limited to software specification and formal verification. (I have seen both definitions in the same formal methods paper!) Until the subject is properly defined, no discussion of the topic can have a successful outcome.

Second, I believe this forum would see that formal methods are underevaluated. I collect evaluative research findings, and my cupboard of formal methods findings is nearly bare. I see some anecdotal evidence of its value (as editor of Elsevier's *Journal of Systems and Software*, I have published a few myself), and I see plenty of analysis that concludes there is enormous value. But I see very little in the way of cost-benefit studies—experimental or otherwise—that give me any confidence that the methods have value.

Third, I believe such a forum would seriously question whether formal methods are taking us in the right direction. There is a strong view (although still in the minority) in contemporary software engineering that software malleability is at the essence of the problem solutions we prepare and that formal methods inhibit that malleability. Further, there is another belief—this one accepted by most—that specifications must be readable by the intended customers/users of the software product and that formal methods do not fulfill that need.

Finally, such a forum would reject the belief that formal methods will lead to any breakthrough in software productivity/quality, such as through automatic code generation from formal specifications, on the same grounds set forth in articles by Rich and Waters ("Cocktail Party Myth"), Brooks ("No Silver Bullet"), and Parnas ("Star Wars").

THE POINT OF A ROUNDTABLE, of course, is to collect a stimulating set of viewpoints to help catalyze the reader into thinking more clearly about the roundtable's subject. I would suggest that, for this topic, a roundtable should be only the beginning. Someone has to take responsibility for beginning that forum mentioned above. Why couldn't it start here? ▌

***Robert L. Glass** is president of Computing Trends, a software engineering consulting/education/publishing firm. He has been active in the field of computing and software for over 40 years, largely in industry (several aerospace companies) but also as an academic (Seattle University, Software Engineering Institute). He is editor of the Journal of Systems and Software and publisher and editor of the* Software Practitioner.

A RIGOROUS APPROACH TO FORMAL METHODS

Cliff B. Jones, *University of Manchester*

I have been active in formal methods for more than twenty years—and may have been one of the first people to use the term. Having experienced the early problems of gaining recognition for the subject, I am actually pleased with today's level of awareness of more formal approaches to computer systems development. But this roundtable asks us to address why more has not been achieved. My own views on how to foster the use of formal methods have changed over time, and the proposals I make now for how an organization might use formal methods differ from the more austere suggestions I put forth a decade and more ago.

My position today could be characterized as "formal methods light." Interestingly, my 1980 book on formal methods deliberately had the word *rigorous* rather than *formal* in its title. My opinion even then was that it was important to understand the formal basis but to use—in most cases—a less than completely formal approach; this course was proposed on the assumption that one was capable of filling in the formal details where necessary. Under the banner of "fm-light," I today teach courses on how to sketch abstract models of systems where a minimum of emphasis is put on the notational detail, and the central idea is that of presenting an abstract state for a system. It is amazing how much understanding of an architecture is captured in its state. For a database system, less than half a page of state formulae can show the overall function; even for a programming language whose definition might cover more than a hundred pages of formulas, the semantic objects capturing the language's essence most likely can be presented on a few pages. The step beyond presenting just a state is to provide a specification of major parts—or possibly all—of a system; and only in extreme cases would I recommend formal proofs for design steps of (portions of) a system. In contrast to the emphasis in much of the literature, I would prefer to see formalism employed to justify the early data-structure/design decisions rather than the detailed control constructs.

Coupled with the idea that a rigorous development is capable of being formalized, we have made numerous experiments with formal documents in inspections or walk-throughs. I remember when I worked for IBM that—with traditional development methods—those inspections conducted late in a development cycle tended to be crisp and useful in locating errors, whereas inspections conducted earlier in the project tended to be woollier and not so useful in locating errors. I believe the reason for this was clearly the lack of formal material available in a project's early stages—the later reviews had the formalism of the code to ensure precision. We all know that the expensive mistakes are made in a project's early stages: Undetected errors in the specification and overall design phases are very expensive to fix. What I have found with specification and design inspections based on formal documents is that they achieve the same level of precision as code inspections and, with that, a considerably greater level of confidence. Furthermore, it is precisely in the review process that the virtue of formalizing rigorous development becomes clear: Doubts raised in an inspection can be resolved by going to another level of formalization.

***Cliff B. Jones** is professor of computing science at Manchester University. Prior to that, he worked for IBM, where his involvement in software testing of major projects led to a deep dissatisfaction with the development methods being used. While at the IBM Vienna Laboratory, Jones helped create the formal development method known as VDM. E-mail cliff@cs.man.ac.uk.*

Perhaps the area where I have changed my views most concerns the necessary education for groups of engineers. I have had bad experiences seeing system architects propose designs in natural language and ask others to formalize them. The inevitable effect was that the people constructing the formal document generated many questions about—and corrections to—the architecture's natural language description and were thanked only with more pages of ambiguous and inconsistent natural language. I had formed the view that the only way to avoid this trap was for all team members to be able to write formal specifications themselves; domain experts would then use formal notations to record and think about their architectures. When I consulted for the IBM Laboratory in Germany, we attempted to educate all people involved in a project (including those from publications) about VDM to a level where they could actually write specifications themselves.

I now feel that it is more realistic to follow the pattern of operations research groups (in which I had the pleasure of working early in my career in the oil industry). A combined team of domain specialists and formalists would, I believe, solve many of the problems of adopting formal methods. The major difficulty in communicating formal methods is the ability to find appropriate levels of abstraction at which to describe serious systems. People capable of doing this would become the team's formal methods specialists who would help the domain specialists understand the system they wished to build and document it at an appropriate level of abstraction. Education in general, and specifically education in the use of abstraction, is a bigger gap than the claimed lack of tools for formal methods. Clearly, all team members need to be able to read the formal specification language, but it is not so much the minutiae of discrete mathematics that inhibits users as it is the ability to abstract.

Today, formal methods are mainly used in the safety-critical area where their detailed application can be justified because of the danger of loss of life. The use of formal methods in a lighter way is both a key to using them on larger-scale applications and a way of penetrating fields outside the safety-critical area. I'd like to see formal methods applied to those (simply) critical applications on which the future of a company depends or whose software systems can affect society at large.

A DANGER OF THIS FM-LIGHT approach is that formalists will obtain no thanks for their efforts! Many contributions of formalism are now absorbed into everyday computing know-how (for example, context-free grammars, finite-state diagrams, and so on), yet there is still a question as to the impact of formal methods. When states, data-type invariants, retrieve functions, loop invariants, rely/guarantee-conditions are all part of general computer knowledge, the formalists will be challenged to justify their continuing research on other topics. ▮

LIGHTWEIGHT FORMAL METHODS

Daniel Jackson and Jeannette Wing,
Carnegie Mellon University

Many benefits promised by formal methods are shared with other approaches. The precision of mathematical thinking relies not on formality but on careful use of mathematical notions. You don't need to know Z to think about sets and functions. Likewise, the linguistic advantages of a formal notation rely more on syntax than semantics.

Mechanical analysis, in contrast, is a benefit unique to formal approaches. An engineer's sketch can communicate ideas to other engineers, but only a detailed plan can be rigorously examined for flaws. Informal methods often provide some analysis, but since their notations are generally incapable of expressing behavior, the results of the analysis bear only on the properties of the artifact's description, not on the properties of the artifact itself.

For everyday software development, the purpose of formalization is to reduce the risk of serious errors in specification and design. Analysis can expose such errors while they are still cheap to fix. Formal methods can provide limited guarantees of correctness too, but, except in safety-critical work, the cost of full verification is prohibitive, and early detection of errors is a more realistic goal.

To make analysis economically feasible, the cost of specification must be dramatically reduced, and the analysis itself must be automated. Experience (of several decades) with interactive theorem proving has shown that the cost of proof is usually an order of magnitude greater than the cost of specification. And yet the cost of specification alone is often beyond a project's budget. Industry will have no reason to adopt formal methods until the benefits of formalization can be obtained immediately, with an analysis that does not require further massive investment.

Existing formal methods, at least if used in the conventional manner, cannot achieve these goals. By promoting full formalization in expressive languages, formalists have unwittingly guaranteed that the benefits of formalization are thinly spread. A lightweight approach, which in contrast emphasizes partiality and focused application, can bring greater benefits at reduced cost. What are the elements of a lightweight approach?

PARTIALITY IN LANGUAGE. Until now, specification languages have been judged primarily on their expressiveness, with little attention paid to tractability. Some languages—such as Larch—were from the start designed with tool support in mind, but they are the exception. Tools designed as an afterthought can provide only weak analysis, such as type checking. The tendency (in Z especially) to see a specification language as a general mathematical notation is surely a mistake, since such generality can only come at the expense of analysis (and, moreover, at the expense of the language's suitability for its most common applications).

PARTIALITY IN MODELING. Since a complete formalization of the properties of a large system is infeasible, the question is not whether specifications should focus on some details at the expense of others, but rather which details merit the cost of formalization. The naive presumption that formalization is useful in its own right must be dropped. There can be no point embarking on the construction of a specification until it is known exactly what the specification is for; which risks it is intended to mitigate; and in which respects it will inevitably prove inadequate.

PARTIALITY IN ANALYSIS. A sufficiently expressive language, even if designed for tractability, cannot be decidable, so a sound and complete analysis is impossible. Most specifications contain errors, and so it makes more sense to sacrifice the ability to find proofs than the ability to detect errors reliably. A common objection to this approach is that it reduces analysis to testing: No reported errors does not imply no actual errors. But this much-touted weakness of testing is not its major flaw. The problem with testing is not that it cannot show the absence of bugs, but that, in practice, it fails to show their presence. A model checker that exhausts an enormous state space finds bugs much more reliably than conventional testing techniques, which sample only a minute proportion of cases.

PARTIALITY IN COMPOSITION. For a large system, a single partial specification will not suffice, and it will be necessary to compose many partial specifications, at the very least to allow some analysis of consistency. How to compose different views of a system is not well understood

***Daniel Jackson** is an assistant professor of computer science at Carnegie Mellon. After receiving a BA in physics from Oxford, he worked as a programmer for Logica UK. He has a PhD in computer science from MIT. He is interested in automated analysis of specifications and designs; requirements; and reverse engineering. E-mail daniel.jackson@cs. cmu.edu.*

***Jeannette Wing** is an associate professor of computer science at Carnegie Mellon. She received her SB, SM, and PhD in computer science all from MIT. Her research interests are in formal methods, programming languages, and distributed systems. E-mail wing@cs.cmu.edu.*

and has only minimal support from specification languages, since it does not fit the standard pattern of "whole-and-part" composition.

MUCH OF WHAT WE SAY HERE IS AT ODDS with the conventional wisdom of formal methods. The notion of a lightweight approach is radical, however, only in its departure from a dogmatic view of formal methods that is detached from mainstream software development. In the broader engineering context, the suggestion of pragmatic compromise is hardly new.

A lightweight approach, in comparison to the traditional approach, lacks power of expression and breadth of coverage. A surgical laser likewise produces less power and poorer coverage than a light bulb, but it makes more efficient use of the energy it consumes, and its effect is more dramatic. ▮

Industrial Practice

WHAT IS THE FORMAL METHODS DEBATE ABOUT?

Anthony Hall, *Praxis*

It is extraordinary that formal methods cause such fierce debate. Some proponents seem committed with an almost religious fervor; some opponents seem hostile beyond all reason. As far as I know, no issue in software engineering causes as much passion, unless it is the use of the goto statement.

One reason for this polarization may be that the two sides are arguing from completely different premises. Perhaps the argument has been between those who say that formal methods are essential because they are the only way to gain assurance and those who say formal methods are impossible because they are too expensive. No amount of argument will resolve that difference unless the two sides start to recognize each other's objectives.

I have been using formal methods in real projects for the past 10 years, and recently I have begun to see a fundamental shift in the argument. Ten years ago, the argument was that formal methods were hugely expensive (they were!) but that you had to use them because there was no other way to ensure that your software was correct. Now, the argument is quite different. We know that it is possible to produce software, even critical software, without formal methods; we also know that it is horribly expensive. What is only recently becoming clear is that it is practical to produce software, even noncritical software, using formal methods; it is also, as far as we can tell, *cheaper* to do it that way.

I say "as far as we can tell" because it is notoriously difficult to get any useful information from software metrics. What I can do is describe some of the projects we have done at Praxis and how we perceive the costs and benefits of using formal methods.[1,2]

***Anthony Hall** is a principal consultant with Praxis, a British software engineering company. He led the analysis and design team on CDIS and has promoted the use of formal methods on many projects. His current interests are in formal aspects of software architectures and in tool-based verification of formal specifications and designs. E-mail jah@praxis.co.uk.*

One of the largest applications of formal methods I know of is a project we completed a few years ago to develop an air traffic control information system called CDIS. This is a safety-related system, but there was no regulatory pressure to use formal methods for that reason. However, we wanted to make sure that we understood the requirements accurately and decided to use formal methods at the early stages of the life cycle to help us do that. We therefore wrote a formal specification of the whole system as the basis for our development. There are about 150 user-level operations in CDIS (the final system is about 200,000 lines of code), so the specification is a large document (about 1,000 pages). This in turn means that we are making fairly "shallow" use of formality—we did not attempt any proofs of consistency or of particular properties. Nevertheless, we found the specification enormously useful in pinning down just what it was that we were going to build.

The system specification was not the only way we used formality on CDIS. We also wrote a similar specification of the main design-level modules, again at a shallow level. In one particular part of the design we used formality in a much deeper way, writing detailed process specifications and attempting to prove them correct. The fact that some proofs failed demonstrated that our design was in fact incorrect, and as it turned out, incorrect in a way that might well have escaped detection in our tests. Fault metrics for CDIS confirmed our hope that it would be of higher quality than systems built using conventional methods. They also showed an unusual distribution in that, unlike many other systems, very few of the faults that survived system test into the delivered system were requirements or specification faults.

Most interesting is the fact that none of this good news cost us anything—our productivity on the project was as good as or better than if we had done it conventionally. I believe that one reason is that the work we put in at the early stages was effective in finding lots of errors that would, if we had not found them, have proved very expensive to correct later. The formal specification enabled us to find these errors effectively.

While CDIS is an example where formal methods at the front of the life cycle pay off, they can also show economic benefits at the code and test stages. For Lockheed, we have recently been analyzing the code for the avionics software for the C130J.[2] The software is coded in the Spark-annotated subset of Ada, working from specifications in the Software Productivity Consortium's Core notation. Here, too, many people would expect that the use of Spark would add to the software's cost, while improving its quality. In fact, however, the added quality decreases the cost of the software because of the huge savings in testing. The use of Spark annotations to capture the specification of the modules has truly led to software that is "correct by construction" and hence passes its tests instead of requiring expensive rework.

THESE TWO EXAMPLES support the prime contention of this article. We can now have a rational discussion about formal methods, because both sides can ask the same questions about them. I believe the right question to ask is "what can formal methods contribute to improve the quality and decrease the cost of our systems?"

References

1. A. Hall, "Using Formal Methods to Develop an ATC Information System," *IEEE Software*, Vol. 13, No. 2, Mar. 1996, pp. 66-76.
2. M. Croxford and J. Sutton, "Breaking Through the V and V Bottleneck," *Lecture Notes in Computer Science 1031*, Springer-Verlag, Berlin, 1996, pp. 344-354.

ACCEPTANCE OF FORMAL METHODS: LESSONS FROM HARDWARE DESIGN

David L. Dill, *Stanford University*
John Rushby, *SRI International*

Despite years of research, the overall impact of formal methods on mainstream software design has been disappointing. By contrast, formal methods are beginning to make real inroads in commercial hardware design. This penetration is the result of sustained progress in automated hardware verification methods, an increasing accumulation of success stories from using formal techniques, and a growing consensus among hardware designers that traditional validation techniques are not keeping up with increasing design complexity. For example, validation of a new microprocessor design typically requires as much manpower as the design itself, and the size of validation teams continues to grow. This manpower is employed in writing test cases for simulations that run for months on acres of high-powered workstations.

Do the benefits of the new method exceed the costs of converting to it and using it by a sufficient margin to justify the risks of doing so?

In particular, the notorious FDIV bug in the Intel Pentium processor has galvanized verification efforts, not because it was the first or most serious bug in a processor design, but because it was easily repeatable and because the cost was quantified (more than $400 million).

Hence, hardware design companies are increasingly looking to new techniques, including formal verification, to supplement and sometimes replace conventional validation methods. Indeed, many companies, including industry leaders such as AT&T, Cadence, Hewlett-Packard, IBM, Intel, LSI Logic, Motorola, Rockwell, Texas Instruments, and Silicon Graphics have created formal verification groups to help with ongoing designs. In several cases, these groups began by demonstrating the effectiveness of formal verification by finding subtle design errors that were overlooked by months of simulation.

Why have formal methods been more successful for hardware than for software? We believe that the overriding reason is that applications of formal methods to hardware have become cost-effective.

The decision to use a new methodology is driven by *economics*: Do the benefits of the new method exceed the costs of converting to it and using it by a sufficient margin to justify the risks of doing so? The benefits may include an improved product (for example, fewer errors), but those most keenly desired are reduced validation costs and reduced time-to-market (for the same product quality). The chief impediments to applying traditional formal methods are that the costs are thought to be high (for example, much highly skilled labor) or even unacceptable (a potential increase in time-to-market), while the benefits are uncertain (a possible increase in product quality). Formal hardware verification has become attractive because it has focused on reducing the cost and time required for validation rather than pursuit of perfection.

Of course, hardware has some intrinsic advantages over software as a target for formal methods. In general, hardware has no pointers, no potentially unbounded loops or recursion, and no dynamically created processes, so its verification problem is more tractable. Furthermore, hardware is based on a relatively small number of major design elements, so that investment in mastering the formal treatment of, say, pipelining or cache coherence can pay off over many applications. And the cost of fabricating hardware is much greater than software, so the financial incentive to reduce design errors is much greater.

However, we believe there are some lessons and principles from hardware verification that can be transferred to the software world. Some of these are listed below.

PROVIDE POWERFUL TOOLS. Technology is the primary source of increased productivity in most areas, and especially this one. In particular, tools that use formal specifications as the starting point for mechanized formal *calculations* are the primary source of cost-effective applications of formal methods. This is exactly analogous to the use of mathematical modeling and calculation in other engineering disciplines. Without tools to deliver tangible benefits, formal specifications are just documentation, and there is little incentive for engineers to construct them or to keep them up to date as the design evolves.

For hardware, a spectrum of tools has evolved to perform formal calculations at different levels of the design hierarchy and with different benefits and costs. At the lowest level are tools that check Boolean equivalence of combinational circuits (this is useful for checking manual circuit optimizations). Techniques based on Ordered

Binary Decision Diagrams (OBDDs) are able to check large circuits quite efficiently and are now incorporated in commercial CAD tools. At a higher level, designs can often be represented as interacting finite state machines, and tools that systematically explore the combined state space can check that certain desired properties always hold or that undesired circumstances never arise. Tools based on explicit state enumeration can explore many millions of states in a few hours; tools that represent the state space symbolically (using OBDDs) can sometimes explore vast numbers of states (for example, 10^{100}) in the same time and can check richer properties (for example, those that can be specified in a temporal logic, in which case the technique is called "temporal logic model checking"). At the highest levels, or when very complex properties or very large (or infinite) state spaces are involved, highly automated theorem-proving methods can be used to compare implementations with specifications. These theorem-proving methods combine rewriting and induction with decision procedures for propositional calculus, equality, and linear arithmetic. In all cases, the tools concerned are highly engineered so that they can deal with very large formulas and require little or no user interaction when applied in familiar domains.

USE VERIFICATION TO FIND BUGS. A tool that simply "blesses" a design at the end of a laborious process is not nearly as impressive to engineers as a tool that finds a bug. Finding bugs is computationally easier than proving correctness, and a potential cost can be attached to every bug that is found, making it easy to see the payoff from formal verification. Traditional validation methods already are used primarily as bug-finders, so formal methods are very attractive if they find bugs different from those found with traditional methods—a much more achievable goal than trying to guarantee correctness.

Shortcuts can be taken when formal verification is used for finding bugs rather than proving correctness. For example, a system can be scaled down—the number or size of components can be drastically reduced. A directory-based cache-coherence protocol can be checked with just four processors, one cache line, and two data values. Such a scaled-down description will still have many millions of states, but will be within reach of state exploration and model checking methods. These methods can check the reduced system *completely*; in contrast, simulation checks the full system incompletely. Both techniques find some bugs and miss others, but the formal methods often detect bugs that simulation does not. Some researchers are now applying these techniques to software.

FORMAL TECHNIQUES MUST BE TARGETED. In hardware, experience shows that control-dominated circuits are much harder to debug than data paths. Effort has thus gone into developing formal verification techniques for protocols and controllers rather than for data paths. Targeting maximizes the potential payoff of formal methods by solving problems not handled by other means. Notice that the targeted problems often concern the *hardest* challenges in design: cache coherence, pipeline (and now superscalar) correctness, and floating point arithmetic. For software, correspondingly difficult and worthwhile challenges include those where local design decisions have complex global consequences, such as the fault-tolerance and real-time properties of concurrent distributed systems.

RESEARCHERS SHOULD APPLY THEIR WORK TO REAL PROBLEMS. Our research priorities are completely different from what they would have been, had we not exercised our ideas on realistic problems. Such efforts have frequently raised interesting new theoretical problems, as well as highlighting the need for improvements in tools.

Of course, applying verification strategies to real problems is also crucial for building credibility. There is now a long string of success stories from academia and industry where finite-state verification techniques have been applied to hardware and protocols. A few documented examples include protocol bugs in IEEE standards for the FutureBus+ (found using symbolic model checking with a version of Carnegie Mellon's SMV system) and SCI (found using explicit state enumeration with Stanford's Murphi verifier), and formal verification of the microarchitecture and microcode of the Collins AAMP5 and AAMP-FV avionics processors (using theorem proving with SRI's PVS system). Several groups have also demonstrated the ability to detect bugs in the quotient-prediction tables of SRT division algorithms (similar to the Pentium FDIV bug), and some have been able to verify specific SRT circuits and tables. There have also been many unpublicized examples of problems found by industrial formal verification groups, which have helped them build credibility among designers and managers in their companies.

WE ATTRIBUTE THE GROWING ACCEPTANCE of formal methods in commercial hardware design to the power and effectiveness of the tools that have been developed, to the pragmatic ways in which those tools have been applied, and to the overall cost-effectiveness and utility that has been demonstrated. We believe formal methods can achieve similar success in selected software applications by following the same principles. ▌

David L. Dill *is an associate professor in the Department of Computer Science at Stanford University. He has done extensive research on formal verification of protocols and hardware designs, using a variety of different techniques. Currently, his primary research interest is finding ways to combine theorem-proving and model-checking verification methods. E-mail dill@cs.stanford.edu.*

John Rushby *is a program director in the Computer Science Laboratory of SRI International in Menlo Park California, where he leads a research program in formal methods and dependable systems. His group develops mechanized formal verification systems (the latest is called PVS), and applies them to problems in computer security, hardware design, and safety-critical and fault-tolerant systems. E-mail rushby@csl.sri.com.*

IMPEDIMENTS TO INDUSTRIAL USE OF FORMAL METHODS

C. Michael Holloway and Ricky W. Butler,
NASA Langley Research Center

NASA Langley Research Center has, for eight years, conducted and sponsored research to develop formal methods for high integrity applications. For the past four years, Langley's effort has focused primarily on projects to inject formal methods technology into commercial use. In particular, NASA Langley has sponsored SRI International to work with Rockwell-Collins to formally verify a portion of the AAMP5 microprocessor and fully verify the AAMP-FV, which is a new microprocessor being developed by Collins for potential use in critical applications. We have also funded Odyssey Research Associates to work with Honeywell Air Transport Systems to develop formally based tools and techniques to analyze and manipulate decision table specifications. Other projects have applied formal methods to fault-tolerant algorithms developed by Allied Signal, advanced control algorithms under development by Union Switch and Signal, and change requests to space shuttle software.

All these projects have met the goals we established for them. In fact, they have often been cited as good examples of industrial use of formal methods. Nevertheless, these projects have also illustrated that there remain serious impediments to the full acceptance of formal methods by industry. Some of the impediments frequently cited by the formal methods community include such industrial problems as inadequately educated engineers, the not-invented-here syndrome, and greater emphasis on reducing costs than on increasing safety. However, we believe that the primary causes for the lack of wide-scale industrial use of formal methods are (1) inadequate tools, (2) inadequate examples, and (3) a "build it and they will come" expectation. The projects cited above have succeeded because our partners have been able to overcome or avoid these pitfalls in one way or another. In the rest of this article, we will discuss each of the three factors, and provide suggestions as to what formal methodists can do to overcome them.

TOOLS. Inadequate tools are a serious impediment to industrial use of formal methods. Almost all formal methods researchers will acknowledge that most existing tools are not production-quality and are difficult to learn to use effectively. Not only do the input languages use specialized notations from mathematical logic, but many tools have numerous bugs in them. Few things are more disconcerting to formal methods neophytes than having to wonder constantly if their inability to complete a proof is a result of their own lack of skill, the falsity of what they are trying to prove, or a bug in the tool. Prototype tools must be built, but no one should expect such tools to be used regularly within industry.

One cause for optimism is that many of the formal methods tool developers are working on their third- or fourth-generation tools. Some of these evolving tools are also being developed with industrial use as a major goal and are being applied to increasingly sophisticated real problems. Also, government sponsors are working to build cooperation between different projects so that the methods and tools developed at different sites work together. This will also help reduce the duplication of effort that has unfortunately characterized the field.

Despite our optimism, increased vigilance is still needed on several fronts. Tool developers must curb their desires to create more and more powerful tools and instead expend increased effort on improving the robustness and performance of existing tools. The emerging tools should be thoroughly exercised within the formal methods community before they are made available to industry; government sponsors can take a large role in this area. Finally, when tools are made available to industry, formal methods experts must be available to provide guidance in the tool's use and ways to overcome its shortcomings. There is a growing need for training courses that are tailored for industrial users.

INADEQUATE EXAMPLES. A second impediment to industrial use of formal methods is the inadequacy of existing examples and models. For too long, formal methods researchers have worked on toy examples and intellectually interesting but industrially irrelevant problems. Although academic-style work is essential to the advancement of the field, it cannot be the only work that is done. Because so few researchers have concentrated on industrially relevant problems, most of the examples and models that have been developed have borne little resemblance to the examples and models needed in industry.

Industry cannot be expected to develop the needed models alone. The first attempt to formalize a new problem domain requires a significant time investment, one that is almost always longer than engineers with product deadlines can afford to make. The time required to formalize the underlying domain-specific knowledge in existing formal verification tools can be considerable, but this is often not recognized even by formal methodists until they tackle a real application. Also, the first endeavor in a new domain requires far greater creativity than subsequent efforts.

The solution to this impediment is conceptually simple: More formal methods researchers must become knowledgeable about the problem domains relevant to industry, and develop examples, models, techniques, and tools appropriate for those domains. This means that some formal methodists must be willing to tackle prob-

C. Michael Holloway *is a research engineer at the NASA Langley Research Center in Hampton, Virginia. He has been a member of the NASA Langley formal methods team since 1992. His research interests include application of formal methods to high-integrity software and programming language theory. E-mail c.m.holloway@larc.nasa.gov.*

Ricky W. Butler *is a senior research engineer at the NASA Langley Research Center. He leads the Formal Methods Team in the Flight Electronics Technology Division. His research is on the application of formal methods to high-integrity systems with particular focus on fault-tolerant architectures used for flight control and flight management. E-mail r.w.butler@larc.nasa.gov.*

lems that are not intellectually interesting but that industry needs solved. It also means that some must be willing to forgo using powerful and sophisticated methods, when simple and pedestrian ones will suffice. Much of the success of NASA Langley's program can be attributed to our contractors' willingness to attempt to understand important problem domains such as fault-tolerant computing, clock synchronization, flight control and management, and microprocessor design. Also, we have tackled mundane and tedious problems when necessary; and our contractors have created simple, special-purpose tools, when such tools were sufficient to meet the customer's needs.

"BUILD IT AND THEY WILL COME." The final impediment to industrial use of formal methods is the expectation that if one builds an advanced tool, it will be used. This viewpoint overlooks the gulf that exists between the research world and the industrial world. Industry rarely has the time and resources to keep up with and distill the vast number of ideas and tools that are emerging from the research community, and the research community is often ignorant of the challenges that industry faces. Literally hundreds of different kinds of formal methods have been promoted, but we lack adequate executive-level guidance on how to match particular methods with specific applications, and few unbiased appraisals exist of the maturity of current methods and tools.

Furthermore, the gulf between academia and industry has been widened by the rhetoric of some formal methods zealots who blame industry for its failure to embrace formal methods and question the intellectual acuity of industry engineers. These zealots have also exaggerated the importance of formal methods by claiming that only formal methods can prevent future catastrophes, and by telling industry that formal methodists know better than professional engineers how to build real systems.

The solution to this impediment is also conceptually simple: Formal methods researchers must stop giving answers and start asking questions. Instead of saying "We know what you're doing wrong and how to fix it," we should be asking "Where are you having trouble and what can we do to help you?" If the potential benefits of formal methods are as great as those of us in the field believe, then we should be able to demonstrate those benefits by dispassionate logic and empirical data. In short, we need more ambassadors and fewer warriors.

THE THREE MAJOR IMPEDIMENTS listed above can be overcome. We believe that NASA Langley's sponsorship of ground-breaking uses of formal methods in several domains has helped to show how this can be done. We hope that continued government sponsorship—from NASA Langley and from other sources—will help speed the process. ▮

FORMAL METHODS ARE RESEARCH, NOT DEVELOPMENT

Pamela Zave, *AT&T Research*

A telecommunications engineer is concerned with feature behavior. In simple cases, call forwarding is easy to understand: If directory number *A* is forwarded to directory number *B*, then a call dialed to *A* is actually connected to *B*, with both the caller and the subscriber of *A* paying some of the charges. But what if the calling subscriber has used a "call blocking" feature to ensure that no one calls *B* from his telephone? Alternatively, *B* may be forwarded to *C*. In that case, is a call to *A* connected to *B* or *C*? These difficult cases are called "feature interactions," and they proliferate endlessly. The engineer must find a way to describe the desired feature behavior that is complete, consistent, unambiguous, and—above all—maintainable as new features are introduced.

***Pamela Zave** received the AB degree in English from Cornell University, Ithaca, New York, and the MS and PhD degrees in computer sciences from the University of Wisconsin at Madison. She began her career as an assistant professor of Computer Science at the University of Maryland, College Park. Since 1981 she has been with AT&T Bell Laboratories at Murray Hill, New Jersey, and is now a Distinguished Member of Technical Staff in AT&T Research. E-mail pamela@research.att.com.*

Another telecommunications engineer is concerned with system architecture. There is an industry-wide trend toward implementing basic switching functions (setup and teardown of voice paths) in a network node separate from feature control and the database of subscriber information. The interface between a switching node and a feature node is a protocol that must be standardized, robust, and independent of switching technology. How can the engineer ensure that the protocol is rich enough to support all the interactions that will be needed between the nodes, now and in the future?

Another telecommunications engineer is concerned with customer programmability. He wants to make it possible for business customers with 800 numbers to program their own services, using building blocks such as touch-tone digit menus, prerecorded announcements, database queries, and voice synthesis. The resulting programs must be able to run as "trusted code" in a highly reliable system.

Presumably, all these engineers could benefit from formal methods. What shall we offer them? Finite-state machines? Typed set theory? Algebra, be it process, datatype, or relational? Logic, be it higher-order, deontic, or temporal? The fact is that the principles, objects, and relationships offered by these notations are absurdly different from the principles, objects, and relationships about which these engineers are concerned. There is no superficial similarity, and no easy way of applying any general-purpose formal method to these engineering challenges.

The conceptual gap between application domains and mathematics must be bridged by building mathematical models of the application domains. Within an appropriate model, formal language is extended to include the vocabulary and relationships of the domain. The lack of appropriate models, on the other hand, constitutes a large barrier to the use of formal methods in an application domain.

For many people, the answer to this problem is better education of software practitioners, so that they will be comfortable with formal methods, and better cost-benefit analysis, so that a significant initial investment in specification can be justified. I am skeptical of this approach, because it places the burden of bridging the conceptual gap on people who have a product-development schedule to meet. It underestimates the difficulty of finding really good models. It also ignores the example given to us by traditional engineering disciplines.

Consider first the characteristics that a successful model must have. The right abstractions must be chosen so that they illuminate rather than obscure important issues. Each notation must cleverly reduce description complexity, or the whole effort will collapse because of the size of the specifications needed. The aspects of the system that are formalized must be exactly those that benefit most from improved documentation, analysis, or code generation, because certainly not everything can be formalized.

Not surprisingly, it is easy to build the wrong model. For example, many articles on feature interaction have discussed the interaction of "call waiting" and "call forwarding on busy." Call waiting enables a person who is talking on an ordinary telephone to answer another incoming call and subsequently time-multiplex the two calls. "Call forwarding on busy" diverts to another destination a call to a busy telephone. Typically these two features are represented on an equal footing, and their interacting behavior is chosen in an arbitrary way.

Despite its popularity and plausibility, this is a disastrous way to look at the situation. It ignores the existence of many kinds of telephones, including multibutton telephones that time-multiplex several calls without call waiting (in fact, call waiting is just a way of imitating a multibutton telephone on an ordinary telephone).

It makes much more sense to separate time-multiplexing of calls from other features. Within the multiplexing module there can be interchangeable submodules, one for multibutton telephones and one for ordinary telephones with call waiting. Either submodule will define when its telephone is "busy," probably when there are as many active calls as its multiplexing capacity allows. Other features, such as "call forwarding on busy," can be specified in terms of the atomic term "busy." Ultimately this will be a far more enlightening and extensible specification than the first version. Note that behavior is structured along with the specification. The separation of concerns shows that it is awkward and inconsistent to give "call forwarding on busy" precedence over call waiting.

On the positive side, the benefits of the right model are inspiring. For the benefits of a good decomposition, think of protocol layers, the clean separation of processor management from memory management in operating systems, and the well-known decomposition of compilers into lexical analysis, parsing, optimization, and code generation phases. As an example of a powerful notation, think of using a BNF grammar to specify a parser. As examples of productive analysis, think of code-optimization algorithms and model checking for protocols.

Other engineering disciplines, such as civil, chemical, aeronautical, nuclear, electrical, mechanical, and biomedical engineering, are specialized. The purpose of research in these disciplines is to discover how to construct *a particular kind* of useful artifact, more efficiently and more successfully, using particular conceptual and mathematical tools. Once the research has been done, the results can be taught and subsequently used by many engineers to make many similar artifacts.

Abstractions must be chosen so that they illuminate rather than obscure important issues.

Computer science is trying to cover the same ground in the virtual realm that all of these engineering disciplines cover in the physical realm. As a result, it promulgates the fiction that one software system is pretty much the same as another, and abdicates responsibility for discovering and teaching how to build a particular kind of software system really well.

From this perspective, it is notable that the software-modeling success stories mentioned above all come from software that is not an end in itself but serves to make computers easier to use for other purposes. This software is central to computer science, and has been the focus of intense effort by computer scientists. So it is an exception to the general attitude that all software is alike, and the special attention has evidently paid off.

BECAUSE OF THE DIFFICULTY OF THE TASK, and taking into account the example of engineering research, the conclusion is obvious: Finding the best way to use formal methods in an application domain is research, not development. It is an unusual kind of research, although certainly not unheard-of. It is intellectually challenging and rewarding, at least when the standards for results are set high. And it is probably the most effective thing we can do to bring formal methods into widespread use. ▮

Engineering Mathematics

CONSUMABLE MATHEMATICS FOR SOFTWARE ENGINEERS

Michael J. Lutz, *Rochester Institute of Technology*

Software development is an engineering activity. The goals of software development are essentially the same as those of other engineering fields: the creation of useful products and processes in an effective, economical, and timely fashion. What differentiates contemporary engineering from software development is the relative immaturity of the latter's processes and methods.

Formal methods hold great promise for improving the practice and moving the profession along the path toward a full engineering discipline. For this promise to be met, however, formal methods proponents must take into account the role of mathematics in other engineering disciplines. To ignore the engineering culture is to risk continued rejection of this technology as an "academic fad."

The first thing to recognize is that engineers are *not* mathematicians—nor should they be. Within the broad range of engineering concerns, mathematics is simply a tool, albeit an important one, that supports synthesis and analysis. In this regard, calculus and differential equations are the cornerstones of traditional engineering because of their utility, not because of their elegant mathematical foundations. Approaches that ignore the application of formal methods to practical, realistic problems are unlikely to garner much support among engineers or among the practitioners and students who aspire to be software engineers.

Second, engineers rarely (if ever) resort to first principles. Instead, they use tables, charts, and equations that capture the essence of the phenomena of interest. These highly compact forms permit accurate modeling and analysis without explicitly resorting to underlying axioms and theorems. What is more, the mathematics, while abstract, is expressed in terms of the problem at hand.

By way of contrast, formal methods often require detailed knowledge of underlying mathematical theory, at least when used for verification. This is due both to the range of potential applications and the newness of these techniques. What would help immensely is the development of handbooks, where common patterns are described formally. Until the mathematics is made more "consumable" by practitioners, formal analysis is simply too tedious for widespread use (on the order, say, of the limit definition of differentiation).

Finally, engineers are pragmatic about their tools, and mathematics is no exception. Assuming a mathematical technique is applicable to a problem, the overall context determines whether the technique is used at all and, if so, to what level of detail. There are few absolutes with respect to the engineering uses of mathematics.

FOR FORMAL METHODS TO CHANGE THE PRACTICE, we must take this pragmatic attitude into account. It does no good (and immense harm) to promote formality as an either/or proposition. There are various levels of formality, from purely descriptive to deep, deductive analysis, that can be applied at the various life cycle stages. We must be careful to provide a balanced treatment, where current and future practitioners learn the advantages and limitations, as well as the benefits and costs, of greater formality. ▮

***Michael J. Lutz** is a professor of Computer Science at the Rochester Institute of Technology in Rochester, NY. His academic activities are complemented by industrial experience as a software engineer and as manager of a software development group. His interest in formal methods is focused primarily on technology transfer, especially those problems that hinder acceptance of formal modeling as standard industrial practice. E-mail mjl@cs.rit.edu.*

MATHEMATICAL METHODS: WHAT WE NEED AND DON'T NEED

David Lorge Parnas, *McMaster University*

I have long disliked the phrase "formal methods" for two quite different reasons:

- It is an unnecessary phrase. We are discussing the use of mathematics in engineering, which is nothing new. Why should we give it a new name?
- Those who use that phrase seem to take as their model (and source of inspiration) the logicians who try to tell mathematicians how they should work, not the mathematicians and engineers who actually obtain practical results. Hilbert set out to revolutionize mathematics, but succeeded only in building a separate field of mathematics. Formalists set out to revolutionize software development but have succeeded only in forming new research cliques in computer science.

My positions on the real issues are captured by the following assertions.

- Mathematics should be part of the everyday toolset of every working engineer. Software design is engineering. Engineers use mathematics routinely in much of their work. Software designers should do the same.
- Mathematical methods offered to the working software engineer are not very practical and are not much like the mathematics used by engineering. Most, though not all, are theoretically sound but very difficult to use. They are much more difficult to use than the mathematics that has been developed for use in other areas of engineering.
- We have all of the "fundamental models of programming" we will ever need. Good sound, relational, and functional models of programming have been known for decades. Much of the work on "formal methods" is misguided and useless because it continues to search for new foundations although the ground is littered with sound foundations on which nobody has erected a useful edifice.
- We need a lot more work on notation. The notation that is purveyed by most formal methods researchers is cumbersome and hard to read. Even the best notation I know (mine of course) is inadequate. The most solid foundations are notation-free (they are expressed in terms of abstract states), but without a notation they are useless. The models that are expressed in terms of a specific notation are shaky foundations because those notations have limited applicability.
- The distinction between model, description, and specification is missed in most books and examples. People often use the word "specification" when they mean "model." As a result, many proposed "specifications" are cluttered with information that should not be there, while lacking essential requirements information.

- There is an unreasonable focus on proof of correctness. In other areas, engineers rarely prove theorems; instead, they use mathematics to derive important properties of their proposed designs. They also use mathematics to describe their designs in a way that makes mathematical analysis possible. The difference is subtle, but it is the key to practicality. If I want to prove that a circuit is correct, I start with the daunting task of trying to write down everything that I expect of that circuit, including some things that are obviously true but hard to express. If I try to determine key attributes of the circuit, for example the maximum voltage across the terminals of a particular component, or the resonant frequencies, I have a concrete and achievable goal.
- An old Dutch expression—"selling the pelt before the bear has been shot"—perfectly describes the activities of many formal methods advocates. Many spend their time berating practitioners for not applying their method. We all need to disseminate our ideas, but most of our time should be spent applying and improving our methods, not selling them. The best way to sell a mouse trap is to display some trapped mice. Trapping real mice also shows you how a trap can be improved.

We all need to disseminate our ideas, but most of our time should be spent applying and improving our methods, not selling them.

MATHEMATICAL METHODS HAVE been used by "real people" to study "real software," but the notation was not the notation of mathematical logic.[1-2] Mathematical methods will be the key to improved professionalism in software engineering, but they must be rescued from the grip of philosophers who preach sermons about formality. ∎

References

1. D.L. Parnas, "Mathematical Descriptions and Specification of Software," *Proc. IFIP World Congress 1994*, Vol. I, Aug. 1994, pp. 354-359.
2. D.L. Parnas, G.J.K Asmis, and J. Madey, "Assessment of Safety-Critical Software in Nuclear Power Plants," *Nuclear Safety*, Vol. 32, No. 2, April-June 1991, pp. 189-198.

***David Lorge Parnas** holds the NSERC/Bell Industrial Research Chair in Software Engineering in the Department of Electrical and Computer Engineering at McMaster University. He is a Fellow of the Royal Society of Canada and the ACM, and a Senior Member of the IEEE.*

Education

THE NEED FOR EDUCATION IN USEFUL FORMAL LOGIC

David Gries, *Cornell University*

Application of "formal methods" in software development can be interpreted narrowly to mean the use of a computing tool that ensures some property (like correctness) of the software being developed. Examples are a tool that checks consistency of a specification, a tool that checks whether a program satisfies its specification, and a tool that enforces a particular discipline on software development.

More liberally, any informal use of theoretical ideas in the development process can be viewed as an application of formal methods. Examples are the use of mathematical notation for part of a specification, the use of an informal invariant and bound function when developing a loop, and the use of an informal coupling invariant that describes how an abstract type is to be implemented.

Even with the most liberal meaning of "formal methods," it is safe to say that formal methods are rarely used. In fact, most programmers eschew all aspects of formal methods. Why? Because computer science as a whole has not embraced formal methods, even in the very liberal sense of the term. Few computer scientists use formal methods when writing algorithms and programs, although most do write algorithms or programs. Aspects of formal methods do appear in some undergraduate courses, but only the faculty who teach those courses understand and perhaps use the formal methods; the others could care less.

Few introductory programming texts mention formal methods, much less apply them. Texts on data structures, algorithms, databases, operating systems, and compiler writing don't use formal methods in presenting specifications or algorithms. Even courses on theory of computation, which rely heavily on theory to analyze complexity issues, rarely use formal methods in presenting algorithms and arguing about their correctness.

Formal methods are eschewed for several reasons. We concentrate on two of them:

1. Many computer science undergraduates fear mathematics and math notation, and the courses they take don't dispel this fear. Some computer science faculty also fear math, or at least don't see the need for theory of any kind.
2. Formal logic—propositional and predicate calculus—is almost universally viewed not as a useful tool but as an object of study. Since almost all formal methods are founded on formal logic, they too are viewed only with academic interest.

LOGIC UNDERLIES MANY FORMAL METHODS. Let us discuss the second point, in the realm of sequential programs rather than more complicated parallel and distributed programs. And, to make our point most easily, let us restrict attention to specifications of correctness of sequential programs.

A specification of a program typically includes a precondition and a postcondition, which describe the relation among input variables and among output variables. (Alternatively, a single predicate could be used.) These predicates are stated in some form of predicate logic, extended to include operations of the domain under consideration. Hence, writing formal specifications requires facility with some notation that includes predicate calculus.

Moreover, to analyze a specification, to rewrite parts of it, and to formally develop an algorithm from it requires agility in manipulating predicates. Essentially, one must be able to develop formal proofs, to calculate, to abstract and refine. Hence, formal logic underlies many formal methods. It can even be said that "Logic is the glue that binds together methods of reasoning, in all domains." For example, almost every proof uses a technique like case analysis, assuming the antecedent, contradiction, mutual implication, and mathematical induction, and each of these techniques is based on some theorem of propositional or predicate logic. Therefore, a good education in logic will teach how to develop, write, and present proofs—certainly an advantage when using formal methods.

Even when computer science majors learn logic in a discrete math course taught by computer scientists, the idea of "logic as a tool" is lacking.

Many computer science programs require a course on logic, but logic is taught as an object of study instead of as a useful tool. Some computer science majors take a logic course from the philosophy department. Philosophers are more interested in relating logic to human thinking than in using logic. Their courses thus do not prepare students for using logic.

Other computer science majors take a logic course from the math department. Mathematicians and logicians are more interested in analyzing issues like completeness and equivalence than in using logic. Their courses also do not prepare students for using logic.

Even when computer science majors learn logic in a discrete math course taught by computer scientists, the idea of "logic as a tool" is lacking. Logic is usually taught in one to two weeks, the notations and concepts of logic are not used thereafter, and students get little skill in (and no appreciation for) the use of logic.

Typically, these courses teach some variation of natural deduction, or some logic with a minimal set of inference rules and axioms, like Church's P1. Such approaches are doomed to fail in convincing that logic is useful. No one uses natural deduction formally in their work; it is used only to illustrate how informal arguments can be formalized. No one uses P1 in their work; it is used only to study logic. How can useless tools be used to illustrate usefulness?

In short, computer science students learn that logic is only of academic interest. Students who later become computer science faculty pass this attitude on to *their* students. This attitude has prevailed since the beginnings of computer science, and as long as it prevails, formal methods—even in the most liberal sense—won't be adopted.

TEACHING LOGIC AS A TOOL. To show that logic is useful, a different approach has to be developed. To determine a suitable approach, one can investigate the kinds of formal manipulation used in proving things in other fields.

In many fields, to prove that one formula is equal to another, one formula is transformed into the other using a series of "substitution of equals for equals." A variation of this technique is to show that one formula is less than another, by applying substitutions that result in a "smaller" formula. Such calculations are taught in high school. They are used in modern algebra, calculus, and differential equations, to some extent. But such calculations have not been used in logic.

Over the past fifteen years, however, a calculational (or equational) logic has been developed by researchers in the field of formal development of programs. Some who have attended lectures on it call it "seductive logic," because it makes formalism seem so useful. Those who use it swear by it. Those who teach it would never switch back to older kinds of logic. For they find that, when it is taught properly, students gain a skill in formal manipulation, acquire an appreciation of formal proofs and rigor, and begin to lose their fear of math. We surmise that students will also take more readily to formal methods in software development, although we have not had a chance to verify this conjecture.

Thus, our experience is that logic can be taught, to great advantage, as a useful tool that can pave the way for later appreciation and perhaps some adoption of formal methods. However, the goal of teaching logic as a tool is not simply to make formal methods more readily usable. Instead, the goal is to provide students with better mental tools and a more positive attitude, making it easier for them to deal with the problems they will face in the future.

Introducing a course on "logic as a tool" is not easy. Without metrics (which we don't have) by which to measure how much of a difference "logic as a tool" can make, it is difficult to convince people to change. And there is resistance to change, to anything that requires people to think differently (even in the face of dissatisfaction with the current state of affairs).

However, logic as a tool is not a panacea, a universal remedy or cure-all for all the ills of software engineering. It is just one aspect that can help change the attitudes of practitioners toward mathematics, theory, and formal methods. ▮

David Gries *is a professor in the Computer Science Department at Cornell University. His research interests include compiling and programming methodologies. He has received a number of national awards for contributions to education and was a Guggenheim Fellow in 1984-85. E-mail gries@cs.cornell.edu.*