Formal Specification and Verification for Critical Systems: Tools, Achievements, and Prospects

Presented at Electric Power Research Institute (EPRI) Workshop on Methodologies for Cost-Effective, Reliable Software Verification and Validation, Chicago, IL, 7–9 August 1991. Published in EPRI TR-100294, January 1992, pages 9-1 to 9-14. Reprinted in "*Advances in Ultra-Dependable Distributed Systems*," N. Suri, C. J. Walter and M. M. Hugue, Eds., pp. 282–296, IEEE Computer Society, 1995.

Note: this paper is now rather old. A more uptodate treatment of similar topics can be found in "*Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS*" by Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke, IEEE Transactions on Software Engineering, Vol. 21 No. 2 (feb. 1995) pp. 107–125.*

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

Abstract

Formal specification and verification use mathematical techniques to help document, specify, design, analyze, or certify computer software and hardware. Mathematically-based notation can provide specifications that are precise and unambiguous and that can be checked mechanically for certain types of error. Formal verification uses theorem proving techniques to establish consistency between one level of formal specification and another.

This paper describes some of the issues in the design and use of formal specification languages and verification systems, outlines some examples of the application of formal methods to critical systems, and identifies the benefits that may be obtained from this technology.

1 Introduction

Formal specification and verification are examples of what are often called *formal methods* in computer science. And formal methods are simply those that use mathematical techniques to help document, specify, design, analyze, or certify computer software and hardware.

*Available from `http://www.csl.sri.com/tse95.html`

Through formal methods, mathematics can play a role in software and hardware engineering similar to that which it plays in other engineering disciplines: it can provide a precise notation and a conceptual framework for developing and describing designs, and it can allow the behavior of the final system to be predicted through calculations based on its design.

Each engineering discipline draws on particular branches of mathematics: aeronautics draws on fluid dynamics, nuclear engineering on quantum mechanics, and so on. The applied mathematics of computer science is mathematical logic: the notations and conceptual frameworks of formal methods are derived from those of logic, and its calculations are based on formal deduction. The mathematically-based notations used in formal methods are called ***formal specification languages***, and the conceptual framework of logic provides precise interpretations for notions such as what it means for a specification to be internally ***consistent***, for one specification to ***satisfy*** another, and for one specification to ***implement*** another. The calculations performed on formal specifications are generally those that check these relations among specifications—that is, we generally try to prove that one specification implements or satisfies another. This process of proving putative theorems about formal specifications is called ***formal verification***.

As with other engineering disciplines, it may be worthwhile to mechanize the calculations that support use of formal methods in computer science. The expected benefits of mechanization will be the same as in other disciplines: improved speed and accuracy in calculation. Since the calculations in formal verification involve proving theorems about specifications, a formal verification system must include a processor for the specification language concerned, and a mechanical theorem prover or proof checker.

If a formal verification were performed through several levels of design, we might be able to provide a program[1] or hardware circuit that is proven to implement its specified requirements. It might seem that the program is then guaranteed to be correct—indeed, this endeavor is sometimes called "proof of correctness." Unfortunately, however, we cannot equate formal verification with unequivocal correctness. This is because correctness is a property of the real system and of real-world expectations, whereas formal verification is concerned with mathematical models of the system and of its requirements. This distinction between model and reality attends all applications of mathematics in engineering: the "correctness" of an aerodynamic calculation, for example, depends on the fidelity of fluid dynamics to the real behavior of air over a wing, on the veracity of the stated requirements, and on the extent to which the calculations are performed without error.

[1]A program is simply a specification that is sufficiently detailed that it can be interpreted automatically by a machine. The distinction between programs and specifications changes as more powerful programming languages are developed: for example, a Prolog program looks like a specification to an assembly-language programmer.

Each of these limitations on correctness poses significant challenges in formal verification. The behavior of the real system may depend on a programming language whose implementation may not coincide with its assumed semantics, it may involve physical processes that do not admit completely accurate descriptions, or that may be subject to random effects, and the properties required of the system may not be fully understood, let alone fully recorded in its specification. Finally, demonstration of consistency across the hierarchy of specifications of the system will be subject to the errors attendant upon any human enterprise.

These limitations on formal verification are identical to those attending all mathematical modeling in engineering. The limitations are minimized in other engineering disciplines by empirical validation: we believe fluid dynamics accurately models the behavior of air over a wing because it has been validated in practice many times, and the specific predictions in a particular case are generally validated by wind-tunnel experiments and flight tests. The same dual process applies in computer science: formal verification is no more than a formalization of one of the components in the widely practiced software quality assurance process called Verification and Validation (V&V). Validation (empirical testing), the other component to this process, is not made redundant or unnecessary by formalizing the verification component.

Just as formal verification does not do away with the need for testing, neither can testing make verification unnecessary. Unlike most analog systems, the behavior of digital systems is generally not smooth or continuous (indeed, this is the main source of their utility). Thus for digital systems, it is not adequate to test the behavior of the system at representative points of the input space and then assume that its behavior at intermediate points can be interpolated from those actually tested. Instead, empirical validation of digital systems requires enormous numbers of tests, which for the higher levels of reliability (i.e., above about $1 - 10^{-6}$) require an infeasible amount of time.

Rather than attempt to replace verification by validation, or vice-versa, the most sensible and pragmatic approach is surely to improve the effectiveness of each as much as possible, and to maximize the extent to which each supports and compensates for weaknesses of the other. Unlike testing, formal verification covers the entire input space; testing at representative points serves to validate the modeling underlying the verification, so that the combination of verification and validation is stronger than either alone. Formal verification also assists validation by making explicit the specification whose satisfaction is verified, and the assumptions on which the verification rests. It therefore identifies those properties whose satisfaction, or utility must be empirically validated.

Formal verification can also support a form of validation early in the design process when regular testing is infeasible. For example, the specification of a particular set of requirements can be evaluated by posing questions such as "if I've stated this right, then the following ought to be true, and the following ought never to happen." These questions can be formulated as putative theorems about the specification, and formal verification techniques can then be used to probe their truth or falsity. This technique provides a way of testing specifications.

2 Tools to Support Formal Methods

In the introduction we considered the role of formal verification in V&V. Formal methods and tools to support them are often considered in this rather narrow context of V&V—or, even more narrowly, in the context of certification. However, the design of computer software and hardware is an intellectual endeavor that can benefit generally from tools that support this endeavor. Thus, formal methods can do much more than simply support V&V: they can assist in the process of design, allowing decisions to be recorded with precision, allowing alternatives to be evaluated, and consequences to be analyzed. Most importantly, they can help in the early detection of errors, oversights, and inconsistencies, and thereby enhance the overall quality of the product and the cost-effectiveness of the software and hardware engineering process. As before, these benefits are analogous to those conferred by mathematical modeling in other engineering disciplines, where calculations allow design alternatives are explored, and performance to be predicted, prior to construction of the final system.

We will therefore consider formal methods and their tools from a more general perspective than simply their role in V&V; we will consider how they can assist in the process of software and hardware development.

2.1 Specification Languages

At their most elementary, formal methods can mean simply the use of mathematical notation (typically logic and set theory) in otherwise informal system documentation. Algorithms and programs are often presented this way in text books on computer programming. True specification languages take things a step further and provide a self-contained notation for certain aspects of system design. Specification languages range from those that are primarily intended for pencil-and-paper descriptions and analyses, through those that support limited mechanical analysis, to those that support full formal verification.

The major design trade-off that must be made in a general-purpose specification language is that between its convenience and expressiveness as a specification medium, and the automation and effectiveness of the mechanical support that can be provided.

In this trade-off, the extremes are illustrated by the Z notation [29] from Oxford University, and the computational logic of Boyer and Moore [5]. Z is very rich specification notation based on set theory which has been used to document several system designs [16], but for which only elementary mechanical support is available. The Boyer-Moore system, on the other hand, uses a very limited, raw logic as its specification language, but has a very powerful theorem prover that has been used to undertake several large and complete formal verifications [4].

We consider neither of these extremes to be ideal. A specification language is used for communication and should facilitate clear and straightforward expression using familiar notations. In addition, specifications tend to be large, so that mechanisms for structuring them into modules and parameterizing them for re-use are essential. The limited, raw logic of the Boyer-Moore system does not satisfy these desiderata. On the other hand, Z is an attractive specification language with little mechanical support. Proponents of Z argue that provision of advanced mechanical support is simply an engineering issue, and point to a prototype typechecker and theorem prover as precursors to more advanced capabilities. I am not convinced by this. Lack of effective mechanical support may indicate underlying complexity and difficulty in a specification language: that which is hard to implement may be hard to understand and use reliably. Some of the basic concepts employed in Z are simply difficult to deal with (for example, partial functions can require three-valued logics [8], or logics of partial terms [13], or other exotic treatments). In my opinion, those who have not developed full mechanical support for their specification languages underestimate the scale of the challenge, and those that have not done hard verifications underestimate the theorem proving power that is needed.

There is another foundation, different from set theory, that can support a familiar and expressive notation for specification, and that is more amenable to mechanization. This is higher-order logic (also known as the simple theory of types), and it is the basis of the HOL system from Cambridge [7,15] and our own EHDM system [26]. Higher-order logic is very expressive, but a specification language must also provide a carefully selected set of built-in data types and type-constructors. In EHDM, for example, the integer and rational numbers are built in, as are constructors such as arrays, records, tuples, and enumerations. In addition, a form of subtyping is provided. This is convenient for many applications, and essential in order to deal with operations, such as division, without getting embroiled in partial functions. For example, in EHDM, we define the signature of division as $\mathcal{Q} \times \mathcal{Q}_z \rightarrow \mathcal{Q}$ where $\mathcal{Q}$ denotes the rationals and $\mathcal{Q}_z$ denotes the nonzero rationals—a subtype of the rationals. We allow a term of a supertype to appear where one of a subtype is required, provided the term can be proved to satisfy the defining predicate of the subtype concerned. For example, an expression such as $\frac{\rho}{n-m}$ is acceptable in EHDM, provided we can prove $n \neq m$ in the context in which it appears. Observe that instead of treating division as a partial function on the domain $\mathcal{Q} \times \mathcal{Q}$, we treat it as a total function on the domain $\mathcal{Q} \times \mathcal{Q}_z$. The price paid is that theorem proving is required during the analysis of a specification in order to establish the admissibility of an expression such as $\frac{\rho}{n-m}$. Such theorem proving is no problem in a fully mechanized verification system with an effective theorem prover.

We have found that delicate and sometimes novel design decisions such as this are essential in order to combine expressiveness with mechanical tractability in a specification language.

The logical foundations of a specification language determine what can be expressed, and also what can be expressed conveniently. In addition, most specification languages encourage or mandate a certain style of specification. For example, some specification languages require constructive specifications, in which each new operation is specified in terms of a mechanism to achieve it. The advantage of such constructive specifications is that they guarantee soundness and consistency: the specification is sure to mean something. Their disadvantage is that they often result in overspecification: we often want a specification to say ***what*** is to be achieved, not ***how*** it is to be achieved.

The alternative to constructive specifications is the axiomatic style: we simply state the properties that the new operation is to satisfy. The danger is that axioms can introduce inconsistencies: the specification may mean nothing at all. One way to overcome this difficulty is to provide a second, constructive, specification and to prove that it satisfies the axioms of the first. The constructive specification is not prescriptive, and need not even be realistic, its purpose is simply to demonstrate that the axioms are satisfiable.

In my opinion, a specification language should support both the constructive and axiomatic specification styles. Constructive specifications should satisfy a definitional principle that guarantees they do not introduce inconsistencies (this can require theorem-proving in the case of recursive definitions), and it should also be possible to demonstrate the consistency of axiomatic specifications by exhibiting a constructive model of the axioms. EHDM fully supports both styles of specification in the manner suggested, and we have found this a great benefit. The HOL and Boyer-Moore systems are generally used in a purely constructive manner (though recent extensions to the Boyer-Moore system allow axiomatic constraints), while Z employs an essentially axiomatic style.

2.2 Verification Systems

A verification system is a computer program that provides tools to support the analysis of formal specifications. At the very least, such a system must include a parser and a typechecker for the specification language concerned, so that specifications can be checked for syntactic and semantic consistency. Since specifications can get quite large, browsers, cross-reference generators, and other similar support tools for navigating a large body of material are highly desirable. In fact, a fully-fledged formal verification system should provide many of the services of a programming support environment, with specification libraries, version management, and so on. EHDM provides a rich set of capabilities of this kind.

Since specifications are used to communicate information among the design team, and between the design team, customer, certifying authority, and others, facilities should be provided that allow specifications to be presented in an attractive form that facilitates comprehension. EHDM's cross-reference generator provides typeset text automatically, and its table-driven LaTeX-printer converts specification text such as

```
abs(c(p, i, T) - c(q, i, T))
```

into the typeset notation of conventional mathematics:

$$|c_p^{(i)}(T) - c_q^{(i)}(T)|.$$

The value of such simple tools should not be underestimated.

Mechanically-checked formal verification requires a theorem prover or proof checker. (A proof checker is simply a theorem prover that requires more human guidance.) It might seem that the more powerful and automatic the theorem prover the better. This is a view that needs severe qualification however.

First, to derive the maximum value from the exercise, we want to obtain a genuine proof—that is a chain of argument that will convince a human reviewer—rather a mere grunt of assent from a mechanical theorem prover. It is the proof that distills the insight into why a certain design does its job, and it is the proof that we will need to examine if we subsequently wish to change the design or its requirements. Many powerful theorem proving techniques (for example, resolution) work in ways that do not lend themselves to the extraction of a proof, and such methods are unacceptable on this count.

Second, many of the theorems that we wish to prove will turn out to be incorrect—i.e., they will not be theorems after all. It is precisely because the details of unverified proofs can be missing or flawed that mechanical verification is of value: but this means that it is at least as important for the theorem prover to provide assistance in the discovery of error, as that it should be able to prove true theorems with aplomb. Powerful automatic theorem provers are generally built on the assumption that they will be put to work on true theorems. When confronted with false theorems, they may relapse into inscrutable silence. Yet it is in the discovery and isolation of mistakes that formal verification can be of most value. The philosopher Lakatos argues similarly for the role of proof in mathematics. Roughly, this view is that successful completion is among the least interesting and useful outcomes of a proof attempt; the real benefit comes from failed proof attempts, since these challenge us to revise our hypotheses, sharpen our statements, and achieve a deeper understanding of our problem:

> "The virtue of a logical proof is not that it compels belief but that it suggests doubts." [17, page 48]

These limitations on the utility of powerful automatic theorem proving in the large do not apply in the small. Routine manipulations of arithmetic, equalities, disequalities, and inequalities can, and should, be completely mechanized. This is so because a human reviewer does not need to examine the argument for elementary deductions such as $x \geq y \geq z \supset x + 1 > z$, and because ***complete*** decision procedures are available (at least for the ground case) [28], so that their behavior is predictable. It is hard to overestimate the contribution of the decision procedures in EHDM towards the successful completion of the verifications that we have performed. Conversely, it is enormously tedious to perform verifications involving even modest quantities of arithmetic in systems such as HOL that lack decision procedures.

In my opinion, the key to truly effective theorem proving will be a productive symbiosis between man and machine: the user should guide the overall strategy and provide the key insights, and the machine should do the straightforward calculations. Key steps may include case-splits, the invocation of a lemma or axiom, and the generalization or instantiation of variables. No current theorem prover satisfies these requirements perfectly, but we are developing one at SRI that we believe will come very close.

The outcome of a successful proof attempt should be two proof descriptions (or possibly one description that serves two purposes): one should be very close to a proof that can accompany a journal-style presentation of the verification for human consumption; the other should be a description that guides the prover to repeat the proof without human guidance. The latter will be needed in order to rerun the proof at later stages in the overall verification, when many surrounding details may have changed slightly. Ideally, this proof description should be robust—describing a strategy rather than a line by line argument—so that unimportant changes to the specification of lemmas will not derail it. This rerunning of proofs is essential if one regards the purpose of mechanically-checked verification as the acquisition of insight rather than mere certification. As a verification develops, one discovers simplifications, improvements, and generalizations that should be assisted, not discouraged, by investment in an existing verification. Even those most interested in certification should recognize that a specification is seldom static: a change in external requirements or in the specification of an assumed service will require reverification.

The dynamic and creative nature of verification development argues strongly for the ability to perform proofs in any order. Some systems (e.g., HOL) require, or strongly encourage, a bottom up development in which only previously proven lemmas can be cited in a new proof. This is not the way real mathematics is performed: one generally prefers to know whether a proposed lemma is adequate to its intended use before attempting to prove it. If proofs can be attempted in any order, it becomes necessary to provide an analysis tool that examines the macroscopic structure of a complete verification in order to ensure the absence of circularities and unproved lemmas. EHDM includes such a tool, which has the additional benefit of providing a complete enumeration of the assumptions, definitions, and axioms on which the verification ultimately depends. This enumeration identifies the requirements that must be satisfied at the next lower level of design, and the real-world assumptions that must be validated empirically.

2.3 Specialized Systems

One way to increase the automation and effectiveness of the tools that can be provided for formal specification and verification is to limit their domain of application, and to build specialized systems tailored to particular problems of interest.

For example, instead of addressing the functional properties of software modules, we could focus on their structural properties—that is on how things are connected together. These properties are very important to system maintenance, since they allow some of the ramifications of changes to be evaluated. From the point of view of formal specification and verification, they have the advantage that specifications can be represented by pictures, and the formal verification can be completely automated [22]. Working with a company involved in power generation and distribution, our laboratory is developing tools for applying formal methods to such pictorial representations of system structure.

Another specialized domain is associated with temporal logic. Propositional temporal logic is not powerful enough to serve as the basis for a general-purpose specification language, but it is well-suited to certain specialized but important problems concerning hardware designs, communications protocols, and finite-state controllers. Certain verification problems in temporal logic can be solved completely automatically using a technique known as model-checking [9].

3 Accomplisments of Formal Methods

Despite the advances reported in the previous section, effective general-purpose tools for formal specification and verification are only just emerging from the laboratory, and are still difficult and time-consuming to use. Although formal verification from "top to bottom" (e.g., from the requirements on a simple program, through the program, compiler, assembler, and hardware design down to the elementary logic gates) has been demonstrated [4], it must still be considered a tour de force rather than a routine capability.

If formal verification is too expensive to apply to all levels and aspects of a design, where should it be used for maximum effectiveness? In my opinion, it makes most sense to use formal techniques in circumstances where other techniques have proven inadequate. These circumstances may vary from one application area to another, but in the one I know best (flight control systems), the most vulnerable and difficult part of the system seems to be that concerned with fault-tolerance and redundancy management. For example, the triplex-redundant flight-control system of the experimental AFTI-F16 was built in a way that sought to deal with the problems of redundancy management in a plausibly simple way, using unsynchronized channels. Unfortunately, its flight tests revealed numerous problems in the approach, to the extent that redundancy management became the ***primary source of unreliability*** in the control system and an impediment to testing [20, pp. 40–41]. These serious difficulties in plausibly simple designs have stimulated research into principled approaches to redundancy management for flight control systems that will yield predictable behavior.

The favored approach uses synchronized channels, Byzantine fault-tolerant distribution of sensor data to all channels, and exact-match voting [12]. Synchronization requires that the clocks of the individual channels be kept within some bounded skew of each other, so that each channel will perform the same computations and will be ready to participate in votes and the distribution of sensor data at approximately the same time. Of course, the clock synchronization and the distribution and voting protocols must themselves be fault-tolerant, and the subtlety and delicacy of these protocols were not properly understood prior to the investigations of the SIFT project [18, 23]. Consequently, early synchronization protocols were seriously flawed. For example, the failure of the first attempt to launch the Space Shuttle was due to a synchronization problem [14], and some observations were lost when the heavy radiation environment at Jupiter caused one of the clocks on the Voyager spacecraft to jump several seconds [1].

The basic means for achieving fault-tolerant synchronization is for each processor to periodically exchange clock values with the other processors, compute a consensus reading from these values, and adjust its own local clock to reflect the consensus. The difficulty is that processors can fail in arbitrary, unpredictable ways and can upset the consensus by communicating one clock value to one processor and a different clock value to another. An algorithm that can cope with such failures is said to be ***Byzantine*** fault-tolerant [19]. There are a number of algorithms for Byzantine fault-tolerant clock synchronization, including one known as Interactive Convergence [18]. These algorithms themselves are fairly simple to describe, but the reasoning required to establish their correctness is extremely delicate. The difficulty stems from having to simultaneously deal with relative and absolute clock drifts, processor failures, reading errors, and the complicated arithmetic that is involved.

The journal proof [18] that the Interactive Convergence Clock Synchronization Algorithm maintains synchronization despite the occurrence of Byzantine faults comprises five lemmas and a main theorem. Our analysis, using EHDM, demonstrated that four of the five lemmas, and the main theorem, were false as stated [25]. As far as we know, these flaws had not previously been detected by the "social process" of informal peer scrutiny to which the journal paper has been subjected since its publication. We corrected the flaws in the journal proof (which were nontrivial and required changes to the external specification of the algorithm, not just to the proofs themselves), formally verified the corrected proof (which requires 190 EHDM `PROVE` declarations), and extracted a revised journal-level proof from the formal verification that is much simpler and easier to follow than the original, incorrect, proof.

A related application undertaken for NASA was the formal specification and verification of a model for fault-masking and transient-recovery in digital flight-control systems [24]. The interest here is that the formal development in EHDM was undertaken in parallel with a similarly detailed development using conventional pencil-and-paper mathematical notation [12]. The EHDM version took no longer to develop than the other, is more general, is equally readable, and has been fully verified. The conclusion we draw from this exercise is that it is feasible to do detailed formal design directly *in* a formal specification language: it is not necessary to do the real design work using informal techniques, nor to require a "shadow" design, divorced from the real development, as the target for formal specification and verification.

I consider that these exercises clearly demonstrate the value of formal verification in clarifying and correcting the underpinnings of difficult and critical algorithms. Others working on NASA contracts have formally verified an algorithm and hardware implementations for the Byzantine fault-tolerant distribution of sensor samples [3, 30] and several other facets of the overall problem are under current investigation. Much other work on formal verification in the USA has been concerned with secure computer systems, where formal methods are required for the highest level of certification [11].

Elsewhere, HOL has been used in the partial verification of the Viper microprocessor for safety-critical applications [10] (though some aspects of the presentation of that work have been criticised [6]), and a number of other applications of formal methods are described in the September 1990 special issue of IEEE Software. Application of formal methods to expert systems and AI software has also been considered [27, 31].

4 Prospects for Formal Methods

Steady progress over the last 15 years has resulted in a variety of effective languages and tools for formal specification and verification and I would expect and hope that these will be of value to the power industry. Already, formal methods have been applied in the analysis of the shutdown system for the Canadian Darlington Nuclear Reactor [2]. That experience, probably because it was unplanned and because no automated tools were available, was not an entirely satisfactory one. Future applications of formal methods will be much more successful and cost-effective if they are introduced in a planned and systematic manner, and if tool support is provided.

Formal specifications can be introduced by using them to provide documentation for existing systems in anticipation of later modification or reuse. Such applications can provide training and familiarization in the use of formal methods in a relatively risk-free manner. If considered successful, later projects can use formal specifications during development. I believe, and there is some evidence from Europe that this belief is justified, that such use of formal specifications can be justified on purely cost-effectiveness grounds. At present, full-blooded formal verification is probably best employed as an additional form of scrutiny in truly critical systems.

The main advantages I would claim for formal specification and verification are the following.

- Formal specifications are precise and unambiguous. With a good specification language, they can also be very readable.

- Formal verification can be used to test specifications early in the design process, when no other form of validation is feasible.

- The requirements of a totally formal specification and mechanically-checked verification impose a degree of scrutiny that is more exacting and thorough than other methods.

- If suitable formal specification languages and proof checkers are used, it is possible to extract a detailed and precise, but informal argument from the formal verification. This can be used for human review and certification.

- Formal verification forces a complete and explicit enumeration of the assumptions that underpin a design.

- Because the verification is mechanically checked, it is possible to explore the consequences of slightly changed assumptions or specifications in a reliable manner.

Although undeniably expensive at present, these advantages of formal verification may render it cost-effective in some critical applications. Significant improvements in cost-effectiveness can be expected over the next few years as we at SRI, and doubtless others too, are developing "new-generation" systems to exploit new ideas in language design and theorem proving, and the power of modern workstations.

References

[1] Anonymous. Reprogramming capability proves key to extending Voyager 2's journey. *Aviation Week and Space Technology*, page 72, August 7, 1989.

[2] G. H. Archinoff, R. J. Hohendorf, A. Wassyng, B. Quigley, and M. R. Borsch. Verification of the shutdown system software at the Darlington nuclear generating station. In *International Conference on Control and Instrumentation in Nuclear Installations*, Glasgow, UK, May 1990. The Institution of Nuclear Engineers.

[3] W. R. Bevier and W. D. Young. The design and proof of correctness of a fault-tolerant circuit. In Meyer and Schlichting [21], pages 243–260.

[4] William R. Bevier, Warren A. Hunt, Jr., J Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[5] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, New York, NY, 1988.

[6] Bishop Brock and Warren A. Hunt, Jr. Report on the formal specification and partial verification of the VIPER microprocessor. In *COMPASS '91 (Proceedings of the Sixth Annual Conference on Computer Assurance)*, pages 91–98, Gaithersburg, MD, June 1991. IEEE Washington Section.

[7] *The HOL System*. Cambridge Computer Science Research Center, SRI International, Cambridge, England, December 1989. Three volumes: Description, Tutorial, and Reference Manual.

[8] J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In Carroll Morgan and J. C. P. Woodcock, editors, *Proceedings of the Third Refinement Workshop*, pages 51–69. Springer-Verlag Workshops in Computing, 1990.

[9] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[10] W. J. Cullyer. Implementing safety critical systems: The VIPER microprocessor. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 1–26. Kluwer Academic Publishers, Boston, MA, 1988.

[11] *Department of Defense Trusted Computer System Evaluation Criteria*. Department of Defense, December 1985. DOD 5200.28-STD (supersedes CSC-STD-001-83).

[12] Ben L. Di Vito, Ricky W. Butler, and James L. Caldwell. High level design proof of a reliable computing platform. In Meyer and Schlichting [21], pages 279–306.

[13] S. Feferman. Polymorphic typed lambda-calculi in a type-free axiomatic framework. In W. Sieg, editor, *Logic And Computation*, pages 101–136. American Mathematical Society, 1990.

[14] John R. Garman. The "bug" heard 'round the world. *ACM Software Engineering Notes*, 6(5):3–10, October 1981.

[15] M. Gordon. HOL: A machine oriented formulation of higher order logic. Technical Report 68, University of Cambridge Computer Laboratory, Cambridge, England, July 1985.

[16] Ian Hayes, editor. *Specification Case Studies*. Prentice-Hall International Ltd., Hemel Hempstead, UK, second edition, 1993.

[17] Imre Lakatos. *Proofs and Refutations*. Cambridge University Press, Cambridge, England, 1976.

[18] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[19] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[20] Dale A. Mackall. Development and flight test experiences with a flight-crucial digital control system. NASA Technical Paper 2857, NASA Ames Research Center, Dryden Flight Research Facility, Edwards, CA, 1988.

[21] J. F. Meyer and R. D. Schlichting, editors. *Dependable Computing for Critical Applications—2*, volume 6 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, Vienna, Austria, February 1991.

[22] M. Moriconi and D. F. Hare. Visualizing program designs through PegaSys. *IEEE Computer*, 18(8):72–85, August 1985.

[23] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[24] John Rushby. Formal specification and verification of a fault-masking and transient-recovery model for digital flight-control systems. Technical Report SRI-CSL-91-3, Computer Science Laboratory, SRI International, Menlo Park, CA, January 1991. Also available as NASA Contractor Report 4384, July 1991.

[25] John Rushby and Friedrich von Henke. Formal verification of the Interactive Convergence clock synchronization algorithm using EHDM. Technical Report SRI-CSL-89-3R, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1989 (Revised August 1991). Original version also available as NASA Contractor Report 4239, June 1989.

[26] John Rushby, Friedrich von Henke, and Sam Owre. An introduction to formal specification and verification using EHDM. Technical Report SRI-CSL-91-2, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1991.

[27] John Rushby and R. Alan Whitehurst. Formal verification of AI software. Contractor Report 181827, NASA Langley Research Center, Hampton, VA, February 1989. (Work performed by SRI International).

[28] Robert E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, April 1979.

[29] J. M. Spivey, editor. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, second edition, 1993.

[30] Mandayam Srivas and Mark Bickford. Verification of the FtCayuga fault-tolerant microprocessor system, volume 1: A case-study in theorem prover-based verification. Contractor Report 4381, NASA Langley Research Center, Hampton, VA, July 1991. (Work performed by ORA Corporation).

[31] Richard J. Waldinger and Mark E. Stickel. Proving properties of rule-based systems. In *Proceedings of the Seventh Conference on Artificial Intelligence Applications*, pages 81–88, Miami Beach, FL, February 1991. IEEE Computer Society.