Ten Commandments of Formal Methods ... Ten Years Later

Jonathan P. Bowen
London South Bank University

Michael G. Hinchey
NASA Software Engineering Laboratory

How have the formal methods commandments fared over the past decade? Are they still valid in the current industrial setting, and have attitudes toward formal methods improved? The authors revisit their 10 maxims to answer these questions.

More than a decade ago, in "Ten Commandments of Formal Methods," (*Computer*, Apr. 1995, pp. 56-63), we offered practical guidelines for projects that sought to use formal methods. Over the years, the article, which was based on our knowledge of successful industrial projects,[1] has been widely cited and has generated much positive feedback. However, despite this apparent enthusiasm, formal methods use has not greatly increased, and some of the same attitudes about the infeasibility of adopting them persist.

In 1995, Bertrand Meyer stated that the advancement of software requires a more mathematical approach.[2] Likewise, formal methodists believe that introducing greater rigor will improve the software development process and yield software with better structure, greater maintainability, and fewer errors.[3]

But while many acknowledge the existence of formal methods and their continued application in software engineering,[4] the software engineering community as a whole remains unconvinced of their usefulness. The myths and misconceptions[5,6] that surrounded formal methods when we wrote our original article in large part still abound.

One misconception is the basic justification for formal methods—that they are essential to avoid design flaws because software is bad, unique, and discontinuous, and testing is inadequate. Mike Holloway, a proponent of formal methods at NASA, argues that the justification is far simpler: Software engineers want to be real engineers. Real engineers use mathematics. Formal methods are the mathematics of software engineering. Therefore, software engineers should use formal methods.

Yet even with this elegant simplicity, most projects hold formal methods at arm's length unless they involve the design and maintenance of critical systems.[7] Some formal techniques such as program assertions are reasonably popular, but they represent only a tiny slice of the vast formal methods pie.

Oddly, despite their spotty application, formal methods continue to appear in the trade literature.[8] Apparently, the software engineering community is not willing to abandon formal methods, given the slight increase in formal methods projects,[9] but neither is it willing to embrace them.

Perhaps revisiting our commandments might explain this curious stalemate. Not all our colleagues agreed with our final commandment choices, arguing that some would not stand the test of time. Would a retrospective prove that our colleagues were right?

I. Thou shalt choose an appropriate notation.

> Notations are a frequent complaint ... but the real problem is to understand the meaning and properties of the symbols and how they may and may not be manipulated, and to gain fluency in using them to express new problems, solutions and proofs. Finally, you will cultivate an appreciation of mathematical elegance and style.

Published by the IEEE Computer Society 0018-9162/06/$20.00 © 2006 IEEE

By that time, the symbols will be invisible; you will see straight through them to what they mean. — C.A.R. Hoare

Many blame the use of mathematical notation for formal methods' slow uptake and believe it inhibits industrial application. The common view is that mathematical expressions are beyond normal comprehension. In reality, the mathematics of formal methods is based on notations and concepts that should be familiar to anyone with a computing background, such as set theory and propositional and predicate logics. Of course, customers and end users would need some training and explanation, but the point is that formal methods notations *are* accessible or can be made that way.

But the first commandment addresses a larger issue than user comprehension. "Appropriate" means that the notation has to fit the system it is meant to describe, which can be tricky because some systems are quite large and complex. The more popular notations—B, Calculus of Communicating Systems, Communicating Sequential Processes, and Z, for example—apply to a wide range of systems, but they are not inclusive.

Thus, larger applications often require a combination of languages. Indeed, many argue that no single notation will *ever* address all aspects of a complex system, implying that future systems will require combinations of methods. Process algebras and logics will become particularly important as systems become more sophisticated.

As Table 1 shows, the trend over the past decade seems to support the augmenting of notations. The table gives just a flavor of the myriad hybrid formal methods that have emerged, strongly indicating the acceptance of combining notations to address specific system aspects. We see three categories of these combinations:

Table 1. A sampling of hybrid formal methods since 1995.

Name	Combines	Advantage
CSP-OZ	Z, CSP Combines	Z and CSP
Object Z	Z, object-oriented principles, temporal logic	Adds object orientation to Z
PiOz	Object-Z, πcalculus	Adds πcalculus-style dynamic communication capabilities to Object-Z
Temporal B	B, temporal logic	Adds time to the B method
Timed CSP	CSP, time	Adds time to CSP
TLZ	Z, TLA	Adds temporal aspects plus fairness constraints to Z specification
WSCCS	CCS, probability	Adds probabilistic constraints to CCS specifications
ZCCS	Z, CCS	Combines CCS process algebra and state-based aspects of Z

CCS: Calculus of Communicating Systems; CSP: Communicating Sequential Processes; OZ: Object-Z; TLA: Temporal Logic of Actions; WSCCS: Weighted Synchronous CCS

- *Viewpoints.* In this loose coupling, different notations present different system views, with each notation emphasizing a particular system aspect, such as timing constraints.
- *Method integration.* In a closer coupling, several notations (both formal and informal or semiformal) combine with manual or automatic translation between notations. The idea is to provide an underlying semantics for the less formal notations, to enable well-understood graphical (or other) presentations, and to offer the benefits of formal verification.
- *Integrated methods.* In a tight coupling, multiple notations combine within a single framework (such as propositional logic) to give a uniform semantics to each notation.

A decade ago, method integration was hot, and it seemed that integrated methods would become equally popular. Although we see progress in integrated methods,[10] the viewpoints approach is the only one that seems to have gained ground. Perhaps this is because of industry's reluctance to take up full formal proofs, which the more tightly coupled approaches would support. But it could also be its general unwillingness to become preoccupied with semantic details.

This unwillingness underlines another misconception—in reality, an appropriate notation can hide unnecessary detail and complexity, and this is a major benefit of formal methods, not a liability. Developers are not only free to concentrate on the essential issues, but they also gain a richer understanding of the system to be developed.

Because formal specifications will often be significantly shorter than their implementation, they are likely to be more understandable. Some argue that a formal specification *must* be significantly shorter, but we disagree. The use of formal methods and formal specification techniques can highlight problems or issues that developers might not see at the coding level. In this case, even a longer formal specification is valuable.

II. Thou shalt formalize but not overformalize.

Strange as it seems, no amount of learning can cure stupidity, and formal education positively fortifies it. — Stephen Vizinczey

Table 2. Formalization levels.

Level	Name	Involves
0	Formal specification	Using formal notation to specify requirements only; no analysis or proof
1	Formal development/verification	Proving properties and applying refinement calculus
2	Machine-checked proofs	Using a theorem prover or checker to prove consistency and integrity

In our original article, we advised projects to distinguish between using formal methods appropriately and formalization just for the sake of it. In some areas, such as user interface design, projects *could* apply formal methods, but doing so might not be the best choice.

In fact, a prominent myth (and one we listed in "Seven More Myths of Formal Methods,"[5]) is that formal methods people always use formal methods. In reality, many highly publicized projects proclaimed as great formal methods successes formalized only 10 percent or less of the system.

Ten years ago, we noted the dearth of toolsets for most formal methods. Not much has changed, although Perfect Developer by Escher Technologies (www.eschertech.com/products/) and Atelier-B from ClearSy (www.atelierb.societe.com/contact_en.htm) are attempts to develop such tools.

Escher Technologies has even partially applied Perfect Developer to the tool's own redevelopment (for all but the graphical user interface), proving around 95 percent of the approximately 130,000 verification conditions the tool generated. For development of simpler systems, it has been used to achieve 100 percent proof checking of the verification conditions. Mistakes are often found to be caused by underspecification in practice. The Spark toolset from Praxis High Integrity Systems (www.praxis-his.com/sparkada/) is another example of applying an industrial formal methods tool to itself.

The formal methods community seems to have taken the warning not to overformalize somewhat to heart, and there is now more widespread belief that it's best to use formal methods as needed, mainly for key product parts. Cliff Jones introduced "formal methods light," which approximates Level 0 of the three formalization levels in Table 2 (taken from our 1995 article).

Even Level 0 formality can accrue many benefits because the importance of getting requirements right at the outset cannot be overstated. Figure 1 shows a graph of investment in the requirements phase of NASA projects and missions plotted against the cost of project overruns. The obvious "demand curve" emphasizes that getting requirements right has major payback later—or, conversely, that not getting requirements right will come back to haunt you.

The use of mathematically based approaches has great potential to help eliminate errors early in the design process. It is cheaper than trying to remove them in the testing phase or, worse, after deployment. Consequently, it is true that using formal methods in the initial stages of the development process can help to improve the quality of the later software, even if formal methods are not used in subsequent phases of development.

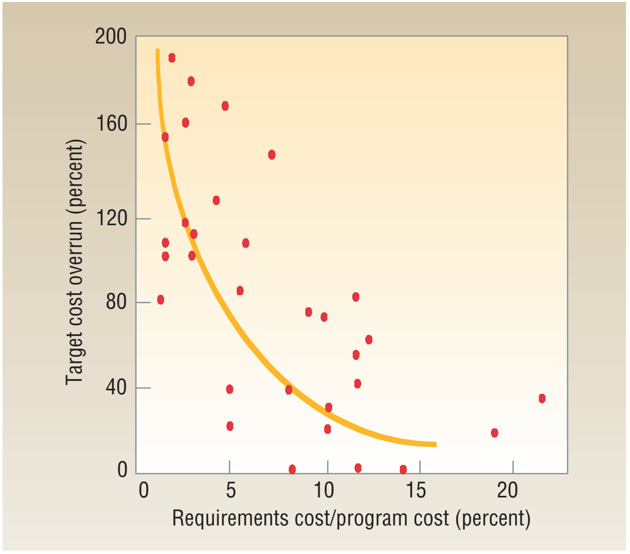

Figure 1. Costs during the requirements phase of NASA projects versus project overrun costs. The curve shows the savings of getting requirements right and the price of getting them wrong. Courtesy of W. Gruhl, NASA Comptroller's office.

III. Thou shalt estimate costs.

I think that God in creating Man somewhat overestimated his ability. — Oscar Wilde

When asked what they'd charge a customer for a software project, software engineers often joke, "As much as we can possibly get away with." Although that's meant to be humor, it reflects a certain mind-set that carries over into estimating development costs, where the strategy is often to make the best (usually highest) estimate and then double it.

In the draft of our 1995 article, we had "guesstimate costs" instead of "estimate," a term we liked because a hybrid of "guess" and "estimate," more closely captures the imprecision of the exercise. (It did not survive the more precise art of copyediting, however.) Even with several established models, among them Cocomo II, cost estimation is far from a science. Development costs sometimes grandly exceed estimates: The Darlington power plant and Space Shuttle software had cost overruns that were significantly more than anyone could have foreseen. It was for that reason that we strongly advocated both initial and continuous cost estimation—and we still do.

Research shows that organizations spend 33 percent to 50 percent of their total cost of ownership (TCO) preparing for or recovering from failures.[11] Hardware costs continue to fall, yet TCO continues to rise, and system availability (and hence reliability) is taking a hit. In this light, any cost estimates could be unrealistic, understated, or even unrealistically understated.

However, we still firmly believe in having a cost estimate as well as some idea of anticipated costs if a team elects to forego formal methods. A cost estimate is essential for convincing the development communities—both software and hardware—that formal methods can indeed produce better systems for less.

IV. Thou shalt have a formal methods guru on call.

An expert is a person who has made all the mistakes that can be made in a very narrow field. — Niels Bohr

Access to an expert outside the team is not enough to ensure success.

Part of what we found in our initial research is that most successful projects had regular access to a formal methods expert. Many had several gurus to guide and lead the formal development process and advise on complex aspects. Occasionally, such experts were able to compensate for the development team's lack of experience in applying formal methods.

But access to an expert outside the team is not enough to ensure success. All team members must understand the applicability of formal methods and contribute to rather than inhibit their application. It is too easy for team members, on either the management or technical side, to prevent effective formalization.

Formal methods require the right mix of effort, expertise, and knowledge. Although not every team member needs the same formalization proficiency, at the very least, all must appreciate what formal methods can achieve.

A formally verified program is only as good as its specification. If the specification does not describe what the team truly wants, even a fully formally developed system will be little more than useless. A team that doesn't understand formal methods has only a notion of what they specified using a formal notation and is unclear about how to refine the development process will almost certainly sink the project. Perhaps this is why some quarters are skeptical about the benefits of formal methods.

So we stand by this commandment, although if we were writing the article today, we might tweak it a bit to read, "Thou shalt have both a formal methods guru and a domain expert from the outset." Our experience with industrial projects over the past decade has highlighted the importance of having both kinds of experts early on.[3]

V. Thou shalt not abandon thy traditional development methods.

A great many of those who 'debunk' traditional ... values have in the background values of their own which they believe to be immune from the debunking process. — C.S. Lewis

The software engineering community persists in embracing fads. Each new notation or technique seems to have the unwritten guarantee of painless success. This is a dangerous mind-set, particularly when the notational flavor of the month becomes an additional source of problems, not a magical solution. The Unified Modeling Language, which has become ubiquitous in industrial applications over the past decade, is a case in point. UML has some serious flaws, such as its lack of formality and scant guidance on applying the newer graphical notations.

Fortunately, the UML community has recognized the need to address the first flaw. Formal methods research has spent some time considering formalization in the context of UML, which has led to the formation of the precise UML (pUML) group. There is also work at the University of Southampton on the tool-based integration of the B-Method, a formal approach, and UML. Such improvements are likely to show up in future UML developments.

Another caveat to using UML is that it essentially standardizes several existing and emerging graphical notations for system specification. Many of these notations have been around since the 1970s, with only slight variations in their representation, but a wide variety of new graphical notations have recently joined the list. Some of these are there for good reason; others, because they had support from particular quarters. Unfortunately, UML tends to deemphasize the particular development method, so although it provides a range of notations, it gives no guidance for what notations fit best with which system types, which notations conflict when combined, and which notations are good complements.

To be fair, most formal methods and most formal approaches to software or hardware development also fail to address development's methodological aspects. Because they have a specification notation and a reasoning mechanism, formal methods are truly formal. However, they are not truly methodical because they don't offer defined ordered steps and guidance for moving between them. Recent formal approaches like the B-Method have addressed this issue to some extent.

Object-oriented techniques are also popular, and research has produced OO extensions to formalisms, such as Object-Z for the Z notation. Formal methods tools, such as Perfect Developer, also target OO development. Software engineers who develop systems with languages such as Java might find such a tool attractive.

Other research at NASA Goddard Space Flight Center[12] is addressing how to increase formality in model-based development and in requirements-based programming. The latter approach aims to transform requirements into executable code systematically and has many of automatic programming's advantages, while avoiding its major deficiency of specifying a solution rather than the problem to be solved.

Someone must record the reasons for various specification, design, and decomposition decisions.

VI. Thou shalt document sufficiently.

I have always tried to hide my own efforts and wished my works to have the lightness and joyousness of a springtime which never lets anyone suspect the labours it cost. — Henri Matisse

Matisse was a master of abstraction. While most artists prepared rough preliminary drawings for their works and then added detail, Matisse took the opposite approach, making his preliminary drawings extremely detailed. After he had finished working, he would have his assistant photograph what he had done so that he had a record of his decisions and the work he had completed. The next morning he would destroy the work, undoing most (sometimes all) of what he had added the previous day. Consequently, Matisse's final works are often highly abstract, with few lines, but all of what's there is essential to the representation. Perhaps the most compelling example of this is the 1935 edition of James Joyce's *Ulysses*, which Matisse illustrated without even having read it (using Homer's *Odyssey* as a basis instead).

In an attempt to combine abstract documentation with concrete programs, Donald Knuth introduced the idea of *literate programming*. Using this style, programmers connect code fragments to relevant documentation in a way that justifies coding (and hence design) decisions. Literate programming would seem to be an excellent fit with the use of formal methods, since it could also associate code with the relevant formal specification fragments, as well as the requirements that drive those fragments. However, industry did not act on that association. Instead, attempts to build literate programming tools led to the development of extreme programming (XP), which provides little documentation and emphasizes product development and frequent releases.

Formal methods demand quality documentation, some of which can be automated, but someone must fully explain formal specifications so that they are understandable to both nonspecialists and those working on the specification after its initial development. Someone must also record the reasons for various specification, design, and decomposition decisions as a courtesy to future developers.

In addition to the benefits of abstraction, clarification, and disambiguation, which accrue from the use of formal methods at Level 0 in Table 2, using formal methods at the formal specification level provides invaluable documentation. Experience has shown that quality documentation can greatly assist future system maintenance. In fact, several collaborative European projects have involved the documentation of legacy systems or reverse engineering.

All development involves iteration, and documentation must reflect that. Often, when engineers change the system implementation, they neither record that change nor update the related documentation. True formal development would use formal methods to help avoid such inconsistencies since the formal specification is part of the documentation.

Properly documenting decisions during the formal specification process is also important, which is why we have always advocated augmenting formal specifications with natural language narrative. A proper paper trail is critical. Without it, the organization loses the benefits of abstraction and might even lose useful information.

VII. Thou shalt not compromise thy quality standards.

If people knew how hard I worked to get my mastery, it wouldn't seem so wonderful at all. — Michelangelo Buonarroti

According to the National Institute of Standards & Technology, 2002 losses from poor software quality amounted to more than $60 billion (www.mel.nist.gov/msid/sima/sw_testing_rpt.pdf). Software quality is still a huge issue that no one has yet addressed adequately. The ISO 9000 quality standards have been in force since 1994, and ISO even revised them in 2000, yet poor software quality still plagues users. Standards could be crucial in changing this destructive trend.

Standards are also critical in high-integrity areas like safety- and security-critical applications. For example, the IEC 61508-3 International Standard on Software Requirements for Safety-Related Systems covers software design, development, and verification. Obviously, formal methods can be part of this process, but most standards merely suggest that a project could use such methods—they don't mandate use. The onus is on the developer to demonstrate that using formal methods makes sense and is worthwhile.

Safety and security standards continue to drive formal methods use at the highest levels of integrity, and this trend is likely to continue. In the UK, for example, the two-part Defence Standard 00-55 from the Ministry of Defence, which regulates defense contracts, has a mandate in the "Requirements" section of part 1 (italics are ours): "Assurance that the required safety integrity has been achieved is provided *by the use of formal methods* in conjunction with dynamic testing and static analysis." The standard also mandates formal methods use for safety-related software: "The methods used in the SRS development process shall include …: a) *formal methods of software specification and design*; …" Finally, the "Guidance" section in part 2 mentions formal methods in many places and includes an explicit section under "Required Methods."

Even standards that mandate formal methods use are not enough to ensure quality.

However, even standards that mandate formal methods use are not enough to ensure quality. Formal methods practitioners must also adhere to quality standards in the development processes—not only standards for various specification notations (such as Z), but also standards that reflect best practice in software development. Following such standards is the best way to ensure correctness, regardless of whether someone deems that software critical. Formal methods are meant to complement existing quality standards, not supplant them.

Standards documentation itself can use formality, as does the documentation for Prolog, and even formal notations can have associated standards—there are ISO standards for LOTOS, VDM, and Z, for example. ISO approved the Z standard in 2002 after nearly a decade of production. Progress was slow and painstaking in part because much effort centered on formalizing a revised version of Z notation. On the other hand, the process did reveal some semantic inconsistencies, so at least in that context it was a success. Regardless of viewpoint, there are lessons for any future efforts to produce a formal method standard.

VIII. Thou shalt not be dogmatic.

… And I am unanimous in that! — Mollie Sugden, a.k.a. Mrs. Slocombe, in "Are You Being Served?" BBC TV (1972–1985)

Perhaps one of the worst misconceptions about formal methods is that they can guarantee correctness.[5] They can certainly offer greater confidence that an organization has correctly developed the software or hardware, but that's all. In fact, it is absurd to speak of correctness without referring to the system specification.[5] If the organization has not built the right system (validation), no amount of building the system right (verification) can overcome that error. In an investigation of failed safety-critical systems, one study found nearly 1,100 deaths attributable to computer error.[13] Many of these errors stemmed from poor or no specifications, not an incorrect implementation.

The danger for many projects is the analysis-specification gap—the space between what is in the procurer's mind (real-world entities) and the writing of the specification (notation software professionals choose, either formal or informal). Formal methods—with only a few exceptions—offer very little or no methodological support to close this gap.

The solution for some is to use less-formal methods or formal methods augmented with methods that offer greater development support. The argument is that such adaptations would be more intuitive to users.

Model-based development aims to address this by placing great emphasis on getting an appropriate model of reality. Likewise, requirements-based programming is attempting to fully integrate requirements in the development process. Both these approaches reduce the analysis-specification gap by ensuring that what is specified (and ultimately implemented) is a true reflection of real-world requirements.

IX. Thou shalt test, test, and test again.

I believe the hard part of building software to be the specification, design and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation. — Frederick P. Brooks Jr.

Largely because of formal methods research in the 1960s (before the community had even coined the term), most programs include assertions. The intent of assertions was to prove programs correct, and, at that time, most people believed this was *all* that formal methods were supposed to do.[5] Now, testers use assertions to check if a program's state is correct during runtime. Promising research, centered on the Java Modeling Language, is attempting to broaden the use of assertions to include formal verification as well.

Perhaps some day, a verifying compiler, such as the one Tony Hoare proposed, will be able to verify assertions at compile-time rather than at runtime, eliminating the need to use assertions in testing. A current computer science Grand Challenge proposes the development of such a compiler over the long term.

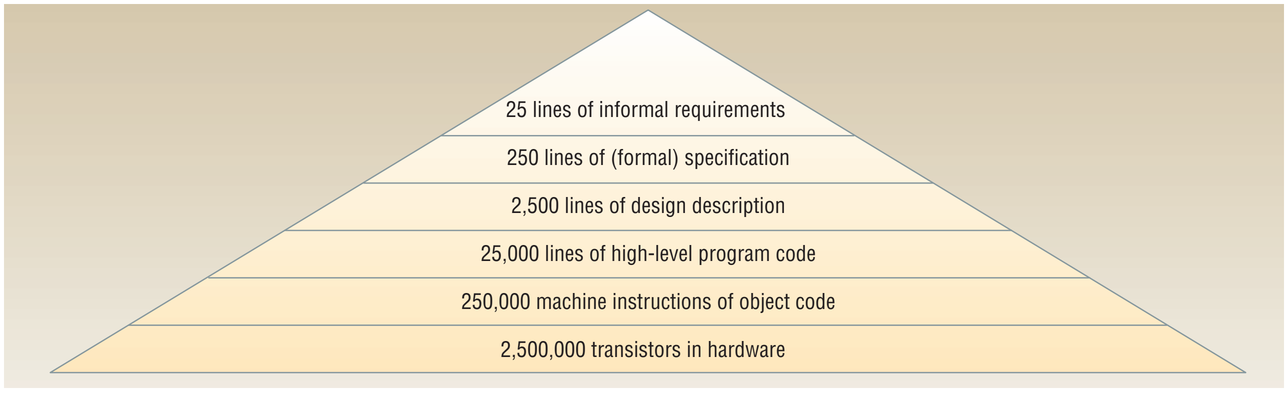
25 lines of informal requirements
250 lines of (formal) specification
2,500 lines of design description
25,000 lines of high-level program code
250,000 machine instructions of object code
2,500,000 transistors in hardware

Figure 2. The size explosion as development progresses (numbers are hypothetical).

For the near term, the use of formal methods to improve testing has much potential. A formal specification can aid automatic test-case generation, but the time required to produce a formal specification could be far greater than the time saved at the testing stage. In the UK, researchers are using the Fortest (Formal Methods and Testing) network as a framework (www.fortest.org.uk) to investigate the tradeoffs.

Formal methods also have potential use in clarifying test criteria. The MC/DC (Modified Condition/Decision Coverage) is a criterion in many safety-related applications and standards recommendations, such as the RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Certification. The criterion is normally defined informally, but the Centre for Applied Formal Methods at London South Bank University has investigated its meaning formally using Z notation and has developed an even stricter criterion.

Although we see formal methods making some inroads into software testing, application is challenging because software is unique in many ways:

- Even very short programs can be complex and difficult to understand.
- Software does not deteriorate with age. In fact, it improves over time because engineers discover and correct latent errors, but the same error correction can introduce defects.
- Changes in software that appear to be inconsequential can result in significant and unexpected problems in seemingly unrelated parts of the code.
- Unlike hardware, software cannot give forewarnings of failure. Many latent errors in software might not be visible until long after the organization has deployed the software.
- Software lends itself to quick and easy changes.

The last characteristic does not translate into quick and easy error location and correction. Rather, organizations must use a structured, well-documented development approach to ensure comprehensive validation. We would never claim that formal methods can or even should eliminate testing. Quite the contrary: The use of formal methods can *reduce* the likelihood of certain errors or help detect them, but formal methods must partner with appropriate testing.

X. Thou shalt reuse.

The biggest difference between time and space is that you can't reuse time. — Merrick Furst

Traditionally, organizations have encouraged reuse as a way to reduce costs and boost quality. The idea is to then spend more time improving the quality of components targeted for reuse. Both OO and component-based paradigms exploit the idea of reuse.

Theoretically, formal methods can and should aid in promoting software reuse. One inhibitor to the uptake of software reuse is the inability to identify suitable components in a library and to develop libraries of components that are large enough to give a reasonable return, yet small enough to be broadly reusable.

For some time, practitioners have recognized that they can make searching more effective by having formal specifications of components or at the very least of their pre- and postconditions. (Preconditions specify when to apply the component; postconditions describe the results of using it.) Supplying such conditions lets the component remain a black box, which in turn means that the component is much larger and therefore could have a more significant payoff in reuse.

There are significant returns in applying reuse at the formal specification level. Formal specifications are typically shorter than the equivalent implementation in a programming language. Figure 2 provides a comparison of the potential size explosion as development proceeds from specification to hardware implementation. It is obviously easier to search for larger components, while simultaneously getting a sufficient return. Along the same lines, formal specifications could help identify reusable design patterns.

Another way formal specifications can support reuse is in generating implementations on various platforms. This approach essentially reuses the effort expended at earlier development stages and thereby reduces overall cost. The literature reports the successful application of formal specification techniques to developing software product lines—systems (or products) that have only slight variations. Moreover, formal methods generally result in a cleaner architecture, making a system more efficient and more easily maintainable.

Reusing and porting software is not without pitfalls, however. Ariane 5 is a prime example. Its developers assumed that they could reuse the launch software from Ariane 4. Their assumption resulted in a rocket loss within seconds of launch.

The Therac-25 incidents are arguably the most significant failure of software assurance in a medical or biological application. Therac-25 was a dual-mode linear accelerator that could deliver either photons at 25 MeV or electrons at various energy levels. It was based on Therac-20, which in turn was based on the single-mode Therac-6. The Therac-20 included hardware interlocks for safety, but in Therac-25 these interlocks were software-based. Despite several Therac-25 machines operating, reportedly correctly, for up to four years at various US installations, in six separate incidents the device administered lethal doses of radiation to patients.

Subsequent investigations of both Therac-20 and Therac-25 revealed a software error that caused the machines to act erratically. Students at a radiology school had creatively set parameters that caused the Therac-20 machines to shut down after blowing fuses and breakers. The failures were bothersome, but certainly not life-threatening. However, when the same error perpetuated to Therac-25, which did not have mechanical interlocks, the problem became fatal. If the developers of Therac-25 had fully checked the software using formal methods, possibly, they might have realized the significance of this error.

Ten years later, we are surprised to find that the original formal methods commandments are still valid. The use of formal methods is not as prevalent as we had hoped, but we are more certain that formal approaches will always have a niche in computer-based systems development, especially when correct functioning is critical.[14] As the "Looking Ahead" sidebar describes, the next 10 years should see some significant progress in integrating formal methods and traditional development practices.

Like any approach, formal methods work best when applied judiciously. It makes the most sense to use them for the software that performs critical operations, but any application should be part of sound engineering judgment that considers both technical feasibility and economics. For such efforts, well-trained personnel of the highest quality will always be needed.

The rewards can be considerable with the right combination of knowledge and expertise, but formal methods are not a panacea. Some, especially those in academia, have oversold formalism's ability. Given that people must apply formal methods, they will never be completely reliable. The logical models must relate to the real world in an informal leap of faith both at the high-level requirements or specification end and at the low-level digital hardware end (which requires belief in Maxwell's equations, for example).

More effort must be devoted to evaluating the effectiveness of formal methods in software development and maintenance. Hopefully, we have raised issues that others will find worth exploring. Because of the somewhat tarnished reputation of formal methods, largely due to

Looking Ahead

Industrial-strength tools for formal methods have always been lacking. A few exist—notably, Atelier-B and Perfect Developer—but the demand for a range of compatible tools is growing. In the next 10 years, tool support for formal methods will become critical. Some collaborative efforts that are headed in this direction include

- CZT Community Z Tools initiative (czt.sourceforge.net);
- European Rodin Project, an open development environment for complex systems based on B# (/rodin-b-sharp.sourceforge.net);
- development of the B-Method, which includes a free version (www.b4free.com); and
- Higher Order Logic (HOL) 4 theorem prover (hol.sourceforge.net).

Hopefully, such tool advances will make formal methods easier to justify and use in an industrial context.

Online documentation is also becoming increasingly important. The Virtual Library formal methods Web site (vl.fmnet.info), established more than a decade ago, continues to be a central resource for formal methods information. More recently, Wikipedia, an online encyclopedia, has included increasingly useful information on formal methods and related topics. This could be the path for a repository of collaboratively maintained online information. Another path is the effort of the UK Grand Challenge 6 Committee on Dependable Software Evolution, which is planning a verified software repository through the recently funded VSR-net network. Those who want real software to challenge their tools can deposit examples of formally verified software and associated tools for general use.

misunderstandings and inappropriate use, a demonstration of how and where formal methods are effective would be well worth the effort.

There are continuing success stories in the industrial use of formal methods,[7] and the approach remains in the eye of the press.[8] Studies will help practitioners understand how to ensure that the introduction of formal methods has a positive impact on the software development and maintenance process by reducing overall costs.

Above all, formal methodists must have patience. Sculptor Théophile Gautier once said, "L'ouvre sort plus belle, d'une forme au travail rebelled vers," which translates roughly to "The work is more beautiful from a material that resists the process." If that is true, then formal methods use will eventually emerge in near-perfect form. ■

Acknowledgments

We are grateful to our many colleagues and friends who provided us with valuable feedback and reactions to our original article. We also acknowledge the contributions of the formal methods community as a whole and thank them for providing us with material on which to base the original commandments. In particular, we thank David Atkinson, Jin Son Dong, Cliff Jones, Tiziana Margaria, Jim Rash, Chris Rouff, Roy Sterritt, and Bernhard Steffen for their input.

Special thanks go to Tiziana Margaria and Mieke Massink, co-chairs of FMICS 2005, and George Eleftherakis, chair of SEEFM 2005, for inviting earlier conference presentations of this material, in the former case rather aptly to coincide with the 10th anniversary of FMICS. Thanks also to Scott Hamilton for encouraging us to revisit the commandments 10 years after the original paper.

References

1. M.G. Hinchey and J.P. Bowen, eds., *Applications of Formal Methods*, Prentice Hall Int'l Series in Computer Science, 1995.
2. D. Power et al., "Where Is Software Headed? A Virtual Roundtable," *Computer*, Aug. 1995, pp. 20-32.
3. M.G. Hinchey, "Confessions of a Formal Methodist," *Proc. 7th Australian Workshop on Safety-Critical Systems and Software (SCS 02)*, P. Lindsay, ed., Conferences in Research and Practice in Information Technology, vol. 15, Australian Computer Soc., 2002, pp.17-20.
4. K.-K. Lau and R. Banach, eds., *Formal Methods and Software Eng.*, LNCS 3785, Springer-Verlag, 2005.
5. J.P. Bowen and M.G. Hinchey, "Seven More Myths of Formal Methods," *IEEE Software*, July/Aug. 1995, pp. 34-41.
6. J.A. Hall, "Seven Myths of Formal Methods," *IEEE Software*, Sept./Oct. 1990, pp. 11-19.
7. P.E. Ross, "The Exterminators," *IEEE Spectrum*, Sept. 2005, pp. 36-41.
8. R. Sharpe, "Formal Methods Start to Add Up Again," *Computing*, Jan. 2004; www.computing.co.uk/features/1151896.
9. M.G. Hinchey and J.P. Bowen, eds., *Industrial-Strength Formal Methods in Practice*, FACIT Series, Springer-Verlag, 1999.
10. J.M.T. Romijn, G.P. Smith, and J.C. van de Pol, eds., *Integrated Formal Methods*, LNCS 3771, Springer-Verlag, 2005.
11. D.A. Patterson et al., *Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies*, tech. report, UCB//CSD-02-1175, Computer Science Dept., Univ. of Calif. Berkeley, 2002.
12. M.G. Hinchey, J.L. Rash, and C.A. Rouff, "Requirements to Design to Code: Towards a Fully Formal Approach to Automatic Code Generation," NASA tech. monograph TM-2005-212774, NASA Goddard Space Flight Center, 2005.
13. D. MacKenzie, *Mechanizing Proof: Computing, Risk, and Trust*, MIT Press, 2001.
14. J.P. Bowen and M.G. Hinchey, "Ten Commandments Revisited: A Ten-Year Perspective on the Industrial Application of Formal Methods," *Proc. 10th Workshop on Formal Methods for Industrial Critical Systems (FMICS 2005)*, ACM Press, 2005, pp. 8-16.

***Jonathan P. Bowen** is a professor of computing at London South Bank University's Institute for Computing Research. His research interests include software engineering in general and formal methods in particular. Bowen received an MA in engineering science from Oxford University. He is a member of the ACM and the IEEE and a Fellow of the British Computer Society and the Royal Society of Arts. Contact him at www.jpbowen.com or jonathan.bowen@lsbu.ac.uk.*

***Michael G. Hinchey** is director of the NASA Software Engineering Laboratory at NASA Goddard Space Flight Center and affiliate professor at Loyola College in Maryland. His research interests include software engineering in general and formal methods, particularly their application to complex autonomous, autonomic, and biologically inspired systems. Hinchey received a PhD in computer science from the University of Cambridge. He is a Fellow of the British Computer Society, the IEE, the Institute of Engineers of Australia, and the Institute of Mathematics and Its Applications and a senior member of the IEEE. Contact him at sel.gsfc.nasa.gov or michael.g.hinchey@nasa.gov.*