Ten Commandments Ten Years On: An Assessment of Formal Methods Usage

Jonathan P. Bowen
London South Bank University
Institute for Computing Research, Faculty of BCIM
Borough Road, London SE1 0AA, UK
www.jpbowen.com
jonathan.bowen@lsbu.ac.uk

Michael G. Hinchey
NASA Software Engineering Laboratory
Goddard Space Flight Center
Greenbelt, MD, USA
sel.gsfc.nasa.gov
Michael.G.Hinchey@nasa.gov

Abstract. The application of formal methods has been slower than hoped by many in the formal methods community. Over the last decade, there have been some advances and important developments. Despite this, formal methods remain a specialist area, but have an increasingly important niche in the development of critical systems, especially where safety and security are involved. In this paper we examine the use of formal methods within the framework of ten "commandments" (rules for guidance) that were originally published in 1995. These remain remarkably relevant even ten years later, but this paper provides an update with the decade of experience that has been gained since then.

1 Background

> *He proclaimed to you his covenant, which he commanded you to keep: the Ten Commandments, which he wrote on two tablets of stone.*
>
> — Deut.4:13, 10:4, Ex.34:28

Ten years ago, we were pleased to have our paper *Ten Commandments of Formal Methods* [1] promptly accepted for publication in *IEEE Computer*. Although written in a humorous style, it conveyed some serious practical guidelines to help ensure the success of a formal methods project. These guidelines were based, primarily, on our knowledge of several successful industrial projects, many of which were presented in [8] and [9].

J.P. Bowen and M.G. Hinchey, "Ten Commandments Ten Years On: An Assessment of Formal Methods Usage",
Proceedings of the 2nd South-East European Workshop on Formal Methods (SEEFM05),
Ohrid, 18-19 Nov 2005, pp. 1-16.

A decade on, we are surprised at just how popular our paper was. It has been translated into other languages, reprinted in books (such as [3]), widely cited, and "abstracted" in several software engineering textbooks. For example, it is in the "top ten" when searching under "formal methods" on publications databases such as *IEEE Xplore* (`http://ieeexplore.ieee.org`), and Google scholar (`http://scholar.google.com`) lists it as having over a hundred citations. It is also widely used in undergraduate and graduate software engineering and formal methods courses, from which we have received feedback that it provides useful background material for students. In short, it has become a very popular "non-technical" reference for those entering the field of formal methods.

However, not all of our colleagues have agreed with our choice for the Ten Commandments, feeling that some of them would not stand the test of time and would become invalid. In the light of a further ten years of industrial best practice, here we will reconsider the *Ten Commandments of Formal Methods* ten years later.

2 Introduction

> *It is clear to the best minds in the field that a more mathematical approach is needed for software to advance much.*
>
> — Bertrand Meyer

It is clear to "Formal Methodists" like ourselves, for some of whom formal methods can be something of a "religion", that introducing greater rigor into software development will improve the software development process, and result in software that is better structured, more maintainable, and with fewer errors [10]. While many now acknowledge the existence of formal methods (and they are covered in most good software engineering textbooks), the software engineering community as a whole has not been convinced of the usefulness of formal methods. Many myths and misconceptions (such as those reported in [2] and [6]) still exist.

Mike Holloway points out that the typical argument in favour of formal methods (that software is bad, unique, and discontinuous; that testing is inadequate; and that formal methods are essential to avoid design flaws) is logically flawed, and unnecessarily complex (in logical terms) [4]. He proposes a simpler argument, which is both simple and logically valid: software engineers want to be "real" engineers; real engineers use mathematics; and since formal methods is the mathematics of software engineering, software engineers *should* use formal methods.

Nevertheless, while some formal methods techniques (such as assertions in programs) are routinely used, formal methods and their associated techniques have still not been widely taken up in industry. Despite this, formal methods continue to be used in some parts of industry [11] and to maintain a profile in the press [12].

> *Any intelligent fool can make things bigger and more complex …*
> *It takes a touch of genius and a lot of courage to move in the opposite direction.*
>
> — Albert Einstein (1879–1955)

3 Revisiting the Ten Commandments

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. By that time, the symbols will be invisible; you will see straight through them to what they mean. The great advantage of mathematics is that the rules are simpler than those of natural language, and the vocabulary is much smaller. Consequently, when presented with something unfamiliar it is possible to work out a solution for yourself, by logical deduction and invention rather than by consulting books or experts.*
>
> — C.A.R. Hoare

The use of mathematical notation is often cited as a reason for the slow uptake of formal methods and an inhibitor to their successful utilization in industrial applications. People seem to have a reluctance to employ such notations, believing mathematical expressions to be beyond their comprehension. However, we all routinely use mathematical symbols as we employ arithmetic and algebra in our daily lives. The mathematics of formal methods is actually relatively simple, being based on notations and concepts that *should* be familiar to anyone with a background in computing (set theory, propositional and predicate logics, etc.).

We agree that it may be unreasonable to expect customers and end users to be familiar with such notations without at least some training and explanation, which may complicate the issue of interaction with those people whose involvement is vitally important to the success of a project. Nevertheless, with appropriate training, the notations of formal methods are certainly accessible to software engineers.

However, the original intent of our first commandment was not to deal with comprehension of notations by users, but rather "appropriate" in that it was useful in describing the system at hand in a manner that fits well with that system. While several of the more popular notations (e.g., B, CCS, CSP, Z, etc.) have emerged and have widespread applicability to a broad range of systems, it has been found in many cases that a combination of languages is needed to address all aspects of a larger application adequately. It has often been argued that no single notation will ever be suitable to address all aspects of a complex system, with the implication that future systems will require combinations of model-based methods, process algebras, and temporal (and other) logics, in particular as we build more sophisticated, advanced, and ambitious systems.

Table 1 illustrates just a small selection of the wide range of "hybrid" formal methods that have emerged over the last ten years, since our original paper was published, indicating a need to augment existing notations with concepts that address specific aspects of a system. These vary in the means by which they are combined, which we categorize as follows:[1]

[1] Our terminology and classification may differ from those of other authors.

- *Viewpoints (loosely coupled*): different notations are used to present different "views" of a system with each notation making emphasis of a particular aspect of the system (e.g., representing timing constraints).
- *Method Integration (close coupling):* several different notations (both formal and informal or semi-formal) are used with (manual or automatic) translation between notations being used to provide both an underlying semantics for the less formal notations, and to enable both graphical (or other) presentations that are well-understood, while simultaneously affording the benefits of formal verification.
- *Integrated Methods (tight coupling):* multiple notations are used along with a single notation (e.g., propositional logic) used to give a uniform semantics to each notation.

Name	Combines	Advantage
TLZ	Z, TLA	Adds temporal aspects plus fairness constraints to Z specification
WSCCS	CCS, probability	Adds probabilistic constraints to CCS specifications
Object Z	Z, OO principles temporal logic	Adds OO to Z
ZCCS	Z, CCS	Combines CCS process algebra and state based aspects of Z
Timed CSP	CSP, time	Adds time to CSP
CSP OZ	Z, CSP	Combines Z and CSP
Temporal B	B, temporal logic	Adds time to the B Method
PiOZ	Object-Z, π-calculus	Adds π-calculus style dynamic communication capabilities to Object-Z

Table 1. Some hybrid formal methods (since 1995)

When we published our original *Ten Commandments*, Method Integration was very popular and it seemed that there would be a greater move towards Integrated Methods. While certainly there has been more progress in these fields, it seems that a Viewpoints approach has been gaining ground, perhaps due to reluctance by industry to take up full formal proof (that the more coupled approaches would support) and unwillingness to become preoccupied with semantic details.

The choice of an appropriate notation can greatly aid in *abstraction*, to hide unnecessary detail and complexity. Many argue that this is a major benefit of formal methods, enabling concentration on the essential issues and affording a better understanding of the system to be developed. It is also argued that formal specifications will often be significantly shorter than their implementations (cf. Figure 2 and our discussion under Commandment X), and therefore easier to understand. Some argue that unless a formal specification is significantly shorter than its implementation then it is of reduced benefit. However, the use of formal methods and formal specification techniques can highlight problems or issues that might

not be noted at the coding level. In this case, the formal specification is very valuable whatever its length.

If I could say it in words there would be no reason to paint.

— Edward Hopper (1882–1967)

II. Thou shalt formalize but not over-formalize.

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

— Stephen Vizinczey

In our original *Ten Commandments* paper, we advocated the need to distinguish between formalization "for the sake of it", and appropriate use of formalization. We highlighted the fact that there were areas where formal methods *could* be applied, but were not necessarily the most appropriate technique (e.g., user interface design).

Indeed, it was also one of our *Seven More Myths of Formal Methods* [2] that "formal methods people always use formal methods"; in reality, it turns out that they do not. Also in [2], we advocated the use of formal methods *when appropriate*, and emphasized that many of the highly publicized projects proclaimed as great success stories have in fact only involved formalizing small parts (often 10% or less) of the system. We also reported that at that point (ten years ago), regrettably, most formal methods toolsets had *not* been formally developed. This is still largely the case, although most of *PerfectDeveloper* (`http://www.eschertech.com/products`) from Escher Technologies has been developed using itself and the SPARK toolset (`http://www.praxis-his.com/sparkada`) from Praxis High Integrity Systems provides another example of a tool that has been applied to itself.

The formal methods community seems to have taken the warning not to over-formalize somewhat to heart, and there is now more widespread belief that formal methods should be used for key parts of the product. Cliff Jones introduced the idea of lightweight formal methods, or "formal methods light", which more or less equates to Level 0 of the three levels of formalization we proposed back in 1995, illustrated in Table 2.

Level	Name	Involves
0	Formal Specification	Formal notation used for specifying requirements only; no analysis/proof
1	Formal Development/Verification	Proving properties and applying refinement calculus
2	Machine Checked Proofs	Use of a theorem prover/checker to prove consistency and/or integrity

Table 2. Levels of formality

Certainly it is true that much benefit can accrue using formality only at the level of requirements specification (Level 0). The importance of getting requirements right at the outset cannot be overstated. Indeed, Plato (428–348 BC) said, "The beginning is the most important part of the work." 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 later!).

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 the use of 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.

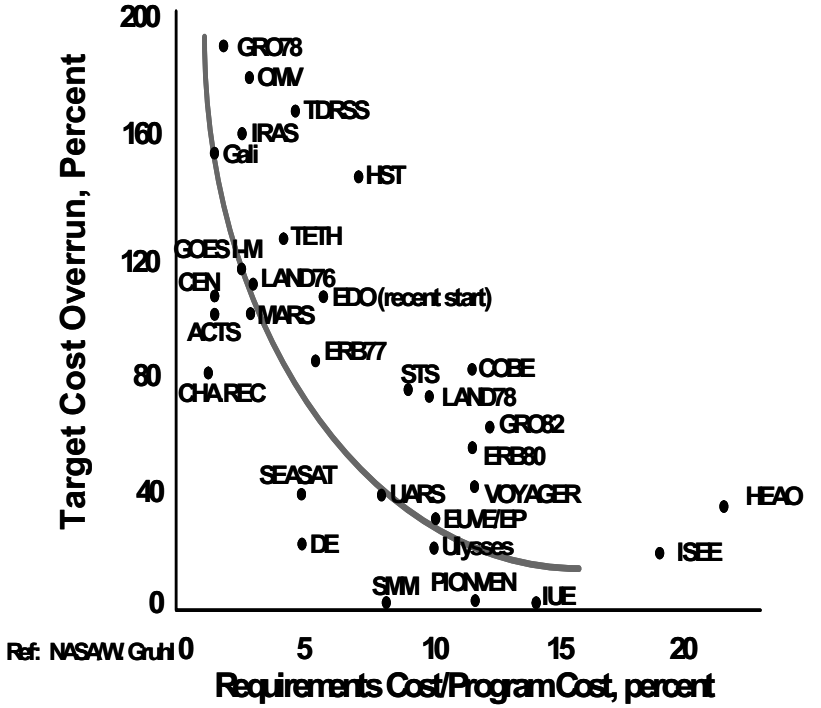

Figure 1. Requirements phase costs compared with project overrun costs (source: W. Gruhl, NASA Comptroller's Office)

III. Thou shalt estimate costs.

I think that God in creating Man somewhat overestimated his ability.
— Oscar Wilde (1854–1900)

It is an old joke amongst software engineers that in setting the price for a software project, one charges the customer as much as one can possibly get away with. Similarly, it is said that in estimating the cost of development, one should make a best (high) estimate, and then double it!

Drafts of our original *Ten Commandments* paper commanded, "Thou shalt guestimate costs". The term "guestimate", a hybrid of "guess" and "estimate" was an attempt to indicate that this is far from a precise science.[2]

Notwithstanding the existence of several established cost estimation models (such as CoCoMo II, etc.), cost estimation is far from an exact science. There have been many notable examples of system development where costs greatly exceeded estimates: for example, the Darlington power plant and Space Shuttle software, where cost overruns were significantly more than could ever have been foreseen. In [2], we strongly advocated both initial and continuous cost estimation.

We concede that in many cases, this may still be guesswork. In particular, research shows that organizations spend 33% to 50% of their total cost of ownership (TCO) on preparing for, or recovering from, failures. While hardware costs continue to fall in price, TCO continues to rise and system availability (and hence reliability) is falling. In this light, *any* cost estimates may well be unrealistic and/or understated.

However, we still firmly believe that having an estimate of costs, and also, ideally, an estimate of anticipated costs were formal methods not employed, is essential for convincing the software (and hardware) development communities that formal methods can indeed produce better systems cheaper.

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 (1885–1962)

The guidance offered in our original *Ten Commandments* paper was based on our knowledge of prior and in-progress industrial projects. It was clear that most of the successful projects had regular or continuous access to a formal methods expert, or "guru". Many projects had several such people available to them to guide and lead the formal development process, to provide advice on complex aspects, and in some cases to compensate for the lack of experience of the development team in applying formal methods.

Perhaps one might infer from this that it suffices to have access (whether occasional or regular) to an expert who is not actually part of the team. In reality, all members of the software project team must understand the applicability of formal methods and contribute in ways that help ensure success. It is very easy for any member of a team, whether on the management or technical side, or both, to prevent their effective use.

---

[2] Unfortunately, the copyeditor did not like the term.

Formal methods require effort, expertise, and significant knowledge in order to be successful. However, the rewards can be very worthwhile if the right mix of people is available. Not everyone in a team needs the same level of proficiency in the application of formal methods, but all must have an appreciation of their role. A formally verified program is only as good as its specification. If the specification does not describe what we truly want, then even a fully formally developed system may be little more than useless. Lack of understanding of formal methods, of what was specified using a formal notation, and of refinement in the development process, will almost certainly result in disaster. This is perhaps why formal methods are still not trusted in some quarters.

While we fully stand by this commandment, our experience of industrial projects over the last decade has indicated to us that it is equally important to have domain expertise on hand, whether supplied by the developer or the customer. It has also been our experience that having both formal methods expertise and domain expertise *from the outset* is important [10].

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 (1898–1963) *The Abolition of Man*

The use of UML (Unified Modeling Language) has become increasingly important and ubiquitous in industrial software development over the last decade. However, UML can justifiably be criticized for a number of reasons. We will concentrate on just two issues here.

The first is UML's lack of formality. However, there has been much work by the formal methods research community in considering the role of formality in the context of UML and a pUML (precise UML) group has been formed. Work is underway to allow tool-based integration of the B-Method with UML. Even the UML community recognizes that improvements could be made in this direction and developments in UML are likely to include aspects that are more formal.

The second criticism[3] is equally important. UML essentially standardizes a number of (pre-existing and emerging) graphical notations for system specification. Many of the notations it uses have been available since the 1970s, although perhaps their representation has varied slightly since then. In recent years, a wide variety of new graphical notations has been added to UML. Some of these have been added for good reason, others because they had support from particular quarters. A major issue when using UML is its lack of emphasis on a development method. While a range of notations is provided, no guidance is given as to what notations should be used for which types of systems, which combination of notations conflict, and which combination of notations complement each other well.

In fairness to UML, it can be said that most formal methods and most formal approaches to software (or hardware) development fail to address methodological aspects of the development process adequately as well. They are truly formal systems (having a specification notation and reasoning mechanism) but not "methods" in the true sense of offering 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 widely used and there has been much research on object-oriented extensions to formal methods, especially the Z notation (for example, Object-Z). In addition,

[3] There are other issues with UML, but these are not relevant here.

there are formal methods tools, such as *PerfectDeveloper*, aimed at object-oriented development. Using such a tool may be more attractive to software engineers who are used to developing systems with programming languages such as Java.

Work has also been undertaken to address formality in Model-Based Development (MBD), and to increase formality in Requirements-Based Programming, an approach that aims to transform requirements into executable code systematically. This has many of the advantages of automatic programming, but avoids one major deficiency, namely that automatic programming specifies a solution rather than the problem to be solved.

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 (1869–1954)

One of the great masters in the use of abstraction was the artist Henri Matisse. While most artists prepare preliminary drawings for their works, and then greatly expand these, Matisse worked the opposite way: his preliminary drawings were extremely detailed. He would have his assistant take photographs of his work each evening when he had finished working, in order to keep a record of the decisions he had made and the work he had completed. Next morning he would destroy the work, undoing most (and, sometimes, all) of what he had added the previous day. The result is that Matisse's preliminary drawings have much detail, whereas the final works are often very abstract, with very few lines, all of which are essential to the representation. Perhaps most effectively this is seen in his illustrations for James Joyce's Ulysses (1935).[4]

Knuth introduced the idea of literate programming, whereby programs and documentation were weaved together into a single framework. The concept is that fragments of code are connected to relevant documentation in a way that provides justification for coding (and design) decisions. Such an approach would certainly seem to be highly appropriate for formal development projects, whereby code could be associated with the relevant formal specification and/or design fragments, as well as the requirements that those specification (and/or design) fragments are driven by. However, it is interesting that the attempt to build tools for literate programming led to the development of Extreme Programming (XP), an approach to software development that provides little documentation and emphasizes development of a product and frequent software releases.

We emphasize our belief, however, that formal methods *necessitate* quality documentation. Some of this could be automated but, in any case, it is necessary that formal specifications be fully explained so that they may be understood by non-specialists and by specialists working on the specification after its initial development. Moreover, it is important that the reasons for various specification and design decisions and various decomposition decisions be recorded for the use of future developers.

It is felt that in addition to the benefits of abstraction, clarification and disambiguation, which accrue from the use of formal methods at Level 0 according to our classification (see Commandment II), using formal methods at the level of formal specification provides invaluable documentation. Experience has shown that quality documentation can greatly assist in future system maintenance. In fact, there have been several collaborative European projects involving the documentation of legacy

[4] Matisse did not even read the book; he illustrated Homer's Odyssey instead.

systems or reverse engineering. Such legacy systems simultaneously generate a large quantity of useful documentation.

All development involves iteration. It is important that documentation should reflect that fact. Often when changes are made to system implementations, a record of the changes is not made and updates are not made to the related documentation. If we truly are developing systems formally, formal methods help us to avoid this inconsistency, as the formal specification itself forms part of the documentation.

Additionally, proper documentation of decisions made during the formal specification process is important. This is why we have previously always advocated augmenting formal specifications with sufficient natural language narrative. It is critical that a proper "paper trail" is available. Abstraction is a very useful tool, but it requires proper documentation, or it may result in the loss of 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 (1475–1564)

The National Institute of Standards & Technology (NIST) estimated in 2002 that economic losses due to poor software quality amounted to more than US$60 billion. Software quality is still a huge issue that has yet to be addressed adequately. The ISO 9000 family of quality standards have been in force for a significant period now and were revised in 2000, yet poor software quality still plagues our economy. Standards may pay a crucial role in ensuring the quality of future software.

Standards are also especially important in high integrity areas like safety-critical and security-critical applications. For example, the IEC 61508-3 International Standard on software requirements with regard to the functional safety of safety-related systems covers software design, development and verification. Obviously formal methods can be used as part of this process. However most standards do not mandate formal methods, but rather suggest that they could be used. The onus is, rightly, on the developer to demonstrate that their use is sensible and worthwhile.

Other standards take even more consideration of formal methods by mandating there use when appropriate. For example, in the UK, the two-part Defence Standard 00-55 from the Ministry of Defence, originally issued in 1991, was reissued in 1997. Part 1 on "Requirements" states: "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." In addition, with regard to safety-related software (SRS): "The methods used in the SRS development process shall include all of the following: a) formal methods of software specification and design; …" Part 2 provides "Guidance" with formal methods mentioned in many places and an explicit section included under "Required methods".

Safety and security standards continue to play an important driving force in the use of formal methods, especially in the associated guidance sections and at the highest levels of integrity. It is likely that this will continue in the near future.

While standards may drive the use of formal methods, it is also important that formal methods practitioners keep quality standards in mind in their development processes. This includes adhering to standards for various specification notations (such as Z, mentioned above), but also in ensuring that other quality standards are applied. The use of formal methods is no guarantee of correctness; we need to ensure that we apply best practice in our software development, whether that software is deemed "critical" or not. Formal methods may complement existing quality standards, not supplant them.

The documentation of computing related standards can use formality (e.g., for the logic programming language Prolog) and formal notations themselves are becoming standardized. LOTOS, VDM and Z all have international ISO standards associated with them. The ISO standard for the Z notation was accepted in 2002 after nearly a decade of effort in its production. This was perhaps an example of over-documentation, since much of the time was spent formalizing (a revised version of) the Z notation. However, the process did reveal some inconsistencies in the semantics; thus, it could be considered a success from this point of view. However, progress was slow and painstaking. Any future efforts to produce formal methods standards should learn from this experience.

VIII. Thou shalt not be dogmatic.

... And I am unanimous in that!

— Mollie Sugden, a.k.a. Mrs. Slocombe
Are You Being Served? BBC TV (1972–1993)

It has been incorrectly claimed that formal methods can guarantee correctness [2]. While formal methods can certainly offer greater confidence that the software (or hardware) which has been developed has been done so correctly, formal methods are no absolute guarantee. In fact, it is absurd to speak of "correctness" without reference to the system specification [2].

However, proving that a system is built "right" (verification) is of extremely limited benefit if we are not building the "right" system (validation). McKenzie examined 1,100 or so deaths where the cause of the death was attributed to be due to computer error. It was determined that many of the errors were due to specifications that were lacking, rather than that the specifications were not correctly implemented.

There is a "gap" (sometimes called the Analysis–Specification Gap) in going from what is in the mind of the procurer (expressed in terms of real world entities) to a specification using the notations of software professionals (whether formal or informal). Because what we term "formal methods" in fact offer very little or no methodological support (with a few exceptions), it has often been suggested that less formal methods are preferable, or that formal methods should be augmented with other methods that offer greater development support and/or are more intuitive to end-users. (Cf. our discussion of the field of method integration under Commandment I.) Model-Based Development (MBD) aims to address this by placing great emphasis on getting an appropriate model of reality (cf. Commandment V). Moreover, as we mentioned earlier (Commandment V), the field of Requirements-Based Programming is attempting to integrate requirements in the development process fully.

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., *No Silver Bullet*

A widely used consequence of early formal methods research from the 1960s (before the term "formal methods" had even been coined by the community) is the inclusion of assertions in most professionally

produced programs. Originally, these were designed for proving programs correct; in fact, it was originally assumed that formal methods only involved proving programs correct [2]. Nowadays, assertions they are normally used for testing purposes to check if a program's state is correct during runtime. There is now promising research based around JML (Java Modeling Language) that allows assertions to be used for both runtime checking and formal verification. Looking further into the future, perhaps a "verifying compiler" as proposed by Tony Hoare will be able to verify assertions at compile-time rather than runtime, thus helpful to avoid the need to use them for testing. A current computer science "Grand Challenge" proposes the development of such a compiler as a long-term goal.

For the nearer term, the use of formal methods to improve testing seems increasingly promising. A formal specification can aid in the automation of generating test cases. In may be that the time required to produce a formal specification more than makes up the time saved at the testing stage in this regard. In the UK, a nationwide government-funded EPSRC network, FORTEST (Formal Methods and Testing, `http://www.fortest.org.uk`) has been acting as a framework for investigations into the interplay between these two aspects.

In addition, formal methods may be used to clarify testing criteria. For example, the MC/DC (Modified Condition/Decision Coverage) criterion used in many safety-related applications, and recommended by standards like the RTCA/DO-178B *Software Considerations in Airborne Systems and Equipment Certification* standard, is normally defined informally, as in this standard. Its meaning has been investigated formally using the Z notation and developed further into an even stricter RC/DC (Reinforced Condition/ Decision Coverage) criterion.

The testing of software has particular problems because it is unique in many senses:

- Even very short programs can be complex and difficult to understand.
- Software does not deteriorate with age. In fact, it may be improved over time by the discovery and correction of latent errors.
- However, new defects may be introduced during corrective changes to software.
- Changes in software that appear to be inconsequential can result in significant and unexpected problems in other (seemingly unrelated) parts of the code.
- While some hardware can give forewarnings of failure, this is not the case with software. Many latent errors in software may not be visible until long after the software has been deployed.
- A characteristic of software is the speed and ease with which it can be changed.

This last point may give the incorrect impression that software errors can easily be found and corrected. Rather, testing must be augmented with other verification techniques, and a structured and well-documented development approach must be combined to ensure a comprehensive validation approach. However, we would never, and have never, claimed that the use of formal methods can eliminate the need for testing.

The FDA concludes: "Because of its complexity, the development process for software should be even more tightly controlled than for hardware, in order to prevent problems that cannot be easily detected later in the development process", and that "time is needed to fully define and develop reusable software code and to fully understand the behavior of off-the-shelf components."

While the use of formal methods can certainly reduce the likelihood of certain types of errors, and/or help in detecting errors that have been admitted in the development process, their use must be augmented by appropriate testing.

X. Thou shalt reuse.

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

Traditionally, reuse has been encouraged as a means of reducing costs and achieving greater quality in software development (as greater effort can be justified on improving the quality of components that will be reused). Object-oriented and component-based paradigms aim to exploit this in developing complex software systems.

Theoretically, formal methods can and *should* aid in promoting software reuse. One of the inhibitors to the uptake of software reuse has been the ability to identify suitable components in a library, and to develop libraries of components that are sufficient large to give a reasonable return, and yet small enough to be reusable in a variety of situations. For some time it has been recognized that searching can be made more effective by having formal specifications of components, or at the very least of their preconditions (which specify appropriate situations in which the component may be applied) and postconditions (which specifies the result of using the component). Supplied with such pre and postconditions, the component may truly remain a "black box", which allows us to use larger components for which the payoff may be more significant.

There are significant paybacks in reuse being applied at the level of formal specifications rather than at the code level. Formal specifications are typically shorter than their equivalent implementation in a programming language. (See Figure 2 for a comparison of the potential size explosion, as development proceeds from specification down to hardware implementation.) As such, it is easier to search for components, while simultaneously getting a sufficient return. The same may apply in an approach based on design patterns, where formal specification can help identify reusable patterns.

Software reuse has been promoted as a means of reducing costs and achieving greater quality in system development (as greater effort can be justified on improving the quality of components that will be reused). Object-oriented and component-based paradigms aim to exploit this in developing complex software systems.

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

Additionally, formal specifications may be used to generate implementations on various platforms, reusing the effort expended at the earlier stages of the development process, and reducing the overall

cost. In particular, success has been reported in applying formal specification techniques to developing software product lines (SPL), whereby a range of similar systems (or products) that have significantly similar properties, with slight variations between them, are implemented. Moreover, formal methods generally result in a cleaner architecture, making a system more efficient and more easily maintainable in the future.

However, care must be taken when reusing and porting software. Ariane 5 is a prime example, where it was assumed that the same launch software used in the prior version (Ariane 4) could be reused. The result was the loss of the rocket within seconds of launch.

Similarly, the Therac-25 incidents are an interesting and relevant example of, arguably, the most significant failure of software assurance in the medical/biological field. 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. While the Therac-20 included hardware interlocks for safety, in Therac-25 these were software-based. Despite several Therac-25 machines operating, reportedly correctly, for up to four years at various installations in the US, six incidents occurred where the device gave massive (and lethal) doses of radiation to patients.

Subsequent investigations discovered that "creative" setting of parameters by students at a radiology school regularly resulted in Therac-20 machines shutting down due to blown fuses and breakers. In fact, in transpired that Therac-20 incorporated the same software error as Therac-25, but what was merely a nuisance in Therac-20 (due to mechanical interlocks) was a fatal problem with Therac-25. The problem was "inherited" and exacerbated in Therac-25.

4 The Future of Formal Methods?

> *Oui, l'ouvre sort plus belle*
> *D'une forme au travail*
> *Rebelle,*
> *Vers, marbre, onyx, émail.*
>
> [Yes, the work comes out more beautiful from a material
> that resists the process, verse, marble, onyx, or enamel.]
>
> — Théophile Gautier (1811–1872) *L'Art*

Although the use of formal methods has not developed as fast as might have been hoped over the last ten years, the formal approach to software and hardware development has certainly not gone away. We believe that formal methods are not just a passing trend, but that they will always have a niche in computer-based systems development, especially when correct functioning is critical (e.g., for safety or security reasons). The use of software in such applications is increasing as in many areas and formal methods are one of the available techniques that should be considered very carefully. They should be applied in the parts of the software that perform critical operations at a level that makes economic sense using engineering judgment. For that, well-trained personnel of the highest quality will always be needed.

Formal methods can have a significant impact on the software development lifecycle. Unfortunately, it is much easier to use formal methods inappropriately than it is to apply them successfully, unless a great deal of engineering skill and expert knowledge is used. As covered under Commandment IV, all members of the team must understand the applicability of formal methods to a

software project and contribute in ways that help ensure success. This is perhaps why formal methods are distrusted in some quarters.

A traditional problem of formal methods has been their overselling by some, especially in academia. They cannot solve all problems and they are certainly not completely reliable since humans, as well as mathematics, are involved. The logical models must relate to the real world in an informal leap of faith, in any case, both at the high-level requirements or specification end, and at the low-level digital hardware end (where ultimately we must believe Maxwell's equations, for example!). Formal methods are not a panacea, but rather they are a useful tool in reducing errors in computer-based systems when applied sensibly, in cost-effective ways, and for appropriate parts of the development.

There needs to be more effort applied in evaluating the effectiveness of formal methods in the software development and maintenance process. We hope that this paper has suggested some issues for consideration in future studies that we believe would be worthwhile. Because of the somewhat tarnished reputation of formal methods, largely due to misunderstandings and inappropriate use, a demonstration of how and where formal methods are effective would be well worthwhile. There are continuing success stories in the industrial use of formal methods [11] and the approach remains in the eye of the press [12]. Leading practitioners with many years experience are still enthusiastic about their use [7]. What are needed are studies that can 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.

Over the next ten years, we see tool support for formal methods as being of great importance. Industrial-strength tools for formal methods have always been lacking. There are a few examples, such as *Atelier-B* and *PerfectDeveloper*, but we need a range of such tools, perhaps compatible using XML interchange formats for example. There are some collaborative efforts in this direction. For example, see the CZT Community Z Tools initiative (`http://czt.sourceforge.net`), the European RODIN Project on Rigorous Open Development Environment for Complex Systems based around B# (`http://rodin-b-sharp.sourceforge.net`), a development of the B-Method (which includes a free version under `http://www.b4free.com`), and the HOL 4 theorem prover (`http://hol.sourceforge.net`). It is hoped that advances in tools will make formal methods easier to justify and use in an industrial context.

Finally, documentation online is increasingly important. The *Virtual Library formal methods* website (`http://vl.fmnet.info`), originally established over ten years ago, continues to be a central resource to find information on formal methods. More recently, Wikipedia (`http://en.wikipedia.org`), an online encyclopaedia to which anyone can contribute directly via a web interface, has included increasingly useful information on formal methods and related topics. This may be the way forward for further online information, maintained in a collaborative manner. A Verified Software Repository is now planned, through the recently funded UK EPSRC VSR-net network, in which examples of formally verified software and associated tools could be deposited for general use by those wishing to have real software to challenge their own tools.

> *... in this area my academic colleagues are doing exactly what they should do: developing and propagating an indispensable technology so that it will be available when "the world out there" undeniably needs it.*
>
> — Edsger W. Dijkstra (1930–2002)

Acknowledgements. We are grateful to our many colleagues and friends who provided us with valuable feedback and reactions to our original paper. We would like to acknowledge the contributions of the formal methods community as a whole, and thank the community for providing us with material on which to base the original commandments. We would particularly like to thank David Atkinson, Jin Son Dong, Cliff Jones, Tiziana Margaria, Jim Rash, Chris Rouff, and Bernhard Steffen, for providing us with input for this paper.

Thank you to Tiziana Margaria and Mieke Massink, co-chairs of FMICS 2005, for inviting an earlier conference version of this paper [4] to coincide with the tenth anniversary of FMICS. This paper is substantially based on that earlier paper. Our special thanks go to George Eleftherakis for inviting this talk to be given at SEEFM 2005 and the South-East European Research Centre for encouraging the use of formal methods in southeast Europe [5]. Finally, thank you to Scott Hamilton of *IEEE Computer*, for encouraging us to revisit the commandments ten years on from the original paper [1]. A revised version of this current paper is to appear in the January 2006 issue of *Computer*.

References

1. Bowen, J.P. and Hinchey, M.G., Ten Commandments of Formal Methods, *Computer,* **28**(4):56–63, April 1995. Reprinted in [3].
2. Bowen, J.P. and Hinchey, M.G., Seven More Myths of Formal Methods, *IEEE Software,* **12**(4):34–41, July 1995. Reprinted in [3].
3. Bowen, J.P. and Hinchey, M.G., editors, *High-Integrity System Specification and Design*, Springer-Verlag FACIT Series, London, 1999.
4. Bowen, J.P. and Hinchey, M.G., Ten Commandments Revisited: A Ten-Year Perspective on the Industrial Application of Formal Methods. In *Proc. 10th Workshop on Formal Methods for Industrial Critical Systems (FMICS 2005)*, Lisbon, Portugal, 5–6 September 2005, ACM Press.
5. Drandis, D. and Tigka, K., editors, *Agile Formal Methods: Practical, Rigorous Methods for a Changing World*, 1st South-East European Workshop on Formal Methods, Thessaloniki, Greece, 21–23 November 2003. South-East European Research Centre, 2004.
6. Hall, J.A., Seven Myths of Formal Methods, *IEEE Software*, **7**(5):11–19, 1990. Reprinted in [3].
7. Hall, J.A., Realising the Benefits of Formal Methods. In K.-K. Lau and R. Banach, editors, *Formal Methods and Software Engineering*. Springer-Verlag LNCS **3785**, pages 1–4, 2005.
8. Hinchey, M.G. and Bowen, J.P., editors, *Applications of Formal Methods*, Prentice Hall International Series in Computer Science, Hemel Hempstead, UK and Englewood Cliffs, NJ, 1995.
9. Hinchey, M.G. and Bowen, J.P., editors, *Industrial-Strength Formal Methods in Practice*, Springer-Verlag FACIT Series, London, 1999.
10. Hinchey, M.G., Confessions of a Formal Methodist, In *Proc. 7th Australian Workshop on Safety Critical Systems and Software (SCS'02)*, Adelaide, Australia, Conferences in Research and Practice in Information Technology, Vol. **15**, Australian Computer Society, 2002.
11. Ross, P.E., The Exterminators, *IEEE Spectrum*, **42**(9):36–41, September 2005.
12. Sharpe, R., Formal Methods Start to Add up Again, *Computing*, **301**, 8 January 2004. `http://www.computing.co.uk/features/1151896`