Ten Commandments of Formal Methods

Jonathan P. Bowen
Oxford University Computing Laboratory
Programming Research Group
Wolfson Building, Parks Road, Oxford OX1 3QD, UK.
Email: `Jonathan.Bowen@comlab.ox.ac.uk`
URL: `http://www.comlab.ox.ac.uk/oucl/people/jonathan.bowen.html`

Michael G. Hinchey
University of Cambridge Computer Laboratory,
New Museums Site, Pembroke Street, Cambridge CB2 3QG, UK.
Email: `Mike.Hinchey@cl.cam.ac.uk`
URL: `http://www.cl.cam.ac.uk/users/mgh1001`

Abstract

The formal methods community is in general very good at undertaking research into the mathematical aspects of formal methods, but not so good at promulgating the use of formal methods in an engineering environment and at an industrial scale. Technology transfer is an extremely important part of the overall effort necessary in the acceptance of formal techniques. This paper explores some of the more informal aspects of applying formal methods and presents some maxims with associated discussion that may help in the application of formal methods in an industrial setting. A significant bibliography is included, providing pointers to more technical and detailed aspects.

Why does this magnificent applied science which saves work and makes life easier bring us so little happiness? The simple answer runs: because we have not yet learned to make sensible use of it.

– Albert Einstein

1 Introduction

Formal methods have been advocated as one of those techniques that are likely, when correctly applied, to result in systems of the highest integrity. A number of standards bodies are recommending their use in security- and safety-critical systems [8, 16]; this is a trend that is likely to continue [14].

Unfortunately, while the number of projects in which formal methods are being employed is growing rapidly, their use is still very much the exception rather than the norm [15]. This is due to not insubstantial misconceptions [17] regarding the costs, difficulties and pay-offs accruing as a result of their use [13, 37].

A number of surveys of the industrial application of formal methods to 'real-life' (as opposed to 'toy') problems [2, 24, 25] are helping to dispel many of these misconceptions and to highlight the fact that formal methods projects can indeed come in on-time, within budget, produce correct software (and hardware), that is well-structured, maintainable, and which has involved system procurers and satisfied their requirements (see, for example, the case studies in [40]).

But, what makes a formal methods project successful? This is a very subjective question, and to attempt a definitive answer would be ludicrous. We have, however, determined a number of factors which we believe can have a great influence on whether or not a formal methods project succeeds. Based on observations (by ourselves and others) on a number of recently completed and in-progress projects, both successful and otherwise, we have drawn up ten rules, or 'commandments', which we feel if adhered to will greatly increase the likelihood of success, and of reaching formal methods Nirvana.

2 Ten Commandments

I Thou shalt choose an appropriate notation. The specification language is the specifier's primary tool during the initial stages of system development. Obviously, as we are concerned with formal methods projects, we are assuming that the notation used at this stage will have a well-defined formal semantics.

Choosing the most appropriate notation is not as trivial as one might think. There are now a myriad of specification languages available, each making its own claims to superiority. Many of these claims are quite valid — different specification languages do indeed excel when used with particular classes of system.

There is always, necessarily, a certain degree of trade-off between the expressiveness of a specification language, and the levels of abstraction that it supports [66]. Certain languages may indeed have wider 'vocabularies' and constructs to support the particular situations we wish to deal with. But, they will also force us towards particular implementations, and while they will shorten the specification, they generally make it less abstract.

Languages with small 'vocabularies' on the other hand, while generally resulting in longer specifications, offer high levels of abstraction and little implementation bias. Consider Hoare's language of Communicating Sequential Processes (CSP) [42, 43], for example. The only first-class entities are processes (or pipes and buffers, which are merely particular types of process). CSP specifications can become quite lengthy as a result, but the fact that there are so few constructs with which to become familiar makes them readily understandable. Likewise, there is no bias towards the implementation of communication primitives, and CSP channels may be implemented as physical wires, buses, mailboxes, or even just shared variables.

The vocabulary is not the only issue to consider, however. Some specification languages are just not as good as others when used with particular classes of system. Trying to specify a concurrent system in a model-based specification language such as Z [62] or VDM, for example, is rather like using a hammer to insert a screw ... it can be done, but it is certainly not the best way to go about things. A process algebra such as CSP or CCS [51] is generally far more appropriate; but these suffer from the drawback of paying very little attention to state-based aspects of the system. This has resulted in much research aimed at integrating process algebras with model-based specification languages (e.g., [63, 68]) and extending model-based specification languages to handle concurrency and temporal aspects (e.g., [46]).

It is important to choose a well established notation with a good user base to ensure successful application in an industrial setting. Typically the development of a formal notation for industrial use takes at least a decade from conception to real application. It takes this long for the notation to be developed by researchers, taught to students, promulgated via academic/industrial liaison, for textbooks to be written, industrial courses to be developed, support tools to be marketed, a user community to be established, etc. The technology transfer of formal notations, as with many new developments, is fraught with hurdles any one of which could cause its downfall.

An important part of the general acceptance of a notation is the production of an international standard. This is of course a chicken and egg situation, since developers desire its existence, but would rather not be involved in the expensive and time-consuming process of its production. However it is essential to have some sort of standard to ensure a reasonably uniform and compatible set of tools to support the notation. It may be noted that conformance to a standard for a specification notation is somewhat more problematic compared to a programming language since it is inherently non-executable in the general case (otherwise it *would* be a programming language!). There is some dispute about the benefit of so-called 'executable specification languages' and we refer the reader elsewhere for a discussion on this topic [31, 38].

> *By relieving the mind of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race.*
>
> – Alfred North Whitehead

II Thou shalt formalize but not over-formalize. Formal methods shouldn't be employed merely to satisfy company whim, or as a result of peer pressure, as it were. Just as the advent of 'object-orientation' saw many firms incur needless expense as they unnecessarily converted their systems to object-oriented implementations, there is a danger that many will unnecessarily adopt formal methods. Realistically, the first thing that must be determined is that you *really* do need to use formal methods – whether it is for increased confidence in the system, to satisfy a particular standard required by procurers, or to aid in conquering complexity, etc.

Even the most fervent supporters of formal methods must admit that there are areas where formal methods are just not as good as more conventional methods. In User Interface (UI) design, for example, although there have been a number of somewhat successful applications using formal specification techniques [27], it is generally accepted that UI design falls within the domain of informal reasoning.

Applying formal methods to all aspects of a system would be both unnecessary and costly. Even in the development of the CICS system, which resulted in Oxford University Computing Laboratory and IBM being jointly awarded a UK Queen's Award for Technological Achievement, only about one tenth of the system was subjected to formal development. This still resulted in 100,000s of lines of code and thousands of pages of specifications, and having saved 9% over costs using conventional methods (confirmed by independent audit) is often cited as a major application of formal methods [56].

Having determined that one really does need formal methods, and having chosen an appropriate notation and identified those components of the system that will benefit from a formal treatment, one must next consider the level to which formal methods will be employed.

We identify three such levels:

1. Formal Specification.

   The use of formal specification techniques can be of benefit in most cases. A formal language aids in making specifications more concise and less ambiguous, making it easier to reason about them, even at an informal level. The use of such techniques can aid in maintaining levels of abstraction and postponing complexity until a more appropriate juncture.

   They can, in essence, help us to gain greater insights into the system under construction, dispel many ambiguities, and aid in structuring both our approach to the problem and also the resultant implementation.

   Such techniques have proven to be useful in developing a software architecture for a family of oscilloscopes [32], and for such diverse activities as formally specifying the algorithm employed in a single-transferable voting system [52], describing the structure of documents [21, 41] and highlighting inconsistencies in the design of the World Wide Web [59].

2. Formal Development/Verification

   Full formal development is, as yet, rarely undertaken [9]. It involves formally specifying the system under consideration, proving that certain properties are maintained, and that undesirable properties are avoided, or overcome; and finally applying a refinement calculus such that the abstract specification is translated into more and more efficient and concrete representations, with the final representation being executable code.

The proofs involved at this stage may be formal, or informal but rigorous. Each of the applications reported in [40] involves some level of rigorous proof.

3. Machine-checked proofs

   With the advent of support tools, and in particular theorem provers or theorem checkers, mechanically checking proofs for consistency and well-foundedness has become feasible.

   Indeed, for certain classes of system, machine checking of proofs is well worthwhile. Naturally, we would include safety-critical and security-critical systems in this category. In fact, a number of bodies are advocating machine-checked proofs in their standards. The European Space Agency, for example, advocates formal proof (in advance of testing) wherever practicable, and suggests that proofs should be checked independently to reduce the possibility of human error [14]; more and more this is likely to involve machine-checking of proofs.

   A number of formal methods incorporate theorem provers as part of the method itself. In this category we naturally include HOL [35], Larch [36] (with LP, the Larch Prover), Nqthm [18], OBJ [34] and PVS [54]. There are also a number of theorem provers and support environments that incorporate theorem provers for methods such as B [1] (e.g., the B toolkit from B-Core), CSP [43] (e.g., FDR from Formal Systems (Europe) Ltd.), RAISE [58] (from CRI), VDM [45] (e.g., the VDM Toolbox from IFAD) and Z [62] (e.g., Balzac/Zola from Imperial Software Technology, and ProofPower from ICL).

   Quite a lot of interest has also focused lately on tailoring theorem provers for use with specific methods. For example, theorem provers for Z have been developed in EVES [61], HOL [11], and OBJ [48].

Each of these three levels is useful in itself. One must determine, however, whether the additional cost (in time, effort, manpower, tool support, etc.) is justified before embarking on full formal development and machine-checked proofs. For systems where the highest integrity is required — that is, where loss of human life, great financial loss, or mass destruction of property could be the result of a system failure — such an investment might very well be justified ... and required!

> *There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.*
>
> – C.A.R. Hoare

III Thou shalt estimate costs. Formal methods *are* expensive when applied extensively or for the first time. There is quite a learning curve in becoming *au fait* with their effective use, and their initial introduction into a development environment is likely to require significant amounts of training, contract consultancy, and investment in support tools. Set-up costs aside, there is considerable evidence that formal methods projects can run as cheaply (and possibly more cheaply) than projects developed using conventional methods.

For example, evidence of this has been provided by the award of the Queen's Award for Technological Achievement to two formal methods projects in the UK in 1990 and 1992. Auditors checked for the financial benefits gained as part of the award process. In the first, an estimated 12 months reduction in testing time was gained in the development of the Inmos floating-point unit for the T800 Transputer by formally developing the microcode using machine-supported algebraic techniques [49]. In the second case an estimated 9% was saved in the development costs for part of the very large IBM CICS transaction processing system by using the Z notation to respecify the software, resulting in a reduction of errors and an increase in quality of the code produced [44, 56].

The fact that a number of formal methods projects have come in over-budget is not evidence that they are more expensive, but rather that we are, as yet, inexperienced in estimating costs [13].

A number of models have been produced for project cost and project development time estimation. By and large, these have assumed the use of conventional (structured) methods in the development, rather than formal methods, and have based measurements on the number of lines of executable code in the final implementation (a very subjective measure). Although there have been a number of approaches suggested for obtaining metrics from formal specifications (e.g., [65]), these have not as yet been extended to usable models for cost estimation.

As yet, we must rely on models developed before formal methods became widely-used. Perhaps the most famous of these is Boehm's COCOMO model [5], which weights various factors according to the historical results of system development within the organization. The intermediate model augments the basic model, adjusting it with 15 attributes which are seen as key contributors to cost.

Many of these factors will indeed have a significant effect on the cost of developing systems using formal methods. The fact that formal methods are employed in systems where the highest integrity, or reliability, is required, and are likely to be very complex systems, or complex components of larger systems, means that the weightings for *RELY* (Required Software Reliability) and *CPLX* (Product Complexity) are likely to be very high. As formal methods are employed more and more in real-time systems, *TIME* (Execution Time Constraints) is also likely to have a significant influence on costs. The remainder of Boehm's "Computer Attributes" are unlikely to have a significant influence, nor are many of his "Personnel Attributes". In fact, the latter are likely to need to be augmented with new attributes, such as *SEXP* (Specification Language Experience), *MCAP* (Mathematical Capability), *FMEX* (Formal Methods Experience) and *DEXP* (Domain Experience).

While his "Project Attributes" are all likely to remain valid, *MODP* (Modern Programming Practices) is likely to be constant, while the development of more useful tools and support environments [13] should greatly increase the impact of the *TOOL* (Software Tools) attribute. Again, new attributes are likely to be required, such as *DFOR* (the percentage of the system that has been subjected to formal specification techniques and formal analysis) and *PROF* (the degree of rigorous and formal proof required).

One would expect that the use of formal methods would greatly increase the weightings of many of these attributes. It is our contention, however, that this does not mean that formal methods themselves are expensive, but rather is symptomatic of the fact that they are used in high-integrity systems, and it is the systems themselves that are expensive, especially if we require high levels of confidence in their "correct" operation.

Determining the values of these attributes is in itself problematic. The model requires that we determine these from historical information derived from projects conducted in the same environment, other projects conducted within the same organization, and similar projects conducted elsewhere. Even with more traditional development methods, such information is not likely to be easily accessible. With formal methods, there will be even greater difficulties in obtaining this information. We have not yet applied formal methods to a sufficient number of projects to determine trends, and many formal methods projects are in very specialized domains that are unlikely to be addressed very regularly, and are hence very unrepresentative. Greater attention to technology transfer [64] and surveys of formal development [24, 25] will eventually provide us with the levels of detail we require. Elsewhere [40] we attempt to consolidate much of this information in an industrially useful way.

We do not claim that formal methods are cheap, but that for high integrity systems such an investment is warranted and that the returns are sufficient to justify this. We must however be willing to make significant efforts to estimate development lead-times and development costs. Perhaps entirely new cost models are required; but for the time-being extending existing models is a useful starting point, provided that we allow for significant margins of error.

> *The advantages of implicit definition over construction are roughly those of theft over honest toil.*
>
> – Bertrand Russell

IV **Thou shalt have a formal methods guru on call.** The majority of successful formal methods projects to date have had access to at least one consultant who is already expert in the use of formal techniques. It appears to be very difficult to learn to use formal methods successfully without such help until sufficient local expertise has been built up to make this unnecessary. Examples where this has been the case include the IBM CICS project [44] and the Inmos T800 floating point unit for the Transputer [49].

In the case of IBM, the formal methods experts spent months at a time on-site. Training courses were set up and gradually a significant number of people at IBM became fluent in the application of formal techniques. Eventually a critical mass of expertise meant that IBM became self-sufficient in the use of formal methods, no longer requiring continual access to external experts.

At Inmos a rather different approach was adopted. Here the mathematical consultants and the engineers remained as separate pools of expertise. However, sufficient communication between the two groups ensured success. Unless the mathematicians and engineers can each appreciate the role and problems of the other, success will prove elusive. A number of people with the relevant mathematical background and training were hired at Inmos to enable critical parts of designs to be produced and checked formally. Where deemed necessary, external consultancy has still been called upon for specific problems.

Both the above approaches have proved successful and the choice for a particular organization must depend on the style of that organization and its long term aims.

> *Progress will only be achieved in programming if we are willing to temporarily fully ignore the interconnection between our programs (in textual form) and their implementation as executable code ...*
>
> *... In short: for the effective understanding of programs, we must learn to abstract from the existence of computers.*
>
> – Edsger W. Dijkstra

V **Thou shalt not abandon thy traditional development methods.** There is considerable investment in existing software development techniques and it would be foolhardy to replace these *en masse* with formal methods. Instead it is desirable to integrate formal methods into the design process in a cost-effective manner. One way to do this is to investigate how an existing formal method can be combined effectively with an existing structured method already in use within industry. One attempt to do this is the "SAZ" method, a combination of SSADM and Z [57]. Of course structured methods and formal methods each have their strengths and weaknesses and ideally the combination of the two should make the most of the benefits of both. For example, formal methods allow increased precision in a specification, whereas a structured method may be more presentable to a non-expert.

An alternative to integration of techniques is the use of formal methods to review an existing process. It may be possible to provide feedback to a design team using traditional development methods by having a separate team analyze the specification formally early on in the design process, thus catching many errors before they become too expensive to correct. Z has been applied successfully and apparently cost-effectively using this approach [20].

The *Cleanroom* approach is a technique that could easily incorporate the use of existing formal notations to produce highly reliable software by means of non execution-based program development [29]. This technique has been applied very successfully using rigorous software development techniques with a proven track record of reducing errors by a significant factor, in both safety-critical and non-critical applications. The programs are developed separately using informal (often just mental) proofs before they are certified (rather than tested). If too many errors are found, the process rather than the program must be changed. The pragmatic view is that real programs are too large to be formally proven correct, so they must be written correctly in the first place! The possibility of combining Cleanroom techniques and formal methods has been investigated [53].

Sometimes it is possible to combine different formal methods together usefully and effectively. For example, HOL [35] has been used to provide tool support for Z [11]. This allows the more readable Z notation to have the benefit of mechanical proof checking by HOL, thus increasing confidence in the development.

The management of a project using formal methods must be more technically aware than is perhaps normally the case. The use of a formal approach means that code is produced much later on in the design cycle. Far more effort than normal is expended at the specification stage. Many more errors are removed at this point, but early progress might not be as obvious as in a more typical project. One way to provide feedback, particularly for a customer, might be to produce a rapid prototype from the specification [19].

> *But two permissible and correct models of the same external objects may yet differ in respect of appropriateness.*
>
> – Heinrich Hertz

VI **Thou shalt document sufficiently.** An important part of a designed system is its documentation, particularly if subsequent changes are required. Formalizing the documentation leads to less ambiguity and thus less likelihood of errors. In the case of safety-critical systems, timing issues become significant and methods for documenting these are especially important [55].

Formal methods provide a precise and unambiguous way of recording the expected and delivered system functionality, and can therefore be used as a powerful documentation aid. The normal expectation would be that the system documentation contains both the requirements and the system specification in a suitable formal notation, accompanied where appropriate with natural language narrative. The latter is particularly important for conveying information on system aspects which are not formally specified for various reasons.

In general it is highly recommended to produce an informal specification to explain the formal description [7]. This reinforces the reader's understanding of the formal text and connects it with the real world. If there is any discrepancy between the two, the formal specification should be taken as the final arbiter for the documentation since this is the less ambiguous of the two descriptions.

Having formal documentation could be of great benefit when the software needs to be maintained. In the future, it could be possible to maintain the formal description rather than the executable code directly, only undertaking redevelopment of the parts of the code that the modifications necessitate [10].

> *You should not put too much trust in any unproved conjecture, even if it has been propounded by a great authority, even if it has been propounded by yourself. You should try to prove it or disprove it ...*
>
> – George Polya

VII Thou shalt not compromise thy quality standards. Up until relatively recently there have been few standards concerned specifically with software where formal methods are particularly applicable, such as in safety-critical systems. Often software quality standards such as the ISO9000 series have been used instead since these were the nearest relevant guidelines. Now a spate of standards in this area have been, or are about to be, issued [8, 14]. Some of these are recommending or even mandating the use of formal methods. These are not the only standards that need to be adhered to, however.

There is a grave danger that developers will look on the application of formal methods as a means of developing *correct* software. On the contrary, and as we will discuss further later, they are merely a means of achieving higher integrity systems *when applied appropriately.*

There is nothing magical about formal methods, and the organization must ensure that it continues to satisfy its quality standards. This includes ensuring appropriate feedback between development teams and management; ensuring continuity of software inspection and walk-throughs; developing, expanding and maintaining testing policies; and ensuring that system documentation meets the quality standards that were set for conventional development methods.

> *Have nothing in your houses that you do not know to be useful, or believe to be beautiful.*
>
> – William Morris

VIII Thou shalt not be dogmatic. Formal methods are not a panacea; they are just one of a number of techniques that when applied correctly have been demonstrated to result in systems of the highest integrity, and one should not dismiss other methods entirely. Formal methods are no guarantee of correctness; they are applied by humans, who are obviously prone to error. Various support tools such as specification editors, type-checkers, consistency checkers and proof checkers should indeed reduce the likelihood of human error ... but not eliminate it. System development is a human activity, and always will be. Software engineering will be prone to human whim, indecision, the ambiguity of natural language, and simple carelessness.

One can never have absolute correctness, and to suggest that one can is ludicrous. Ongoing debates in *Communications of the ACM* [30] and other fora, have been criticized on the grounds that there is a mismatch between the mathematical model and reality [3]. This is no great deduction — no proponent of formal methods would ever make a claim of definitive correctness. In fact, one should never speak arbitrarily of correctness, but rather correctness *with respect to the specification.* As such, an implementation may be proven to be correct with respect to the specification derived at the outset, but if the specification was not what the procurers really intended, then their (albeit subjective) view will be that the system is incorrect.

One must be conscious of the need to communicate with procurers and the systems users; and one should not be afraid to admit that the specification was not what was intended, and to go back and rework portions of it. System development is by no means a straight-forward one-pass process. Royce's 'Waterfall' model [60] of system development was abandoned because of the simplistic view it held of system development. Every developer has experienced the need to revisit requirements and to rework the specification at various stages in the development. Ideally all inconsistencies will be discovered during implementation, or at worst during post-implementation testing. However, in extreme cases, errors in the system specification may be uncovered during post-implementation execution.

System development is not so simple as the model proposed by Royce [33, 50], but rather an iterative and non-linear process as exemplified by Boehm's 'Spriral' model [6]. As such, the developer should not make claims to having determined all of the requirements just because a certain stage in the development process has been reached; indeed such claims should be considered dubious even post-implementation. The developer must always be ready to make changes to the specification to meet the procurer's requirements; after all, in the best traditions of Roland H. Macy, 'the customer is always right'[1]. Even if the requirements have been fully satisfied, there are still plenty of opportunities for error.

One must always be conscious of the level of abstraction. If one is too abstract, then it is difficult to determine omissions and to determine what the system really is intended to do. If one is not sufficiently abstract, however, there is a tendency, or *bias* towards particular implementations. Couching the specification at the appropriate level of abstraction is a matter of experience — experience with both the specification language and the application domain. One should never be afraid to admit that the level of abstraction is not the most appropriate and to rework the specification accordingly.

Similarly, no proof should be taken as definitive. Hand-proofs are notorious in not only admitting errors as one moves from one line to the next, but also at making gigantic leaps which are unfounded. Even the use of a proof checker does not guarantee the correctness of a proof, but it does aid in highlighting unsubstantiated jumps, and avoidable errors.

> *Errors are not in the art but in the artificers.*
>
> – Sir Isaac Newton

IX **Thou shalt test, test, and test again.** Dijkstra [26] has pointed out a major limitation of testing — while it can demonstrate the presense of 'bugs', it cannot demonstrate their absence. Just because a system has passed unit and system testing, it does not follow that the system will necessarily be bug-free.

That is where formal methods offer considerable advantages over more traditional methods when developing systems where the highest integrity is required. Formal methods allow us to propose properties of the system and to demonstrate that they hold. They allow us to examine system behavior and to convince ourselves that all possibilities have been anticipated. Finally, they enable us to prove the conformance of an implementation with its specification.

In this way, one would hope to eliminate the ubiquitous bug. Unfortunately, contrary to the hyperbolic claims made by many so-called 'experts', formal methods are no guarantee

[1] Of course, in the best traditions of P.T. Barnum, 'there's a sucker born every minute'.

of correctness. Certainly the use of formal methods can give increased confidence in the integrity of the system, and increased confidence that the system will indeed perform as expected, but errors still exist and bugs are still found post-implementation.

Even where full formal development is employed (i.e., the specification is refined to executable code) there must be a certain degree of human input. It is debatable as to whether automatic refinement can ever realistically be achieved, and indeed whether it ever *should* be achieved.

A formal specification is an abstract representation of reality. It has an infinite number of potential implementations. However, when we turn from the abstract world of sets, sequences and formal logic to considering an implementation in a conventional programming language, we find that very few programming languages support the required structures explicitly (and certainly not in an efficient manner). We must then determine the most appropriate data structures to implement the higher level entities (data refinement) and translate the operations already defined to operate on pointers, arrays, records, etc. If a computer program is allowed to choose the eventual implementation structures (assuming that it could be relied upon to choose these appropriately), it will cause a bias towards particular implementations ... one of the things that should be avoided if possible to give the implementor the greatest possible freedom of choice in the design. As such, refinement will always require a certain degree of human input, admitting possibilities of human error.

Even when formal methods are used in the design process, testing, at both the unit and system level, should never be completely abandoned. On the contrary, a comprehensive testing policy should be employed to trap those errors that have been admitted during refinement and/or cases that have not been considered earlier. Although such testing would not need to be as exhaustive as in the case where formal methods had not been employed, a substantial degree of testing is still required.

In the case of the formally developed Inmos floating-point unit for the T800 Transputer, one error *was* found by testing. This was as a result of an "obviously correct" change to the microcode being made after the formal development had been undertaken. We should never underestimate human fallibility, and testing will always be a useful check that a formally produced system does work in the real world.

Testing may be performed in a traditional fashion, using techniques such as McCabe's Complexity Measure to determine the required amount of testing. Alternatively it may employ some form of simulation, using executable specification languages, or some form of specification animation [19, 39].

Formal methods offer yet another alternative when it comes to testing, namely specification-based testing. The formal specification may be used as a guide for determining functional tests for the system. The tester may exploit the abstraction made in the specification to concentrate on the key aspects of the functionality. The approach offers a structured means of testing, which simplifies regression testing [23] and helps to pin-point errors.

The specification itself can be used to derive expected results of test data, and to aid in determining tests in parallel with the design and implementation, hence enabling unit-testing at an earlier stage in the development (which should aid in reducing system maintenance costs).

'Look at this mathematician', said the logician. 'He observes that the first ninety-

nine numbers are less than a hundred and infers hence, by what he calls induction, that all numbers are less than a hundred'

'A Physicist believes', said the mathematician, 'that 60 is divisible by all numbers. He observes that 60 is divisible by 1, 2, 3, 4, 5 and 6. He examines a few more cases, as 10, 20 and 30, taken at random as he says. Since 60 is divisible also by these, he considers the experimental evidence sufficient.'

'Yes, but look at the engineer', said the physicist. 'An engineer suspected that all odd numbers are prime numbers. At any rate, 1 can be considered as a prime number, he argued. Then there comes 3, 5 and 7, all indubitably primes. Then there comes 9; an awkward case, it does not seem to be a prime number, Yet 11 and 13 are certainly primes. "Coming back to 9", he said, "I conclude that 9 must be an experimental error".'

– George Polya

X Thou shalt reuse. The programming phase of system development is actually a minor contributor to system development costs, and is quickly being out-weighed by system maintenance costs. Rising costs of software development can be significantly offset by exploiting software reuse (including code, specifications, designs and documentation). This applies to formal development as well as to more conventional development methods; indeed, exploiting reuse in formal development can (theoretically at least) aid in offsetting some of the set-up costs (e.g., tools, training and education) of the development.

Studies quoted by Capers Jones [22] claim that in 1983 only about 15% of all new code was unique, novel and specific to the individual applications. The remaining 85%, it was claimed, was common and generic, and theoretically could have been rewritten from reusable components.

There are four major factors which conspire against software reuse, however:

1. The VLSR Problem

   The VLSR (Very Large Scale Reuse) Problem [4] holds that the cost of developing an architectural superstructure to support the composition of components is prohibitive when compared to the potential savings to be gained from reuse.

2. Generality versus Specialization

   Smaller components tend to have a more general applicability; larger units tend to be more specialized and less likely to be reusable. But, the larger the component, the greater the payoff, and a seemingly endless dichotomy exists.

3. Cost of Library Population

   Determining the components of programs that are suitable for inclusion in a library tends to be very time-consuming, yet essential if reuse is to be exploited. Having propagated a library of reusable components, one is still faced with the question of how suitable components can be identified for future reuse.

4. The NIH Syndrome

   The Not-Invented-Here Syndrome holds that components reused from previous developments cannot be relied upon to work as anticipated, to satisfy the organization's quality control, and to be sufficiently understood so as to be exploited in new systems.

The use of formal methods in system development can help to overcome each of these problems, and should aid the promotion of software reuse.

Formal methods (or formal specification languages, specifically) provide a means of unambiguously stating the requirements of a system, or of a system component. In this way, formally specified system components that meet the requirements of components of the new system can easily be identified. Thus components that have been formally specified *and sufficiently well documented* can be identified, reused and combined to form components of the new system. Library population costs are not eliminated, but substantially reduced, and confidence in the integrity of the components is greatly increased, as each component is unambiguously specified, and can have various properties about it proposed and proven.

It is important however not just to focus on the reuse of code that has been developed using a formal approach, but rather to reuse the formal specifications themselves also. Such reuse of specifications should help to overcome the generality versus specialization trade-off. Formal specifications are written at a high level of abstraction with (ideally) no bias towards particular implementations. It is during the refinement process that we translate abstract specifications into more and more concrete representations, ending with a representation that can be executed in a programming language. Reusing specifications rather than source code admits the possibility of many different implementations in many different environments, with the most appropriate implementation chosen for the environment in question. In this way, even large components (which offer greater pay-offs) can be made very general and reusable.

Even code that was previously written (using informal development methods) can be reused without compromising the formal development itself. Techniques have been investigated for the reverse engineering of dusty-deck (mainly COBOL) programs to a formal specification and other associated documentation using an interactive tool-based approach which can then be redeveloped into a better structured more understandable program [10]. These have been successfully applied to programs of the order of tens of thousands of lines long. Once this process has been undertaken once, it is possible to maintain the formal specification as opposed to just the program code, so that the two may be kept in line with each other.

> ... *A method was devised of what was technically designated* backing the cards *in certain groups according to certain laws. The object of this extension is to secure the possibility of bringing any particular card or set of cards into use* any number of times successively *in the solution of one problem* ....
>
> – Augusta Ada Lovelace

3 Conclusions

We have attempted to provide some guidelines to help ensure the successful application of formal methods in an industrial context.

It is important to have up-to-date information to hand when deciding which formal method to use. There are a plethora of notations and methods from which to choose, although the number which have been used in an industrial setting is considerably smaller [2].

Chosing an appropriate notation (or notations) and integrating it (them) with existing development processes, being careful to ensure that existing guidelines and procedures are retained as much as possible, is vital for the successful industrialization of formal methods, and to ensure the success of any given formal methods project.

One must always consider that software engineering is a human activity, and that formal methods are no guarantee of correctness. However, if we are willing to keep our own limitations in mind, to recognize these, to learn from our own mistakes and the mistakes of others; and, if we are willing to exploit existing best practice, and to check our work both through appropriate testing and using automated tools, then we can successfully use formal methods in the development of industrial-scale high-integrity systems.

Acknowledgements

The authors are grateful to Robert France and Larry Paulson for comments on an earlier draft of this paper, and to Andy Lambert for drawing the cartoon to our (informal) specification.

References

[1] Abrial, J.-R.: *Assigning Meanings to Programs.* Prentice Hall International Series in Computer Science, to appear.

[2] Austin, S. & Parkin, G.I.: *Formal Methods: A Survey.* National Physical Laboratory, Queens Road, Teddington, Middlesex, TW11 0LW, UK, March 1993.

[3] Barwise, J.: Mathematical Proofs of Computer System Correctness. *Notices of the American Mathematical Society*, 36(7):844–851, September 1989.

[4] Biggerstaff, T.J. & Perlis, A.J. (eds.): *Software Reusability Vol. 1: Concepts and Models*, preface. ACM Press, 1989.

[5] Boehm, B.W.: *Software Engineering Economics.* Prentice Hall, 1981.

[6] Boehm, B.W.: A Spiral Model of Software Development and Maintenance. *IEEE Computer*, 21(5):61–72, May 1988.

[7] Bowen, J.P.: Formal Specification in Z as a Design and Documentation Tool. In *Proc. Second IEE/BCS Conference, Software Engineering 88*, Liverpool, UK, 11–15 July 1988. IEE Conference Publication No. 290, 1988, pp 164–168.

[8] Bowen, J.P.: Formal Methods in Safety-Critical Standards. In *Proc. Software Engineering Standards Symposium (SESS'93)*, Brighton, UK, 30 August – 3 September 1993. IEEE Computer Society Press, 1993, pp 168–177.

[9] Bowen, J.P. (ed.): *Towards Verified Systems*. Elsevier, Real-Time Safety-Critical Systems series, 1994.

[10] Bowen, J.P., Breuer, P.T. & Lano, K.C. Formal Specifications in Software Maintenance: From code to Z^{++} and back again. *Information and Software Technology*, 35(11/12):679–690, November/December 1993.

[11] Bowen, J.P. & Gordon, M.J.C.: Z and HOL. In [12], pp 141–167.

[12] Bowen, J.P. & Hall, J.A. (eds.): *Z User Workshop, Cambridge 1994*. Springer-Verlag, Workshops in Computing, 1994.

[13] Bowen, J.P. & Hinchey, M.G.: Seven More Myths of Formal Methods. Technical Report PRG-TR-7-94, Programming Research Group, Oxford University Computing Laboratory, June 1994. To appear in a shortened form in *Proceedings FME'94 Symposium*, Springer-Verlag LNCS, 1994.

[14] Bowen, J.P. & Hinchey, M.G.: Formal Methods and Safety-Critical Standards. *IEEE Computer*, August 1994.

[15] Bowen, J.P. & Stavridou, V.: The Industrial Take-up of Formal Methods in Safety-Critical and Other Areas: A Perspective. In [67], pp 183–195.

[16] Bowen, J.P. & Stavridou, V.: Safety-Critical Systems, Formal Methods and Standards. *Software Engineering Journal*, 8(4):189–209, July 1993.

[17] Bowen, J.P. & Stavridou, V.: Formal Methods: Epideictic or Apodeictic? *Software Engineering Journal*, 9(1):2, January 1994.

[18] Boyer, R.S. & Moore, J.S.: *A Computational Logic Handbook*. Academic Press, 1988.

[19] Breuer, P.T. & Bowen, J.P.: Towards Correct Executable Semantics for Z. In [12], pp 185–209.

[20] Aujla, S., Bryant, A. & Semmens, L.: A Rigorous Review Technique: Using Formal Notations within Conventional Development Methods. In *Proc. Software Engineering Standards Symposium (SESS'93)*, Brighton, UK, 30 August – 3 September 1993. IEEE Computer Society Press, 1993, pp 247–255.

[21] Cahill, A., Hinchey, M.G. & Relihan, L.: Documents are Programs. In *Proceedings ACM SIGDOC'93, 11th International Conference on System Documentation*, Waterloo, Canada, 5–8 October 1993, ACM Press, New York, 1993, pp 43–55.

[22] Capers Jones, T.: Reusability in Programming: A Survey of the State of the Art. *IEEE Transactions on Software Engineering*, September 1984, pp 488–494.

[23] Carrington, D. & Stocks, P.: A Tale of Two Paradigms: Formal Methods and Software Testing. In [12], pp 51–68.

[24] Craigen, D., Gerhart, S. & Ralston, T.: *An International Survey of Industrial Applications of Formal Methods.* Atomic Energy Control Board of Canada, U.S. National Institute of Standards and Technology, and U.S. Naval Research Laboratories, NIST GCR 93/626. National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161, USA, 1993.

[25] Craigen, D., Gerhart, S. & Ralston, T.: Applications of Formal Methods: Observations and Trends. In [40].

[26] Dijkstra, E.W.: Why Correctness must be a Mathematical Concern. In Boyer R.S. & Moore, J.S. (eds.), *The Correctness Problem in Computer Science*, Academic Press, 1981.

[27] Dix, A.: *Formal Methods for Interactive Systems.* Academic Press, Computers and People Series, 1991.

[28] Draper, C.: Practical Experiences of Z and SSADM. In Bowen, J.P. & Nicholls, J.E. (eds.), *Z User Workshop, London 1992*, Springer-Verlag, Workshops in Computing, 1993, pp 240–254.

[29] Dyer, M.: *The Cleanroom Approach to Quality Software Development*, John Wiley & Sons, Series in Software Engineering Practice, 1992.

[30] Fetzer, J.H.: Program Verification: The Very Idea. *Communications of the ACM*, 31:1048–1063, 1988.

[31] Fuchs, N.E.: Specifications are (Preferably) Executable, *IEE/BCS Software Engineering Journal.* 7(5):323–334, September 1992.

[32] Garlan, D. & Delisle, N.: Formal Development of a Software Architecture for a Family of Instrumentation Systems. In [40].

[33] Gladden, G.R.: Stop the Life Cycle – I Want to Get Off. *ACM Software Engineering Notes*, 7(2):35–39, 1982.

[34] Goguen, J.A. & Winkler, T.: Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI, Menlo Park, USA, August 1988.

[35] Gordon, M.J.C. & Melham, T.F. (eds.): *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[36] Guttag, J.V. & Horning, J.J: *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, Texts and Monographs in Computer Science, 1993.

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

[38] Hayes I.J. & Jones, C.B.: Specifications are not (Necessarily) Executable. *IEE/BCS Software Engineering Journal*, 4(6):330–338, November 1989.

[39] Hinchey, M.G.: Towards Visual Methods. Submitted for publication.

[40] Hinchey, M.G. & Bowen, J.P. (eds.): *Applications of Formal Methods*. Prentice Hall International Series in Computer Science, 1995. In preparation.

[41] Hinchey, M.G. & Cahill, A: Towards a Canonical Specification of Document Structures. In *Proceedings ACM SIGDOC'92, 10th International Conference on System Documentation*, Ottawa, Canada, 13–16 October 1992. ACM Press, New York, 1992, pp 297–307.

[42] Hinchey, M.G. & Jarvis, S.A.: *Concurrent Systems: Formal Development in CSP*. McGraw-Hill International Series in Software Engineering, In press.

[43] Hoare, C.A.R.: *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985.

[44] Houston, I.S.C. & King, S.: CICS Project Report: Experiences and Results from the Use of Z in IBM. In Prehn S. & Toetenel, W.J. (eds.), *VDM'91: Formal Software Development Methods*, Springer-Verlag, LNCS 551, 1991, pp 588–596.

[45] Jones, C.B.: *Systematic Software Development Using VDM*. Prentice Hall International Series in Computer Science, 2nd edition, 1991.

[46] Lamport, L.: *TLZ*. In [12], pp 267–268.

[47] Larsen, P.G., Plat, N. & Toetenel, H.: A Formal Semantics of Data Flow Diagrams, *Formal Aspects of Computing*, to appear.

[48] Martin, A.: Encoding $\mathcal{W}$: A Logic for Z in 2OBJ. In [67], pp 462–481.

[49] May, D., Barrett, G. & Shepherd, D.: Designing Chips that Work. In Hoare, C.A.R. & Gordon, M.J.C., *Mechanized Reasoning and Hardware Design*. Prentice Hall International Series in Computer Science, 1992.

[50] McCracken, D.D. & Jackson, M.A.: Life Cycle Concept Considered Harmful. *ACM Software Engineering Notes*, 7(2):28–32, 1982.

[51] Milner, R.: *Communication and Concurrency*. Prentice Hall International Series in Computer Science, 1989.

[52] Mukherjee, P. & Wichmann, B.A.: Formal Specification of the STV Algorithm. In [40].

[53] Normington, G.: Cleanroom and Z. In Bowen, J.P. & Nicholls, J.E. (eds.), *Z User Workshop, London 1992*, Springer-Verlag, Workshops in Computing, 1993, pp 281–293.

[54] Owre, S., Rushby, J.M. and Shankar, N.: PVS: A Prototype Verification System. In Kapur, D. (ed.), *Automated Deduction – CADE-11*, Springer-Verlag, LNAI 607, 1992, pp 748–752.

[55] Parnas, D.L. & Madey, J.: *Functional Documentation for Computer Systems Engineering*. Version 2, CRL Report No. 237, TRIO, Communications Research Laboratory, Faculty of Engineering, McMaster University, Hamilton, Ontario, Canada L8S 4K1, September 1991.

[56] Phillips, M.: CICS/ESA 3.1 Experiences. In Nicholls, J.E. (ed.), *Z User Workshop, Oxford 1989*, Springer-Verlag, Workshops in Computing, 1990, pp 179–185.

[57] Polack, F. & Mander, K.C.: Software Quality Assurance using the SAZ Method. In [12], pp 230–249.

[58] The RAISE Language Group: *The RAISE Specification Language*. Prentice Hall, BCS Practitioner Series, 1992.

[59] Relihan, L., Cahill, A., & Hinchey, M.G.: Untangling the (World-Wide) Web. In *Proc. SIGDOC'94*, Banff, Canada, October 1994, to appear.

[60] Royce, W.W.: Managing the Development of Large Software Systems. In *Proc. WESTCON'70*, August 1970. Reprinted in *Proc. 9th International Conference on Software Engineering*, IEEE Press, 1987.

[61] Saaltink, M.: Z and Eves. In Nicholls, J.E. (ed.), *Z User Workshop, York 1991*, Springer-Verlag, Workshops in Computing, 1992, pp 233–242.

[62] Spivey, J.M.: *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[63] Toetenel, H.: VDM + CCS + Time = MOSCA. In *18th IFAC/IFIP Workshop on Real-Time Programming – WRTP'92*, Bruges, Belgium, 23–26 June 1992. Pergamon Press, 1992.

[64] Weber-Wulff, D.: Selling Formal Methods to Industry. In [67], pp 671–678.

[65] Whitty, R. Structural Metrics for Z Specifications. In Nicholls, J.E., *Z Users Meeting, Oxford 1989*, Springer-Verlag, Workshops in Computing, 1990, pp 186–191.

[66] Wing, J.M: A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23(9):8–24, September 1990.

[67] Woodcock, J.C.P. & Larsen, P.G. (eds.): *FME'93: Industrial-Strength Formal Methods*. Springer-Verlag, LNCS 670, 1993.

[68] Woodcock, J.C.P. & Morgan, C.C.: Refinement of State-based Concurrent Systems. In Bjørner, D., Hoare, C.A.R. & Langmaack, H. (eds.), *VDM and Z – Formal Methods in Software Development*. Springer-Verlag, LNCS 428, 1990, pp 340–351.

‘Can’t I just read your URL?’

For readers with access to the World-Wide Web global hypermedia system on the Internet, the following WWW page may be of interest:

`http://www.comlab.ox.ac.uk/archive/formal-methods.html`

This provides hyperlinks to many on-line repositories of information relevant to formal methods, including some freely available tools, around the world.