Ten Commandments of Formal Methods

Jonathan P. Bowen
Oxford University

Michael G. Hinchey
University of Cambridge

Formal methods permit more precise specifications and earlier error detection. Software developers who want to benefit from formal methods would be wise to heed these ten guidelines.

> 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*

Producing correct, reliable software in systems of ever increasing complexity is a problem with no immediate end in sight. The software industry suffers from a plague of bugs on a near-biblical scale. One promising technique in alleviating this problem is the application of formal methods that provide a rigorous mathematical basis to software development. When correctly applied, formal methods produce systems of the highest integrity and thus are especially recommended for security- and safety-critical systems.[1-3] Unfortunately, although projects based on formal methods are proliferating, the use of these methods is still more the exception than the rule,[4] which results from many misconceptions regarding their costs, difficulties, and payoffs.[5,6]

Surveys of formal methods applied to large problems[7-9] in industry help dispel these misconceptions and show that formal methods projects can be completed on schedule and within budget. Moreover, these surveys show that formal methods projects produce correct software (and hardware) that is well structured, maintainable, and satisfies customer requirements. Representative case studies explain this in detail.[10]

The subjective question "What makes a formal methods project successful?" cannot be definitively answered. However, through observations of many recently completed and in-progress projects—successful and otherwise—we've come up with ten "commandments" that, if adhered to, will greatly increase a project's chances for success.

I THOU SHALT CHOOSE AN APPROPRIATE NOTATION.

The specification language is the specifier's primary tool during the initial stages of system development. To be consistent with the principles of formal methods, the notation should have a well-defined formal semantics.

Choosing the most appropriate notation is a significant task. Each of the myriad specification languages available claims superiority, and many of these claims are valid. Different specification languages excel when used with particular system classes.

There's always a trade-off between the expressiveness of a specification language and the levels of abstraction it supports. Certain languages may have wider "vocabularies" and more constructs to support the specific situations we want to deal with, but they also force us toward specific implementations. Although these languages will shorten the specification, they also reduce its abstraction.

Languages with small vocabularies, on the other hand, generally result in longer specifications, offering high levels of abstraction and little implementation bias. For example, consider Hoare's language, Communicating

0018-9162/95/$4.00 © 1995 IEEE

Sequential Processes. CSP's only first-class entities are processes (or pipes and buffers, which are particular process types). CSP specifications can thus become lengthy, but they're easier to understand due to the small number of constructs. Likewise, there's no bias toward implementing communication primitives—CSP channels may be implemented as physical wires, buses, mailboxes, or even shared variables.

Vocabulary is not the only consideration. Some specification languages just aren't as good as others when used with particular system classes. Trying to specify a concurrent system in a model-based specification language, such as Z or VDM, is like using a hammer to insert a screw. A process algebra, such as CSP or Milner's Calculus of Communicating Systems, is much more appropriate; however, the drawback is an inattention to system state aspects. The situation has prompted research to integrate process algebras with model-based specification languages and to extend these languages to handle concurrency and temporal aspects.

"Can't I just read your URL?"

To achieve commercial success, a notation should be well established with a solid user base. Typically, a formal notation developed for industrial use takes at least a decade, from conception to application. It takes this long for the notation to be developed, taught, promulgated, documented, and marketed. And building the requisite user community has many tasks associated with it: writing textbooks, developing courses, fostering industrial-academic liaisons, and developing and marketing support tools. The technology transfer of formal notations, as with many new developments, involves many potentially troublesome hurdles.

An international standard increases the likelihood that a notation will win general acceptance among project developers. This is a chicken-and-egg situation, since developers want the standard but not the expense and time associated with producing it. However, a standard is essential to ensure support-tool uniformity and compatibility. Then, too, conforming to a specification notation standard is more difficult than conforming to a programming language standard because a notation is inherently nonexecutable.

> 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 OVERFORMALIZE.

Just as many companies needlessly converted systems to object-oriented implementations, companies may unnecessarily adopt formal methods, either to satisfy corporate whim or peer pressure. First, determine that formal methods are really needed—whether to increase system confidence, to satisfy a customer-imposed standard, or to reduce complexity, for example.

Even the most fervent formal methods advocates must admit that, occasionally, formal methods fall short of more conventional methods. For example, although some user-interface designs have successfully incorporated formal specification techniques, it's generally accepted that user interface design conformance resides in the domain of informal reasoning.

Applying formal methods to all system aspects is both unnecessary and costly. During development of the Customer Information Control System (CICS), a joint Oxford University Computing Laboratory/IBM effort, scarcely one tenth of the system underwent formal development. The project generated over 100,000 lines of code and thousands of specification pages. The project, having saved nine percent over costs using conventional methods (confirmed by independent audit) is often cited as a major, successful application of formal methods.

When it's been determined that formal methods are required, an appropriate notation should be selected and system components identified that will benefit from a formal treatment. Then, the level to which formal methods will be employed must be considered.

We identify three such levels.

Formal specification

Formal specification techniques are generally beneficial because a formal language makes specifications more concise and explicit. These techniques help us acquire greater insights into the system design, dispel ambiguities, maintain abstraction levels, and determine both our approach to the problem as well as its implementation.

Such techniques have proved useful in developing, for example, a software architecture for a product line of oscilloscopes. They've also been beneficial in specifying the algorithm in a single-transferable voting system, describing document structure, and highlighting inconsistencies in the World Wide Web design.

Formal development/verification

To date, full formal development is rarely attempted. It involves formally specifying the system, proving that certain properties are maintained while others are avoided or overcome, and applying a refinement calculus to translate the abstract specification into progressively more efficient, concrete representations. The final representation, of course, is executable code.

The proofs involved at this level can be formal, or they can be informal but rigorous. Applications using different levels of rigorous proofs are discussed elsewhere.[10]

Machine-checked proofs

Support tools, particularly theorem provers or theorem checkers, have allowed machine-checked proofs for consistency and integrity. Mechanical proof checking is especially important for safety- and security-critical systems; in fact, numerous standards bodies are officially advocating this technique. The European Space Agency, for example, advocates formal proof in advance of testing wherever feasible. The agency also suggests that proofs be checked independently to reduce possible human error;[3] increasingly, this is likely to involve mechanical proof checking.

The fact that many formal methods projects have exceeded their budgets proves not that the methods are more expensive but that cost estimation techniques need improvement.

Several formal methods incorporate theorem provers (for example, HOL, Larch, Nqthm, OBJ, and PVS). There are also a number of theorem provers and support environments that incorporate theorem provers for methods such as B, CSP, FDR, RAISE, VDM, and Z. Lately, there's been interest in tailoring theorem provers to specific methods. For example, theorem provers for Z have been developed in EVES, HOL, and OBJ.

Each level is useful in itself. Full formal development and machine-checked proofs, however, cost additional time, effort, manpower, and tool support. Such an investment might be *required*, however, for high-integrity systems where failure could cause loss of life, great financial loss, or massive property destruction.

> 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.

There's a steep learning curve in becoming an accomplished user, and introducing formal methods into a development environment typically requires significant investment in training, consultancy, and support tools. Set-up costs aside, however, there is considerable evidence that formal methods projects can run at least as cheaply as projects developed with conventional methods.

Savings achieved with formal methods was evidenced a couple of years ago by two formal methods projects in the UK that received the Queen's Award for Technological Achievement. In the first project, test time was reduced by an estimated 12 months for the T800 Transputer's Inmos floating-point unit through formally developed microcode using machine-supported algebraic techniques. In the second, an estimated nine percent of development costs was saved for part of the earlier mentioned IBM CICS system's development by respecifying the software with the Z notation. This also reduced errors and improved code quality. In both cases, formal methods resulted in saving millions of dollars.

The fact that many formal methods projects have exceeded their budgets proves not that the methods are more expensive but that cost estimation techniques need improvement.[6]

Project-cost and development-time estimation models have been produced based on conventional structured, rather than formal, development methods. These models typically base costs on a measure of the number of lines of executable code produced in the final implementation, a subjective measure at best. Although several approaches to obtain metrics from formal specifications have been suggested, these have not yet been extended to usable models for cost estimation.

For now, we must rely on models predating formal methods. Perhaps the most famous of these is Boehm's Constructive Cost Model (Cocomo), which weights various factors according to historical results. The intermediate model augments the basic one with 15 cost-contributing factors categorized as computer, personnel, or project attributes.

These attributes will significantly affect the cost of formally developed systems. When formal methods are applied to complex systems or subsystems requiring the highest integrity and reliability, the weightings for two computer attributes—RELY (required software reliability) and CPLX (product complexity)—are likely to be very high. As formal methods are increasingly employed in real-time systems, TIME (execution time constraints) will significantly influence costs.

Boehm's other computer attributes, like most of his personnel attributes, probably will not significantly affect cost. In fact, additional new personnel attributes can be expected, such as SEXP (specification language experience), MCAP (mathematical capability), FMEX (formal methods experience) and DEXP (domain experience).

Of Boehm's project attributes, MODP (modern programming practices) is likely to be constant, while the development of more useful tools and support environments[6] should enhance the TOOL (software tools) attribute. Again, new attributes are likely to be required, such as DFOR (percentage of the system that has been subjected to formal specification techniques and formal analysis), and PROF (degree of rigorous and formal proof required).

One might expect that formal methods would greatly increase attribute weightings and, by extension, system cost. However, it's not the methods but the systems where they are used that are expensive: High-integrity systems are intrinsically expensive, especially if high confidence levels and a fail-safe operation are required.

Determining the values of these attributes is problematic. The Cocomo model requires that we determine them from historical data derived from other projects, both sim-

ilar and dissimilar. Such information is difficult enough to obtain; it will be even more difficult with formal methods, due to the relatively few projects that have used them. Moreover, many formal methods projects aren't representative because they're very highly specialized. Technology transfer and formal development surveys[7-9] will eventually supply the needed detail. We have, in fact, already begun to consolidate this information.[10]

For high-integrity systems, the investment in costly formal methods is warranted and will be justified by the anticipated returns, as long as development lead times and development costs are correctly estimated. Entirely new cost models may be required; for now, extensions to existing models are 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.

Most successful formal methods projects have relied on at least one consultant with formal techniques expertise. Such outside expertise appears almost indispensable, as evidenced by the IBM CICS project and the Inmos T800 Transputer project.

In IBM's case, formal methods experts extensively trained employees to become self-sufficient in formal techniques. A different approach was adopted at Inmos, where consultants and project engineers worked in tandem but communicated constantly to ensure success. Inmos also hired people with the relevant mathematical experience to formally produce and check critical designs and consulted with outside experts.

Both approaches have proved successful—a company must choose an approach that reflects its organizational style as well as its long-term goals.

> 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's been considerable investment expended on existing software development techniques, so it would be foolhardy to replace them entirely with formal methods. A better approach would be to cost-effectively integrate formal methods into existing design processes, an example of which is known as the SAZ method, which combines SSADM (Structured Systems Analysis and Design Method) and Z. Ideally, any combination of structured and formal methods would meld the best of each. For example, formal methods enable more precise specifications, whereas structured methods are more presentable to a nonexpert.

An alternative is to have a design team, using traditional development methods, obtain feedback from another team that formally analyzes the specification early in the design process. This effectively catches many errors while it still makes economic sense to correct them. Z has been applied successfully and cost-effectively with this approach.

The Cleanroom approach could easily incorporate existing formal notations to produce highly reliable software through nonexecution-based program development. This technique has a successful track record of significant error reduction in both safety-critical and noncritical applications. The programs are developed separately using informal proofs before they are certified, rather than tested. If too many errors are found, the process (not the program) must be changed. Realistically, most programs are too large to be formally proven, so they should be written correctly in the first place. The possibility of combining Cleanroom techniques and formal methods has been investigated, but applying even semiformal development on an industrial scale is difficult without machine assistance.

Sometimes different formal methods can be effectively combined. For example, HOL has provided tool support for Z, which gives the more readable Z notation the benefit of mechanical proof checking, thus increasing development confidence.

The managers of formal methods projects must be more technically cognizant than usual because, with formal approaches, the specification stage requires far more effort than is customary; consequently, code is produced much later in the design cycle. More errors are thus removed prior to coding, but early progress might not be as obvious as in a more conventional project. One way to assure feedback, particularly for a customer, might be to produce a rapid prototype from the specification.

> 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.

Documentation is important to the design process, particularly if changes will later be required. Formalizing the documentation reduces both ambiguity and errors. With safety-critical systems, for example, documenting timing issues is especially important.

Formal methods are a powerful documentation aid because they can precisely record the system's functionality, both expected and delivered. Normally, the system documentation features the requirements and corresponding specification in a suitable formal notation, accompanied with natural language narrative, where appropriate. Natural language is particularly important for conveying information that has not otherwise been formally specified.

It's highly recommended that an informal specification be produced to explain a formal project description. This reinforces and clarifies the reader's understanding of the formal text. If there is any discrepancy, the formal specification takes precedence because it is the more explicit of the two descriptions.

Formal documentation could also facilitate software

maintenance. Eventually, it may be possible to maintain the formal description rather than the executable code directly, with corresponding changes made to those parts of the code affected by the modifications, rather than changing the code directly so that it becomes out of step with the documentation.

> 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.

To date, there have been few standards concerned specifically with software where formal methods are particularly applicable, as they are with safety-critical systems. Software quality standards, such as the ISO 9000 series, have been applied instead because these were the nearest relevant guidelines. Now many standards that address formal methods have recently been, or soon will be, issued.[1,3] Although some of these recommend or even mandate formal methods, these standards are not the only ones to consider.

There is a clear danger that developers will regard formal methods as a means of developing *correct* software. On the contrary, they are merely a means of achieving higher system integrity *when applied appropriately*.

Formal methods alone won't suffice; an organization must continue 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 just one of many techniques that, when applied correctly, have demonstrably resulted in systems of the highest integrity. Other methods should not be simply dismissed, however, because formal methods cannot guarantee correctness; they are applied by humans, who are obviously error prone. Support tools—such as specification editors, type checkers, consistency checkers, and proof checkers—might reduce the likelihood of human error but will not eliminate it. System development is a human activity and will always be prone to human whim, indecision, the ambiguity of natural language, and simple carelessness.

Absolute correctness is, simply, impossible to achieve. Ongoing debates in *Communications of the ACM* and other forums have been criticized because of a mismatch between the mathematical model and reality. No proponent of formal methods however would claim definitive correctness. In fact, one should speak not of correctness but of correctness *with respect to the specification*. And if an implementation was correct with respect to the specification but the specification was not what the customers intended, then the customers' (albeit subjective) view will be that the system is incorrect.

Communication between developers and customers is essential because system development is not straightforward. In fact, Royce's waterfall model was abandoned precisely because it viewed system development simplistically. System development is iterative and nonlinear, as exemplified by Boehm's spiral model; indeed, some requirements may not be determined even at post-implementation.

The developer must anticipate the need to modify the specification to meet the customer's requirements. Every developer has had to revisit requirements and rework the specification periodically—in the best traditions of Roland H. Macy, "the customer is always right." And, even if the requirements have been fully satisfied, there are still opportunities for error—until, and including, post-implementation execution.

The developer must also consider the right level of abstraction for the specification, which is a matter of experience. Too much abstraction makes it difficult to find omissions and to determine what the system is meant to do; too little causes bias toward particular implementations.

Similarly, no proof should be considered definitive. Manual proofs are notoriously error-prone and illogical. Even a proof checker doesn't guarantee a proof's correctness, but it does highlight 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 has pinpointed a major limitation of testing: It can demonstrate the presence of bugs but not their absence. Just because a system has passed unit and system testing, it does not follow that the system will be bug-free.

In testing high-integrity systems, formal methods offer considerable advantages. For example, they let us demonstrate that proposed system properties actually exist. They let us examine system behavior and convince ourselves that all possibilities have been anticipated. Finally, they let us prove that an implementation conforms to its specification.

For all this, unfortunately, formal methods still cannot guarantee correctness, contrary to the hyperbolic claims made by many so-called experts. Formal methods can increase confidence in both a system's integrity and its expected performance, but bugs are still found, even after implementation.

A formal specification abstractly represents reality, and it has an infinite number of potential implementations. In considering a conventional implementation, however, there are very few programming languages that support the required structures, either explicitly or efficiently. Therefore, determine the most appropriate data structures to implement the higher level entities (this is called data refinement) and translate the operations already defined

to operate on pointers, arrays, and records, for example. If a computer program were to choose the eventual implementation structures, assuming that it could be relied on to do so appropriately, it would cause a bias toward particular implementations. Bias should be avoided to give the implementor maximum design freedom. Refinement will always require a certain degree of human input, along with possibilities of human error.

The error resulted from an "obviously correct" change to the microcode. Never underestimate human fallibility—testing will always be a useful check.

Testing is necessary, even when formal methods are used in design, to isolate any errors that either escaped earlier detection or were caused during refinement. For example, in the case of the T800 Transputer's floating-point unit, one error *was* found by testing. The error resulted from an "obviously correct" change to the microcode after the formal development had begun. Conclusion: Never underestimate human fallibility—testing will always be a useful check.

Testing might be conducted in a traditional manner, using techniques such as McCabe's Complexity Measure to first determine the required amount of testing. Alternatively, testing might employ simulation, using executable specification languages or some form of specification animation.

With formal methods, specification-based testing is yet another alternative to determine functional tests to be run. The specification's abstraction can be exploited to concentrate on key functionality aspects, an approach that offers structured testing, which simplifies regression testing and helps isolate errors.

The specification can also be used to derive expected test results and to identify tests to be run in parallel with design and implementation, which enables earlier unit testing and which should reduce 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, such 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*

THOU SHALT REUSE.

Programming is outweighed by system maintenance when it comes to system development costs. Rising software development costs can be significantly offset by exploiting software reuse, including code, specifications, designs, and documentation. This applies to formal as well as more conventional development methods; theoretically, reuse can offset some set-up costs such as tools, training, and education.

Studies quoted by Capers Jones[11] claim that in 1983 only 15 percent of all new code was unique and application specific. The remaining 85 percent, it was claimed, was common, generic, and in fact could have been rewritten from reusable components.

Four major factors conspire against software reuse:

1. *The very large-scale reuse problem.* The VLSR problem is the prohibitive cost of developing an architectural superstructure to support component composition when compared to the potential savings to be gained from reuse.
2. *Generality versus specialization.* In terms of applicability, smaller components are more general, whereas larger units are more specialized and therefore less likely to be reused. But the larger the component, the greater the payoff, and a seemingly endless dichotomy exists.
3. *Cost of library population.* Determining suitable program components for inclusion in a library is time-consuming, yet essential for reuse. Having propagated a library of reusable components, a developer is still faced with the question of how suitable components can be identified for future reuse.
4. *The not-invented-here syndrome.* The NIH syndrome holds that components reused from previous development projects cannot be relied upon to work as anticipated, to satisfy the organization's quality control, or to be sufficiently understood in new systems.

The use of formal methods in system development can help overcome these problems and promote software reuse.

Formal specification languages clearly state the system requirements, which makes it easy to identify those components that also meet the requirements of a new system's specification. Components that have thus been formally specified and sufficiently well documented can be identified, reused, and combined in a new system. Library population costs are substantially reduced, though not eliminated, and confidence in component integrity is greatly increased.

It's important to focus on the reuse of formally developed specifications as well as formally developed code, as such reuse can ameliorate the generality-versus-specialization trade-off. Formal specifications are written at a high level of abstraction with, ideally, no bias toward particular implementations. During the refinement process abstract specifications are translated into ever more concrete representations, resulting in a representation that can be executed in a programming language. Reusing specifications rather than source code makes possible 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 payoffs, can be made very general and reusable.

What's more, code that was previously written via informal development methods can be reused without compromising the formal development itself. Techniques featuring interactive tools have evolved for reverse-engineering legacy programs (mainly Cobol) in accordance with formal specifications and have, in fact, been successfully applied to programs tens of thousands of lines long. These old programs are then redeveloped for better structure and comprehension. After initially applying this process, it is possible to maintain both the formal specification and the program code, so that the two may be correlated.

> . . . 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*

THESE GUIDELINES WILL HELP ENSURE THAT formal methods can be successfully applied in an industrial context.

It is important for the developer to have up-to-date information about best practice and industrial usage when selecting from the many notations and methods available. (The number of those actually used in an industrial setting is, to date, quite small.) The notation must then be carefully integrated with existing development processes, while retaining existing guidelines and procedures as much as possible.

A few last caveats: Always remember that software engineering is a human activity and that formal methods will not—cannot—guarantee correctness. If we're willing to learn from our mistakes and those of others; if we're willing to exploit existing best practice and to check our work, through appropriate testing and tools, then we can successfully use formal methods to develop high-integrity systems. ■

For more information

Readers with access to the World Wide Web may find the following WWW Uniform Resource Locator (URL) 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.

Acknowledgments

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

Jonathan Bowen is funded by the UK Engineering and Physical Sciences Research Council under grant No. GR/J15186. Mike Hinchey is funded by ICL.

References

1. J.P. Bowen, "Formal Methods in Safety-Critical Standards," *Proc. SESS 93, Software Eng. Standards Symp.*, IEEE CS Press, Los Alamitos, Calif., Order No. 4240, 1993, pp. 168-177.
2. J.P. Bowen and V. Stavridou, "Safety-Critical Systems, Formal Methods, and Standards," *IEE/BCS Software Eng. J.*, Vol. 8, No. 4, July 1993, pp. 189-209.
3. J.P. Bowen and M.G. Hinchey, "Formal Methods and Safety-Critical Standards," *Computer*, Vol. 27, No. 8, Aug. 1994, pp. 68-71.
4. J.P. Bowen and V. Stavridou, "The Industrial Take-up of Formal Methods in Safety-Critical and Other Areas: A Perspective," *Proc. FME 93, First Formal Methods Europe Symp.*, in *Lecture Notes in Computer Science*, J.C.P. Woodcock and P.G. Larsen, eds., Vol. 670, Springer-Verlag, Berlin and London, pp. 183-195.
5. J.A. Hall, "Seven Myths of Formal Methods," *IEEE Software*, Vol. 7, No. 5, Sept. 1990, pp. 11-19.
6. J.P. Bowen and M.G. Hinchey, "Seven More Myths of Formal Methods," Tech. Report 357, University of Cambridge Computer Laboratory, Cambridge, UK, Dec. 1994; also to appear in *IEEE Software*, and published in a shortened format as "Seven More Myths of Formal Methods: Dispelling Industrial Prejudice," in *Proc. FME 94, Second Formal Methods Europe Symp.*, T. Denvir, M. Naftalin, and M. Bertran, eds., *Lecture Notes in Computer Science*, Springer-Verlag, Berlin and London, Vol. 873, 1994, pp. 105-117.
7. D. Craigen, S. Gerhart, and T. Ralston, *An International Survey of Industrial Applications of Formal Methods*, Atomic Energy Control Board of Canada, US National Institute of Standards and Technology, and US Naval Research Laboratories, NIST GCR 93/626, National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161, USA, 1993.
8. D. Craigen, S. Gerhart, and T. Ralston, "Formal Methods Technology Transfer: Impediments and Innovation," in *Applications of Formal Methods*, Prentice Hall Int'l Series in Computer Science, M.G. Hinchey and J.P. Bowen, eds., Prentice Hall, Hemel Hempstead, UK, to be published in 1995.
9. J.P. Bowen and M.G. Hinchey, "Ten Commandments of Formal Methods," Tech. Report No. 350, Sept. 1994, Computer Laboratory, Univ. of Cambridge, Cambridge, UK.
10. M.G. Hinchey and J.P. Bowen, eds., *Applications of Formal Methods*, Prentice Hall Int'l Series in Computer Science, Prentice Hall, Hemel Hempstead, UK, to be published in 1995.
11. T.C. Jones, "Reusability in Programming: A Survey of the State of the Art," *IEEE Trans. Software Eng.*, Vol. 10, No. 5, Sept. 1984, pp. 488-494.

Jonathan P. Bowen *is a senior researcher at the Oxford University Computing Laboratory. He has worked in the field of computing in both industry and academia since 1977. His interests include formal specification, Z, provably correct compilation, rapid prototyping using logic programming, decompilation, hardware compilation, safety-critical systems, and software/hardware codesign. He holds an MA degree in engineering science from Oxford University.*

Bowen won the 1994 IEE Charles Babbage Premium award and currently manages the ESPRIT ProCoS-WG Working Group of 24 European partners. He is chairman of the Z User Group and a member of the IEEE Computer Society, the ACM, and Euromicro.

Michael G. Hinchey *is a researcher with the University of Cambridge Computer Laboratory and a faculty member of*

the Real-Time Computing Laboratory at the New Jersey Institute of Technology. His research interests include formal specification, formal methods, real-time systems, concurrency, method integration, CASE, and visual programming and environments. He has published widely on various aspects of software engineering and is the author-editor of several books on software development and formal methods.

Hinchey is treasurer of the Z User Group and program chair for ZUM 95, and newsletter editor of the IEEE Computer Society Technical Segment Committee on Engineering of Complex Computer Systems. He is a member of the IEEE, the ACM, AMS, and an associate fellow of the Institute of Mathematics.

Readers can contact Jonathan Bowen at the Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK; e-mail jonathan.bowen@comlab. ox.ac.uk, and Mike Hinchey at the University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, UK; e-mail mike.hinchey@cl.cam.ac.uk.