Appears as Appendix C.1, pages 121–125 in Dan Craigen and Karen Summerskill, editors, *Formal Methods for Trustworthy Computer Systems (FM89)*, Halifax, Nova Scotia, Canada, July 1989. Springer-Verlag Workshops in Computing.

Formal Methods and Critical Systems In the Real World

John Rushby
Computer Science Laboratory
SRI International

Programmable computers make it possible to construct systems whose behavior is unimaginably complex. These systems are built because their complexity is believed to confer operational benefits, but this same complexity can harbor unexpected and catastrophic failure modes. The source of these failures can often be traced to software faults—for example, a software bug in the control system of the Therac-25 radiation therapy machine was responsible for the death of three patients and serious injury to several others [Jac89].

Software doesn't wear out: all software-induced failures are due to design faults, and design faults are largely attributable to the complexity of the designs concerned—complexity that exceeds the intellectual grasp of its own creators. The only way to reduce or eliminate software design faults is to bring the complexity of the software into line with our ability to master that complexity. This might mean choosing not to build certain types of system (such as flight-critical computer control systems for passenger aircraft), and it should mean enhancing the intellectual tools available to software designers.

Engineers in established fields use applied mathematics to predict the behavior and properties of their designs with great accuracy. Software engineers, despite the fact that their creations exhibit far more complexity than physical systems, do not generally do this and the practice of the discipline is still at the pre-scientific or craft stage. Unlike most physical systems, the behavior of software admits discontinuities and so interpolation between known points is unreliable: formal logical analysis is needed to address the discrete, formal, character of software. Thus, the applied mathematics of software is formal logic, and calculating the behavior of software is an exercise in theorem proving. Just as engineers in other disciplines need the speed and accuracy of computers to help them perform their engineering calculations, so software engineers can use the speed and accuracy of computers to help them prove the (large number of relatively simple and not intrinsically interesting) theorems required to predict the behavior of software.

"Formal Methods" are nothing but the application of applied mathematics—in this case, formal logic—to the design and analysis of software-intensive systems. Formal methods can be used during the design and documentation of systems, and they can also provide evidence for consideration during the assessment and certification of systems that perform critical functions. The former is surely uncontroversial—one should rather have to defend the absence than the use of formal methods (though see [Nau82])—and some projects, notably several undertaken in the UK, attest to a practical benefit from using formal methods.

Concern that faults could have very serious consequences has led to the introduction of special standards requiring use of formal methods during the construction and quality assurance of certain classes of computer systems (for example, the US "Orange Book" [DoD85] for secure systems and the British Defence Standard 00-55 [MOD91] for safety critical systems). Such use of formal methods—particularly mechanically checked formal verification—in the certification of critical systems is more controversial.

In established engineering disciplines, the reliability and accuracy of predictions of system behavior are determined by the fidelity of the mathematical models on which they are based, and by the extent to which the necessary calculations are performed without error. For example, the accuracy of the predicted performance of an airfoil depends on how well the chosen aerodynamic model captures the real behavior of air over a wing. It is obvious that similar considerations apply to the reliability and accuracy of predictions concerning the behavior of digital systems made by formal verification. For example, program verification depends on an assumed semantics for the programming language concerned, but these assumed semantics may not coincide with those of its implementation; additionally, the proofs performed during verification may be flawed. The nature and significance of these potential flaws in the efficacy of formal methods are no different from those that attend the use of applied mathematics in other engineering disciplines. Yet—perhaps because the calculations performed in formal verification are proofs, and the inexperienced tend to associate proofs with absolute guarantees—these limitations to formal verification have caused some to excoriate the field [DLP79, Fet88] (see [Bar89] for one of the few well-informed discussions of this controversy). These limitations to verification are not, contrary the assertions of its detractors, denied or minimized by those in the field (see [Coh89] for example), but the more interesting question—what to do about them—has received scant attention.

Assurance and Certification for Critical Systems

One body of opinion suggests that limitations on the value of the assurance provided by formal verification can be minimized by applying the technique more completely and rigorously. For example, the fear that a verification may be unsound because the assumed and the implemented semantics of the programming language do not agree can be minimized by verifying the language implementation concerned. Of course, this verification will depend on the semantics of the hardware interpreter concerned—and that can be assured by verifying the hardware down to, say, a simple gate model. This approach is valuable and interesting, but it must be careful to address at least two objections.

- The lowest model in a verification hierarchy cannot be verified; evidence for its correctness must be obtained by other means. In addition, purely logical models become increasingly incomplete as increasingly lower levels of implementation are considered. For example, formal gate models do not capture all the relevant properties of hardware implementations: physical properties such as excessive fan-out or power dissipation, violations of topographical design constraints, metastability, or an excessive clock rate, not to mention manufacturing defects, can all undermine formal verification at the hardware level.

- If we build the wrong system, then it avails us little if the semantics of the programming language that supports it are implemented perfectly and run on perfect hardware. In complex systems, the limiting factor may be imperfect understanding of all the properties required for a given component or set of components. System failures can often be traced to the interfaces between components: each component performs as required, but their interaction produces undesired and unexpected behavior [Lev86]. Again, these issues can be modeled, and perhaps they, rather than the relatively routine aspects of system implementation, should most urgently receive the benefit of formal analysis.

The problem of flawed calculations—i.e., unsound proofs—can be minimized by using mechanical proof checkers built on a very simple and evidently sound foundation. The class of provers derived from LCF are in this category. The disadvantage of this approach is that the provers concerned are orders of magnitude less effective than those built on more powerful—but less assured—foundations.

The root difficulties of the "verify everything" approach and the use of "slow but sure" provers are economic. All engineering is about compromise—wisely chosen and justifiable compromise. In the case of a critical system, we have to consider not only the absolute value of the resources that should be expended on its construction and certification, but also how those resources should best be apportioned. A thousand dollars spent on formal verification will mean a thousand dollars less for testing, or a thousand dollars less for protective redundancy. We have to ask: if some formal verification is good, is more always better?

My own view is that judicious application of formal methods in system design and documentation is essential—indeed, there is no credible alternative. Formal methods used for this purpose do not absolutely require mechanical support; they provide intellectual tools that can be practiced with pencil and paper. The Oxford Z methodology, discussed by several participants in this workshop, has this character.

For assurance and certification, formal methods also have much to offer, but heavy-duty (i.e., mechanically checked) formal verification is just one option among several, to be applied only where it will be the most effective choice. The alternative choices include formally-based methods other than conventional verification (for example, fault-tree analysis, structured walk-throughs, and anomaly detection), empirical testing, and approaches based on error-detection and fault-tolerance. The overall goal should be the construction of Dependable Systems: those in which reliance may *justifiably* be placed on certain aspects of the quality of service delivered [Lap85].

An important facet of dependability is that it is a systems (i.e., big picture) concept. Perfection in components and subsystems does not necessarily provide a dependable system overall—and may not be the best way to achieve dependability. If dependably safe control cannot be guaranteed of digital avionics (and many believe it cannot), then it may still be possible to build a safe airplane containing digital avionics—provided the digital control system is not a single point of failure. At the least, this requires the airplane to be aerodynamically stable, and that there should be some direct connection between the pilot and the control surfaces.

The concept of dependability also introduces a significant compromise: we do not require that every aspect of system functionality should be provided dependably, only that selected aspects should. Thus, for example, although we would like the digital avionics of passenger aircraft to provide both safe and fuel-efficient control, we might insist that only safety is assured to the highest levels of dependability. In these cases, it is possible to work backwards from hypothesized dependability failures in order to discover whether any errors in design or implementation can allow those failures to occur. This technique of "software fault-tree analysis" [LH83] can be considered a formal method but, because it reasons backwards from hypothesized failures rather than forward from one supposedly good state to another, it rests on rather different assumptions than other methods and can identify faults that have been overlooked in more conventional analyses.

Direct testing also probes assumptions; it can be used to validate the explicit assumptions of a formal analysis (just as a wind-tunnel may be used to validate the assumptions used in an aerodynamic calculation), and it can expose hidden assumptions and misunderstood or overlooked requirements. Testing is sometimes dismissed by those who espouse the purest of formal methods ("testing reveals the presence of faults, never their absence"), but I can imagine no serious approach to the certification of critical systems that does not require explicit and extensive testing. The interesting challenge is to identify ways in which testing can reinforce and support formally-based analyses.

The same is true of run-time error checking. Even a verified microprocessor is vulnerable to surges on its power line, and to strikes by alpha-particles and bullets. It is only prudent to monitor the progress of a computation in order to ensure that nothing is going drastically awry. By combining run-time checking with formal analysis, it should be possible to make stronger and more reliable statements than is possible with either technique alone [AW78].

If run-time error checking is performed, it is natural to provide some form of recovery, or fault-tolerance, in response to detected errors. Some degree of fault tolerance is normally considered essential in high-dependability systems. Physical components wear out, and the external environment may introduce unanticipated circumstances. Fault-tolerance requires the monitoring of performance, and the presence of redundancy. Both add complexity, and may thereby reduce, rather than enhance, dependability. The algorithms needed to provide "Byzantine fault-tolerant" synchronization and agreement, for example, are of considerable difficulty, and the software that manages the coordination, voting, error-detection, recovery and reconfiguration in a fault-tolerant system becomes a single point of failure in the overall system. Because of its criticality and difficulty, it is an excellent candidate for heavy-duty formal verification.

From fault-tolerance intended to protect against component malfunction, it is but a small step to contemplate redundancy and fault-tolerance as a safeguard against *design* faults. "Software-fault tolerance" relies on "diversity" (multiple designs and implementations for each program module), run-time checking, and majority voting to safeguard against design faults in critical software [AL86].

Formal Methods and Systems Engineering

Some researchers in the formal methods community are reluctant to involve themselves with systems that are to be used in the real world. Since the invitation to participate usually comes only from those who are developing systems of unusual criticality (where conventional methods of assurance are known to be inadequate), this reluctance is understandable. Many feel that the complexity of the systems concerned, the constraints that surround them, and the serious consequences that could attend their failure, are such that formal methods can provide only partial assurances that could be misinterpreted.

Others believe that the systems concerned are going to be built anyway, and that it is better to bring some of the benefits of formal methods to bear than none at all; in the absence of participation by the formal methods community, other—perhaps inferior—technologies may become entrenched. It should be noted, for example, that a particular incarnation of software fault tolerance ("N-version programming" [Avi85]) is the *dominant technology* for dependable systems in aerospace, where the flight control computers for the Boeing 737-300 and for the Airbus A310 and A320 are N-version systems; I am unaware of any deployed flight control system that has been subject to formal verification.

The presentations that we have heard today all describe serious attempts to apply formal methods to real systems in a limited but responsible manner. In each case, we have seen that it has been important to consider the system context: formal methods should be targeted where they will do the most good. And we have seen that formal methods must be integrated with other approaches to validation and assurance.

I believe that integration of formal methods with other forms of assurance is necessary for truly dependable systems. Calling for verification to simply be used in addition to fault-tree analysis, testing, run-time checking, and fault-tolerance is trite. What we need is not simple redundancy of analysis, but integration: we need to know how to use testing and verification in support of each other, to know how the one can validate the assumptions of the other. And we need to investigate how run-time checking and fault-tolerance can be used to provide principled detection and response to violated assumptions, rather than a gamble on the laws of probability.

In summary, I invite those who are willing to essay the application of formal methods to critical real-world systems to consider the following points:

1. Our techniques are not infallible: any really critical system should be designed so that it can operate safely (though possibly in a degraded mode) without the computer system: even if all the flight control computers fail, it should still be possible to land the plane.

2. Formal methods are not synonymous with mechanically checked formal verification. The "manual" use of formal methods during design, and for documentation, may be of considerable benefit.

3. Formal verification, if it is to be employed at all, cannot be applied everywhere; we need to target its application with great care. Software components that constitute a single point of system failure (such as those that manage the fault-tolerance mechanisms) are natural candidates.

4. We need to develop a foundation for the integration of multiple forms of assurance: formal methods do not stand alone.

The foregoing presentations have shown us some of the opportunities and challenges that await those who are willing to apply formal methods in a systems engineering context. I hope more of us will follow their example.

References

[AL86] A. Avižienis and J. C. Laprie. Dependable computing: From concepts to design diversity. *Proceedings of the IEEE*, 74(5):629–638, May 1986.

[Avi85] Algirdas Avižienis. The N-Version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, December 1985.

[AW78] T. Anderson and R. W. Witty. Safe programming. *BIT*, 18:1–8, 1978.

[Bar89] Jon Barwise. Mathematical proofs of computer system correctness. *Notices of the American Mathematical Society*, 36:844–851, September 1989.

[Coh89] Avra Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(2):127–139, June 1989.

[DLP79] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979.

[DoD85] *Department of Defense Trusted Computer System Evaluation Criteria.* Department of Defense, December 1985. DOD 5200.28-STD (supersedes CSC-STD-001-83).

[Fet88] James H. Fetzer. Program verification: The very idea. *Communications of the ACM*, 31(9):1048–1063, September 1988.

[Jac89] Jonathan Jacky. Programmed for disaster: Software errors that imperil lives. *The Sciences*, pages 22–27, September/October 1989.

[Lap85] J. C. Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *Fault Tolerant Computing Symposium 15*, pages 2–11, Ann Arbor, MI, June 1985. IEEE Computer Society.

[Lev86] Nancy G. Leveson. Software safety: Why, what and how. *ACM Computing Surveys*, 18(2):125–163, June 1986.

[LH83] N. G. Leveson and P. R. Harvey. Analyzing software safety. *IEEE Transactions on Software Engineering*, SE-9(5):569–579, September 1983.

[MOD91] *Interim Defence Standard 00-55: The Procurement of Safety Critical Software in Defence Equipment.* UK Ministry of Defence, April 1991. Part 1, Issue 1: Requirements; Part 2, Issue 1: Guidance.

[Nau82] Peter Naur. Formalization in program development. *BIT*, 22(4):437–453, 1982.