Ten Commandments Revisited

A Ten-Year Perspective on the Industrial Application of Formal Methods

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

Ten years ago, our 1995 paper *Ten Commandments of Formal Methods* [5] suggested some guidelines to help ensure the success of a formal methods project. It proposed ten important requirements (or "commandments") for formal developers to consider and follow, based on our knowledge of several industrial application success stories, most of which have been reported in more detail in two books [17],[18]. The paper was surprisingly popular, is still widely referenced, and used as required reading in a number of formal methods courses. However, not all have agreed with some of our commandments, feeling that they may not be valid in the long-term. We re-examine the original commandments ten years on, and consider their validity in the light of a further decade of industrial best practice and experiences.

Categories and Subject Descriptors

D.3.3 [**Software/Program Verification**]: Formal methods.

General Terms

Design, Economics, Experimentation, Human Factors, Standardization, Languages, Theory, Verification.

Keywords

Formal methods, correctness, industrial application, software engineering.

1. 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

FMICS'05, September 5–6, 2005, Lisbon, Portugal.

We, as the formal methods community, are (presumably) convinced of the validity of formal approaches to software specification, design, implementation, and subsequent maintenance. In fact, it seems a logical argument to "Formal Methodists", for whom formal methods are a "religion," that the introduction of greater rigor into software development will result in improvements both in the software itself and in the development process.

Unfortunately, the rest of the world (and the software engineering community, in particular) has not been convinced on a wide scale, despite a significant number of success stories [17],[18]. At least their existence is now acknowledged by most [8], and their usefulness is accepted by some [41]. Certain aspects of formal methods, such as assertions in programs [20], are widely used, although not as much as originally anticipated. However, many myths regarding formal methods, first identified as far back as 15 years ago [6],[16] still abound.

Holloway [22] points out that the typical argument in favor 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). He proposes a simpler argument, which is both simple and logically valid: software engineers want to be "real" engineers; such engineers apply mathematics; and since formal methods is the mathematics of software engineering, software engineers *should* use formal methods.

Nevertheless, formal methods are still *not* widely used outside the specialized formal methods community. Our original *Ten Commandments of Formal Methods* [5] were aimed at encouraging the practical use of formal methods among the software engineering community as a whole, and, more importantly, to provide some practical guidance to formal methods practitioners based on insights received from a number of real-life projects, many of which were reported in [17],[18]. The future of formal methods was also considered by others at around the same time [10].

Given the lack of acceptance of the use of formal methods outside our own specialized group of people, the former was at best a very limited success (or perhaps even a failure!). But we were considerably more successful in achieving the latter goal.

Over the intervening ten years, we have received many comments and much feedback on our "commandments". They have been widely cited, and included in several textbooks (e.g., see [36]) and are recommended (or even required) reading for a number of academic courses. Naturally, not everyone agrees with all of them.

They did not do so when we first published them in 1995, pointing out that many of them would not necessarily hold in the long term, so it is not surprising that not everyone agrees with them now.

In this paper, we re-examine the *Ten Commandments of Formal Methods* in the light of a further ten years of industrial best practice.

> *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)

2. REVISITING THE COMMANDMENTS

> *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

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 use in industrial applications. However, as we pointed out in [6], the mathematics of formal methods are actually relatively simple, and exploits notations and concepts that *should* be well-understood by computing professionals (set theory, propositional and predicate logic, etc.). While we concede that non-professionals may not be so *au fait* with such notations — which makes communication more difficult, in particular with system procurers, who in many cases are the people with whom the formal specifier most needs to communicate — in general, the notations are not beyond the understanding of well-educated software engineers.

The formal methods community must also take some of the blame for this: all too often we find authors of technical papers introducing new symbols, Greek hieroglyphics, which are just alternative ways of representing existing operators and concepts. This kind of obfuscation should be avoided if possible.

However, the original intent of our commandment ("thou shalt choose an appropriate notation") concerned a notation that was not "appropriate" in the sense that it was easily understood by non-specialists, but "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, Z, CCS, CSP) have emerged and have widespread applicability to a broad range of classes of system, it has been found in many cases that a combination of languages is needed to adequately address all aspects of a larger system. It has 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 some of the wide range of "hybrid" formal methods that have emerged over the last decade, 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:[1]

- *Viewpoints (loosely coupled*): different notations are used to present different "views" of a system with each notation making emphasis of, or understand of, a particular aspect of the system (e.g., representing timing constraints) [9].
- *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 both to provide a semantics for the less formal notations and to enable graphical (or other) presentations that are well-understood, while simultaneously affording the benefits of formal verification [40].
- *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 [46].

At the time we published [5], 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 winning out, perhaps due to reluctance by industry to take up full formal proof (that the more coupled approaches would support) and reluctance to get involved in semantic details. The *Integrated Formal Methods* conference [2] continues to provide a forum for this research topic.

Choosing the right notation can great aid in *abstraction*, in hiding unnecessary detail and unnecessary complexity, where is where (many argue) the real benefit of formal methods can most be felt.

> *If I could say it in words there would be no reason to paint.*
>
> — Edward Hopper (1882–1967)

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

Table 1: A sample of some hybrid formal methods (developed since our 1995 paper [5]).

Name	Combines	Advantage	Reference
Temporal B	B, temporal logic	Adds time to the B-Method	Bonnet *et al*. (1995) [1]
ZCCS	Z, CCS	Combines CCS process algebra and state based aspects of Z	Galloway and Stoddard (1997) [15]
CSP OZ	Z, CSP	Combines Z and CSP	Fischer (2000) [13]
Object Z	Z, OO principles, temporal logic	Adds OO to Z	Smith (2000) [41]
PiOZ	Object-Z, π-calculus	Adds π-calculus style dynamic communication capabilities to Object-Z	Taguchi *et al*. (2004) [45]

II. Thou shalt formalize but not overformalize.

Back in 1995, 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* [6] that "formal methods people always use formal methods"; in reality, they do not. Also in [6], we advocated the use of formal methods *when appropriate*, and emphasized that many of the highly publicized projects touted as great success stories have in fact only involved formalizing small parts (often 10% or less) of the system. (We also reported that, regrettably, at that point most formal methods toolsets had *not* been formally developed. To our knowledge this is still largely the case, although PerfectDeveloper [11] has made some attempt in this direction.)

The formal methods community has taken this somewhat to heart. Jones introduced the idea of lightweight formal methods with "Formal Methods Light" [39], which more or less equates to level zero of the three levels of formalization we proposed back in 1995, illustrated in Table 2.

It is certainly true that much benefit can accrue through the use of formality only at the level of requirements specification (Level 0). 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 later!).

Table 2: Levels of formality.

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 theorem prover/checker to prove consistency and/or integrity.

It is clear that the use of mathematically-based approaches has the potential to help eliminate errors early in the design process, rather 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.

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

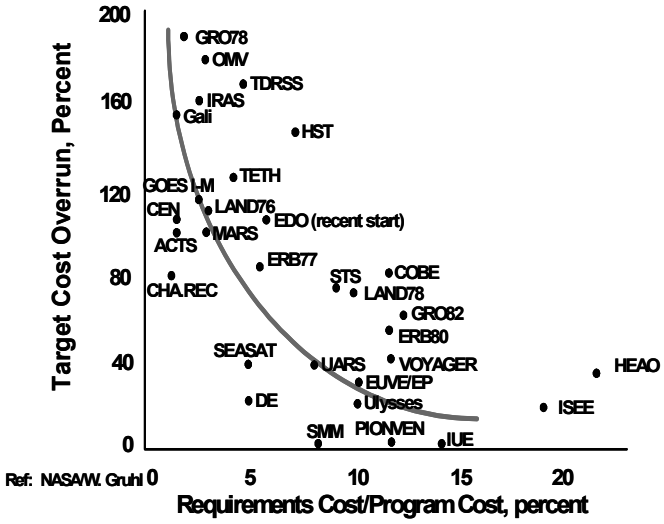

Figure 1: Requirements phase costs compared with project overrun costs (source: W. Gruhl, NASA Comptroller's Office).

III. Thou shalt estimate costs.

Earlier drafts of our 1995 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, involving a lot of guesswork.[2]

Notwithstanding the existence of several excellent cost estimation models (such as CoCoMo II, etc.), cost estimation is still 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 the Space Shuttle software where cost overruns were significantly more than were foreseen. In [5], 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, TCO continues to rise and system availability (and hence reliability) is falling [44]. Therefore *any* cost estimates are likely to 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 to convincing the software (and hardware) development communities that formal methods can indeed produce better systems cheaper.

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

— Oscar Wilde (1854–1900)

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

Our experience prior to 1995 was that most successful formal methods projects had significant support in the guise of a formal methods expert, or "guru". Many projects had several such gurus available 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 (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 only too easy for any member of a team, whether on the management or technical side, or both, to prevent their effective use. Formal methods require effort, expertise, and significant knowledge in order to be successful. However, the rewards can be great 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. Lack of understanding will almost certainly result in disaster. This is perhaps why formal methods are still not trusted in some quarters.

It is still particularly important that the manager of the team understands the shift of emphasis of effort towards earlier phases of the development cycle (e.g., specification), with the potential to reap the rewards in the later phases (e.g., during testing).

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

— Niels Bohr (1885–1962)

V. Thou shalt not abandon thy traditional development methods.

In the last decade, the use of UML (Unified Modeling Language) has become increasingly important and ubiquitous in industrial software development. A criticism that has been leveled at UML is its 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 [13],[25] and a pUML (precise UML) group has been formed. Work is underway to allow tool-based integration of the B-Method with UML [43]. Even the UML community recognizes that improvements could be made in this direction and developments in UML are likely to include more formal aspects.

Object-oriented techniques are also widely used and there has been much research on object-oriented extensions to formal methods, especial the Z notation (for example, Object-Z [42]). In addition, there are formal methods tools aimed at object-oriented development, such as PerfectDeveloper [11]. Using such a tool may be more attractive to software engineers who are used to developing systems using 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 [37], an approach that aims to systematically transform requirements into executable code, having many of the advantages of automatic programming, but avoiding one major deficiency, namely that automatic programming specifies a solution rather than the problem to be solved [35].

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*

VI. Thou shalt document sufficiently.

The ISO standard for the Z notation was accepted in 2002 after nearly a decade of effort in its production [24]. 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 awkward corners in the semantics, and so could be considered a success from this point of view. But progress was slow and painstaking.

[2] Unfortunately the copyeditor did not approve of the term.

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.

All development involves iteration. It is important that documentation reflects 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.

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 a lot of 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).[3]

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)

VII. Thou shalt not compromise thy quality standards.

In 2002, the National Institute of Standards & Technology (NIST) estimated that economic losses due to poor software quality amounted to more than US$60 billion [34]. Thus the issue of 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.

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 [23]. 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 [32]. 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 carry on for the foreseeable future.

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

— Michelangelo Buonarroti (1475–1564)

VIII. Thou shalt not be dogmatic.

It is often erroneously claimed that formal methods can guarantee correctness [6]. 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 [6].

However, proving that a system is built "right" (verification) is of extremely limited benefit if we're not building the "right" system (validation) [26]. McKenzie [30],[31] 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) [6], 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 achieving an appropriate model of reality (cf. Commandment V). And, as we mentioned earlier (Commandment V), the field of Requirements-Based Programming is attempting to fully integrate requirements in the development process.

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

... And I am unanimous in that!

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

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

One of the most widely used results 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 [21]. Originally these were designed for proving programs correct. However they are now 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 both for runtime checking and formal verification [27]. Even further into the future, perhaps a "verifying compiler" will be able to verify assertions at compile-time rather than runtime, thus helpful to avoid the need to use them for testing [20].

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 network, FORTEST (Formal Methods and Testing) has been acting as a framework for investigations into the interplay between these two aspects [4].

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 [48].

Testing of software has particular problems because it is unique in many senses [44]:

- 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 changes to software.
- Seemingly insignificant changes in software 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 [46]: "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."

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*

X. Thou shalt reuse.

Reuse has been promoted 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.

In theory, formal methods can and *should* aim in promoting software reuse [25]. 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 accruing to exploiting reuse 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 through to implementation in hardware).[4] As such, it is easier to search for components, while simultaneously getting a sufficient return.

[4] There are those researchers who argue that unless a formal specification is significantly shorter than its implementation, it is worthless. While this is a preferable and normal situation, if the formal specification enables an insight that could not be achieved at the programming language level, it is of great benefit.

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 product lines, 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 [26], 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 [29].

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 [38]. 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 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 4 years at various installations in the US, 6 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 [28]. The problem was "inherited" and exacerbated in Therac-25 [38].

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

3. CONCLUSION

> *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*

Formal methods can have a great deal of 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. All members of the team must understand the applicability of formal methods to a software project and contribute in ways that help ensure success. It is only too easy for any member of a team, whether on the management or technical side, or both, to prevent their effective use.

Formal methods *do* require effort, expertise, and significant knowledge, in order to be successfully applied. However the rewards can be 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. Lack of understanding will almost certainly result in disaster. 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 and 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 should be more effort to evaluate the effectiveness of formal methods in the software development and maintenance process. It is hoped that this paper suggests 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 success stories in the industrial use of formal methods. 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.

While the use of formal methods has not developed as fast as it might have done over the last ten years, it has not gone away either. We believe that formal methods are not just a passing fad, but that they will always have a niche in software development, especially when it is critically important that the software functions correctly (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.

For 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 [47]. There are some efforts in this direction. E.g., see the CZT Community Z Tools initiative [32] and the European RODIN Project on Rigorous Open Development Environment for Complex Systems based around B# [1], a development of the B-Method. It is hoped that such advances will make formal methods increasingly easy to justify and use in an industrial environment.

> *... 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.* [12]
>
> — Edsger W. Dijkstra (1930–2002)

ACKNOWLEDGMENTS

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 David Atkinson, Jin Son Dong, Cliff Jones, Jim Rash, and Chris Rouff, for providing us with input and references for this paper.

Our special thanks go to Tiziana Margaria and Mieke Massink, the FMICS 2005 Co-chairs, for inviting this paper to coincide with the tenth anniversary of FMICS, and to Scott Hamilton of *IEEE Computer*, for encouraging us to revisit the commandments ten years on.

REFERENCES

[1] Abrial, J-R., B#: Towards a Synthesis between B and Z. In Bert, D., Bowen, J.P., King, S., and Waldén, M., editors, *ZB2003: Formal Specification and Development in Z and B*, Third International Conference of B and Z Users, Turku, Finland, June 2003., pp 168–177, Springer-Verlag, LNCS **2651**, 2003.

[2] Boiten, E.A., Derrick, J., and Smith, G., editors, *Integrated Formal Methods*, 4th International Conference, IFM 2004, Canterbury, UK, April 2004. Springer-Verlag, LNCS **2999**, 2004.

[3] Bonnet, L., Florin, G., Duchien, L., Seinturier, L. A Method for Specifying and Proving Distributed Cooperative Algorithms. In *Proc. DIMAS'95: Decentralized Intelligent and Multi-agent Systems*, Cracow, Poland, 22–24 November 1995.

[4] Bowen, J.P., Bogdanov, K., Clark, J., Harman, M., Hierons R., and Krause P., FORTEST: Formal Methods and Testing. In *Proc. 26th Annual International Computer Software and Applications Conference (COMPSAC 02)*, Oxford, UK, 26–29 August 2002, pp 91–101. IEEE Computer Society Press.

[5] Bowen, J.P. and Hinchey, M.G., Ten Commandments of Formal Methods, *Computer,* **28**(4):56–63, April 1995. Reprinted in [7].

[6] Bowen, J.P. and Hinchey, M.G., Seven More Myths of Formal Methods, *IEEE Software,* **12**(4):34–41, July 1995. Reprinted in [7].

[7] Bowen, J.P. and Hinchey, M.G., editors, *High-Integrity System Specification and Design*, Springer-Verlag FACIT Series, London, 1999.

[8] Bowen, J.P. and Hinchey, M.G., Formal Methods. In Allen B. Tucker, Jr., editor, *Computer Science Handbook*, 2nd edition, Section XI, Software Engineering, Chapter 106, pp 106-1–106-25, Chapman & Hall / CRC, ACM, 2004.

[9] Bowman, H., Steen, M.W.A., Boiten, E.A., and Derrick, J., A Formal Framework for Viewpoint Consistency, *Formal Methods in System Design,* **21**(2):111–116, September 2002.

[10] Clark, E.M., Wing, J.M., *et al*., Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys,* **28**(4): 626–643, 1996.

[11] Crocker, D., Safe Object-Oriented Software: The Verified Design-by-contract Paradigm. In Redmill, F. and Anderson, T., editors, *Practical Elements of Safety: Proceedings of the 12th Safety-Critical Systems Symposium*, Birmingham, UK, 17–19 February 2004, Chapter 2, Springer-Verlag.

[12] Dean, C.N. and Hinchey, M.G., editors, *Teaching and Learning Formal Methods*. Academic Press International Series in Formal Methods, London, 1996.

[13] Evans, A., France, R., Lano, K., and Rumpe, B., The Unified Modeling Language. In Bézivin, J. and Muller, P-A., editors, *UML'98: Beyond the Notation: First International Workshop, Mulhouse, France, June 3–4, 1998*, pp 336–348. Spinger-Verlag, LNCS **1618**, 1998.

[14] Fischer, C. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. PhD Dissertation, Fachbereich Informatik, Universität Oldenburg, Germany. 2000.

[15] Galloway, A.J. and Stoddart, W.J. An operational semantics for ZCCS. In M. Hinchey and S. Liu, editors, *Proc. IEEE International Conference on Formal Engineering Methods (ICFEM'97)*, pp 272–282, Hiroshima, Japan, November 1997, IEEE Computer Society Press.

[16] Hall, J.A., Seven Myths of Formal Methods, *IEEE Software,* **7**(5): 11–19, September 1990. Reprinted in [7].

[17] 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.

[18] Hinchey, M.G. and Bowen, J.P., editors, *Industrial-Strength Formal Methods in Practice*, Springer-Verlag FACIT Series, London, 1999.

[19] Hoare, C.A.R., How did Software get so Reliable without Proof? In *Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods*, pp 1–17. Springer-Verlag, LNCS **1051**, 1996.

[20] Hoare, C.A.R., The Verifying Compiler: A Grand Challenge for Computing Research. *Journal of the ACM,* **50**(1):63–69, January 2003.

[21] Hoare, C.A.R., Assertions: A Personal Perspective. *IEEE Annals of the History of Computing,* **25**(2):14–25, April–June 2003.

[22] Holloway, C.M., Why Engineers Should Consider Formal Methods, In *Proc. 16th Annual Digital Avionics Systems Conference*, October 1997; also available as NASA Langley Research Center Technical Report number 290694, 1997.

[23] IEC, *Functional Safety of Electrical/Electronic/ Programmable Electronic safety-related systems*, Part 3: Software Requirements, IEC 61508-3 International Standard, 1998.

[24] ISO/IEC, *Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics*, ISO 13568:2002 International Standard, 2002.

[25] Johnson, I., Snook, C., Edmunds, A., and Butler, M., Rigorous Development of Reusable, Domain-specific Components, for Complex Applications. In Jurgens, J. and France, R., editors, *Proceedings of 3rd International Workshop on Critical Systems Development with UML*, pp 115–129, Lisbon, 2004.

[26] Jones, C.B., Keynote Speech, In *Proc. 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2005)*, Shanghai, China, 16–20 June, 2005, IEEE Computer Society Press.

[27] Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R., How the Design of JML Accommodates both Runtime Assertion Checking and Formal Verification. *Science of Computer Programming*, **55**(1–3):185–208, March 2005.

[28] Leveson, N. and Turner, C.S., An Investigation of the Therac-25 Accidents, *Computer*, **26**(7):18–41, July 1993.

[29] Lyons J.L., *Ariane 5: Flight 501 Failure*, Report by the Inquiry Board, 19 July 1996.

[30] MacKenzie, D., Computer-Related Accidental Death: An Empirical Exploration, *Science and Public Policy,* **21**:233–248, 1994.

[31] MacKenzie, D., *Mechanizing Proof: Computing, Risk and Trust*, MIT Press, Cambridge, MA, 2001.

[32] Malik, P. and Utting, M., CZT: A Framework for Z Tools. In Treherne, H., King, S., Henson, M., and Schneider, S., editors, *ZB2005: Formal Specification and Development in Z and B*, 4th International Conference of B and Z Users, Guildford, UK, April 2005., pp 65–84, Springer-Verlag, LNCS **3455**, 2002.

[33] MOD, *Requirement for Safety Related Software in Defence Equipment*, Def Stan 00-55/Issue 2, Ministry of Defence, UK, 1 August 1997. Part 1: Requirements, Part 2: Guidance.

[34] National Institute of Standards and Technology, *The Economic Impact of Inadequate Infrastructure for Software Testing*, Planning Report 02-3, May 2002.

[35] Parnas, D.L., Software Aspects for Strategic Defense Systems. *American Scientist,* November 1985.

[36] Pressman, R.S., *Software Engineering: A Practitioner's Approach*, 6th edition. McGraw-Hill International Edition, 2004.

[37] Rash, J.L., Hinchey, M.G., Gračanin, D., and Rouff, C.A., An Approach to Generating and Verifying Complex Scripts and Procedures. In "Controlling Complexity" workshop, *Proc. Computational Sciences Bioinformatics 2005 Workshops*, 11 August 2005, Stanford University, USA, IEEE Computer Society Press.

[38] Rawlinson, J.A., *Report on the Therac-25,* OCTRF/OCI Physicist's Meeting, Kingston, Ontario, Canada, 7 May 1987.

[39] Saiedian, H., editor, An Invitation to Formal Methods. *IEEE Computer* **29**(4):16–30, April 1996.

[40] Semmens, L.T., France, R.B., and Docker, T.W.G., Integrated Structured Analysis and Formal Specification Techniques. *The Computer Journal* **35**(6):600–610, 1992.

[41] Sharpe, R., Formal Methods Start to Add up Again, *Computing,* **301**, 8 January 2004. http://www.computing.co.uk/features/1151896

[42] Smith, G., *The Object-Z Specification Language.* Kluwer Advances in Formal Methods Series, Boston, 2000.

[43] Snook, C. and Butler, M., U2B – A tool for translating UML-B models into B. In Mermet, J., editor, *UML-B Specification for Proven Embedded Systems Design*, chapter 6. Springer-Verlag, 2004.

[44] Sterritt, R. and Hinchey, M.G., Why Computer-Based Systems *Should* be Autonomic. In *Proc. 12th IEEE International Conference on Engineering Computer-Based Systems (ECBS 2005),* Greenbelt, MD, USA, 4–7 April 2005, pp 406–412, IEEE Computer Society Press.

[45] Taguchi, K., Dong, J.S., and Ciobanu, G. Relating π-calculus to Object-Z. In *Proc. 9th International Conference on Engineering of Complex Computer Systems (ICECCS 2004)*, Florence, Italy, April 2004.

[46] United States Department of Health and Human Services, Food and Drug Administration, *General Principles of Software Validation: Final Guidance for Industry and FDA Staff*, 11 January 2002.

[47] Utting, M. Toyn, I., Sun, J., Martin, A., Dong, J.S., Daley, N., and Currie, D., ZML: XML Support for Standard Z. In Bert, D., Bowen, J.P., King, S., and Waldén, M., editors, *ZB2003: Formal Specification and Development in Z and B*, Third International Conference of B and Z Users, Turku, Finland, June 2003., pp 437–456, Springer-Verlag, LNCS **2651**, 2003.

[48] Vilkomir, S. and Bowen, J.P., Reinforced Condition/ Decision Coverage (RC/DC): A New Criterion for Software Testing. In Bert, D., Bowen, J.P., Henson, M.C., and Robinson. K., editors, *ZB2002: Formal Specification and Development in Z and B*, Second International Conference of B and Z Users, Grenoble, France, 23–25 January 2002., pp 295–313, Springer-Verlag, LNCS **2272**, 2002.