Formal Methods and their Role in the Certification of Critical Systems[*]

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

Rushby@csl.sri.com
Phone: +1 (415) 859-5456 Fax: +1 (415) 859-2844

Technical Report CSL-95-1 March 1995

[*]Preparation of this report was sponsored by the Federal Aviation Administration, FAA Technical Center, Atlantic City NJ, and by the National Aeronautics and Space Administration Langley Research Center, Langley VA, under Contract NAS1-18969, monitored by NASA Langley Research Center.

Abstract

This report is based on one prepared as a chapter for the FAA Digital Systems Validation Handbook (a guide to assist FAA Certification Specialists with Advanced Technology Issues).[1] Its purpose is to explain the use of formal methods in the specification and verification of software and hardware requirements, designs, and implementations, to identify the benefits, weaknesses, and difficulties in applying these methods to digital systems used in critical applications, and to suggest factors for consideration when formal methods are offered in support of certification.

The presentation concentrates on the rationale for formal methods and on their contribution to assurance for critical applications within a context such as that provided by DO-178B (the guidelines for software used on board civil aircraft)[2]; it is intended as an introduction for those to whom these topics are new. A more technical discussion of formal methods is provided in a companion report.[3]

[1] *Digital Systems Validation Handbook–Volume III.* Federal Aviation Administration Technical Center, Atlantic City, NJ. Forthcoming.

[2] *Software Considerations in Airborne Systems and Equipment Certification.* Requirements and Technical Concepts for Aviation (RTCA), Washington, DC, December 1992.

[3] John Rushby, *Formal Methods and the Certification of Critical Systems*, Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA. Also NASA Contractor Report 4551.

Contents

Glossary

These explanations are provided to help the nonspecialist. They are intended to reflect the technical uses of the terms considered, but do not attempt to incorporate subtleties that concern the specialist.

Abstraction: the process of simplifying certain details of a system description or model so that the main issues are exposed. Abstraction is the key to gaining intellectual mastery of any complex system, and a prerequisite to effective use of formal methods. It requires great skill and experience to use abstraction to best effect.

In formal methods, abstraction is part of the process of developing a mathematical model that is a simplification or approximation of reality but that retains the properties of interest. In physics, for example, it is customary to model a moving object as a point mass, and to ignore its shape. Similarly in the case of a flight-control system, one can analyze properties of, say, the clock synchronization algorithm or the redundancy management mechanisms by abstracting these away from the larger and more complex system in which they are embedded.

Correctness: the property that a system does what it is expected and required to do. Formal methods cannot establish correctness in this most general sense because they deal with formal *models* of the system that may be inaccurate or incomplete, and with formal statements of requirements that may not capture all expectations. The difference between the real and modeled worlds is a potential source of error that attends all uses of mathematical modeling in engineering (e.g., in numerical aerodynamics or stress calculations) and that must be controlled by *validating* the models concerned. The difference between expectations and documented requirements is another problem that attends all engineering activities. Formal methods provide ways to make the specifications of assumptions and requirements precise; *formal validation* (q.v.) can then be used to ensure that the specifications are adequately complete and correct.

Correctness does not ensure *safety* or other critical properties, since the system requirements and expectations may not address these issues (correctly or at completely). System requirements usually describe functional properties (i.e., what the system is to *do*); it is necessary to establish nonfunctional properties such as safety and security (which often describe what the system is *not* to do) by separate scrutiny (based, e.g., on hazard analysis, or threat analysis). Formal methods can be used in these processes.

Design Faults: mistakes in the design of the system, or in the understanding of its requirements and assumptions, that cause it to do the wrong thing or to fail in certain circumstances. Also called *generic* faults. Modular redundancy provides no protection against these faults.

Formal logic: symbolic notation equipped with rules for constructing *formal proofs* (q.v.). Formal logic consists of a language for writing statements and syntactic rules of inference for constructing proofs using these statements. Formal logic supports a form of reasoning that does not rely on the subjective interpretation of the symbols used. For example, "All $\diamondsuit$s are $\sharp$; this $\odot$ is a $\diamondsuit$; therefore, this $\odot$ is $\sharp$" is sound reasoning, no matter what the symbols mean. Because they do not depend on intuition, formal proofs can be developed or checked by computer (see *theorem proving*).

There are many formal logics; they differ in what concepts they can express, and in how difficult it is to discover or check proofs. Propositional logic, first-order logic, higher-order logic, the simple theory of types, and temporal logic are all examples of formal logics that find application in formal methods. These logics are generally augmented with certain *theories* defined within them that provide definitions or axiomatizations for useful mathematical concepts, such as *sets*, *numbers*, *state machines*, etc.

Formal Proof and formal deduction. Formal *deduction* is the process of deriving a sentence expressed in a formal logic from others through application of one or more rules of inference. In the example above, formal deduction allows us to derive the sentence "this $\odot$ is $\sharp$" from the two sentences "All $\diamondsuit$s are $\sharp$" and "this $\odot$ is a $\diamondsuit$."

A formal *proof* is a demonstration that a given sentence (the *theorem*) follows by formal deduction from given (i.e., assumed) sentences called *premises*.

Formal methods: methods that use ideas and techniques from mathematical or *formal logic* (q.v.) to specify and reason about computational systems (both hardware and software).

Formal specification: a description of some computational system expressed in a notation based on formal logic. Generally, the specification states certain *assumptions* about the context in which the system is to operate (e.g., laws of physics, properties of subsystems and of systems with which the given system is to interact), and certain *properties* required of the system. A *requirements specification* need specify no more than this; a *design specification* will specify some elements of how the desired properties are to be achieved—e.g., algorithms and decomposition into subsystems.

Formal validation: a process for gaining confidence that top-level formal specifications of requirements and assumptions are correct. Formal verification (q.v.) cannot be applied at these levels because there are no higher-level requirements or more basic assumptions against which to verify them: processes of review and examination must be used instead. Formal validation consists of *challenging* the formal specifications by proposing and attempting to prove theorems that ought to follow from them (i.e., "if I've got this right, then this ought to follow.")

Formal verification: the process of showing, by means of formal deduction, that a formal design specification satisfies its formal requirements specification. The formal description of a design and its assumptions supply the premises, and the requirements supply the theorem to be proved. In hierarchical developments, assumptions and designs at one level become requirements at another, so the formal verification process can be repeated through many levels of design and abstraction. At the topmost level, *validation* (q.v.) must be employed.

Theorem proving and proof checking. Given a putative theorem and its premises, a theorem prover attempts to *discover* a proof that the theorem follows from the premises; on the other hand, a proof checker simply *checks* that a given proof is valid according to the rules of deduction for the logic concerned. Both these processes can be automated. A theorem prover is a computer program that uses search, heuristics, and user-supplied hints to guide its attempt to discover a proof. A proof checker is a computer program that is used interactively: a human user proposes proof steps and the proof checker checks they are valid and carries them out. The most effective automated assistance for formal methods is generally obtained by a hybrid combination of these approaches: the user proposes fairly big steps and the proof checker uses theorem proving techniques to fill in the gaps and take care of the details. Examples of theorem provers include Otter, Nqthm, PTTP, RRL, and TPS. Examples of proof checkers include Automath, Coq, HOL, Isabelle, and Nuprl. Hybrids include Eves, IMPS, PC-Nqthm, and PVS. Other forms of automated analysis that can be applied to formal specifications include model checking, language inclusion, and state exploration; examples of systems that perform these analysis are SMV, COSPAN, and Murϕ.

Introduction

This report is based on one prepared as a chapter for the FAA Digital Systems Validation Handbook (a guide for aircraft certifiers) [FAA89]. Its purpose is to outline what is meant by "formal methods" and to explain their rationale and suggest techniques for their use in providing assurance for critical applications. The report is intended as an introduction for those to whom these topics are new and assumes no background beyond some exposure to software engineering and to safety-critical systems. A more technical examination of formal methods is provided in a companion report [Rus93].

The presentation is in three sections: the first outlines the general rationale for formal methods; the second considers the different kinds of formal methods, and some of the issues in their selection and application; the third considers their contribution to assurance and certification for critical applications, using the requirements concerning software in civil aircraft for concrete illustration.

1 The Rationale for Formal Methods

Formal methods are a very different approach to software development and assurance than traditional methods. In order to describe why formal methods can be worthwhile, I begin by explaining why the assurance problem is so hard for software.

1.1 The Problem With Software and Its Assurance

Software is notorious for being late, expensive, and wrong. Exasperated technical managers often ask "what's so different about software engineering—why can't we (or, less generously, *you*) do it right?" The unstated implication is that the traditional engineering disciplines—in which technical managers usually received their training—do things better.

In my opinion, this unflattering comparison of software with other engineering endeavors is somewhat justified; in particular, the traditional disciplines are founded on science and mathematics and are able to model and predict the characteristics and properties of their designs quite accurately, whereas software engineering is more of a craft activity, based on trial and error rather than calculation and prediction. The comparison is too glib, however, in that it fails to acknowledge that in two important respects software *is* different. These respects are the complexity of behavior that is achieved by software, and its lack of continuity. These are discussed in the next two sections.

1.1.1 Complexity and Design Faults

Software provides much of the functionality of modern systems, and software therefore directly expresses the scale and complexity of these systems. Complexity is a source of *design* faults, by which I mean faults in the intellectual construction of the system—faults that will cause the system to do the wrong thing in some circumstances. Design faults can occur in any system, independently of the technologies used in its construction (see, for example, [BJ94]) but, because design faults are often due to a failure to anticipate certain interactions among the components of the system, or between the system and its environment, they become more likely as the number and complexity of possible behaviors and interactions increases.

Individual software components perform complex functions in modern systems, and collectively they provide the focus for interaction among all parts of the system, and between the system and its environment and operators. Furthermore, software, because of its mutability, is also the target for most of the changes that are generated in requirements and constraints as the overall design for a system evolves. Thus, software carries the burden of overall system complexity and volatility, and it is to be expected that design faults will most commonly be expressed in software.

The Need for Correctness. Because software is found in active control systems, it is usually infeasible to compensate for possible faults or uncertainties in its design by "overengineering" it to provide a "design margin" in the same way as physical systems: whereas a wing spar may be constructed to withstand loads far greater than any it should encounter in normal flight, the software in an autoland system, for example, has to do exactly the *right* thing.

An Aside on Defensive Programming. A plausible counterpart to overengineering in software may be *defensive programming*, whereby each software component explicitly checks for "impossible" conditions and tries to do something sensible if they arise. The problem is that if the impossible happens, then some failure of design must have already occurred, and there is no telling what impact an autonomous decision to do something locally "sensible" may have on overall system behavior. This is the central problem with complex, interacting systems: local actions can have highly nonlocal consequences.

Another technique whose protection does not extend from physical to design faults is simple modular redundancy. There is always the possibility that physical components may fail—either through manufacturing defects, fatigue and wear-out, improper maintenance, physical damage (e.g., shrapnel from a disintegrating engine, or crushing from a collapsing floor), or environmental effects (e.g., heavy ions from cosmic rays, or excessive moisture and heat following loss of air-conditioning)—so it is a good idea to have spares and backups to provide fault tolerance. A fault-tolerant system must be designed to avoid *common mode failures* in which all its redundant components are brought down by a single cause. An example of a common mode failure is the loss of all hydraulic systems in the Sioux City DC-10 crash: the separate hydraulic systems were routed through a single space near the tail engine and all were severed when that engine disintegrated.[4]

Design faults are the quintessential source of common mode failures, so simple replication can provide no protection against them. It is, of course, possible to provide redundant components based on different designs—so-called multiple-version dissimilar (or diverse) software—but this is not a fully satisfactory solution. I give a very brief summary why this is so in an aside on page 12.

Evidence for Correctness and the Need to Consider All Behaviors. Although defensive programming and software diversity provide palliatives in some circumstances, for most critical software systems there is no alternative to the daunting task of eliminating all design faults—or at least those that could have serious consequences. And it is also necessary to provide evidence that this has been done successfully. This evidence is usually in two parts: one is concerned with the *process* of design and construction—it seeks to show through evidence of good practice that everything has been done to prevent serious design faults being introduced and remaining undetected and uneradicated; the second seeks to demonstrate directly, through examination of the system in operation and under test, and through an analysis of its design and supporting rationale, that it is free of serious faults. The first of these forms of evidence concerns *quality control*, the second provides *quality assurance*.

Assurance for a safety-critical system must, at least in principle, consider all possible behaviors of the system under all the circumstances it might encounter. Since "all possible" behaviors may be too many to examine, two complementary approaches have evolved that attempt to reduce the number of behaviors that must be considered. One way tries to show that the system always does the right thing, the other tries to show that it never does a seriously wrong thing.

For the first approach, we use a combination of analysis and empirical testing to examine those behaviors that are considered most likely to harbor serious faults—for example, those that are close to boundary conditions, or that represent "off nominal" conditions, such as those where some subsystems or redundant components have failed. Examples of this approach are *fault injection* (an empirical method) and *failure modes, effects and criticality analysis* (FMECA, an analytical method).

[4]For a critical examination of ethical and regulatory issues concerning the DC-10, see the compendium edited by Fielder and Birsch [FB92].

An Aside on Multiple-Version Software. The topic of constructing systems that can tolerate faults in their own design using multiple-version software is controversial. The main questions are whether this approach provides any significant additional assurance of safety, and whether that assurance is quantifiable. Answers to these questions hinge on "how much" dissimilarity of design can be achieved in the different versions, and on the extent to which any failures of dissimilar designs will be independent.

The extent of dissimilarity depends on how much of the overall design is developed in multiple versions. If dissimilar design is limited to multiple versions of low-level modules, then no protection is provided against design faults above that level; in particular, the system is fully exposed to faults in the modules' requirements. Furthermore, the degree of design freedom, and hence the scope for diversity, is limited when small components are built to a common set of requirements, and there is some evidence that different designers or implementers do tend to make similar mistakes [ECK$^+$91,KL86]. If dissimilarity is at the level of whole systems or subsystems (e.g., an independent backup to a digital flight control system), then there is the question whether the dissimilar system should have the full capability and assurance of the primary system: if it does, then development and maintenance costs will be at least doubled (and that money could have been spent improving the quality, or the assurance, of the primary system); if not, there is concern whether the secondary system can be relied on in an emergency (e.g., the control envelope of an analog backup system is often less than that of the primary flight control system).

In all cases, there is the critical problem of designing and implementing redundancy management across the dissimilar versions: that is, how to decide when one version has failed and another should be given control (in the case of backup systems), or how to resolve voter disagreements in the case of parallel systems. (Dissimilar designs cannot be expected to produce bit-for-bit identical behavior, so threshold voting has to be used.) Like other problems involving synchronization and coordination of concurrently active distributed components, redundancy management—whether of identical or dissimilar components—is among the most difficult and fault-prone aspects of software design. Redundancy management does not lend *itself* to diversity (e.g., you cannot vote the voters ad infinitum; ultimately a decision must be made and the algorithm by which that decision is accomplished represents a single design), and can be made more complex and fault prone by the need to manage diversity in other components. For example, when, on test flight 44, disagreements among the threshold voters in the AFTI-F16 digital flight control system caused each computer to declare the others failed, the analog backup was not selected because simultaneous failure of two or more digital channels had not been anticipated in design of the redundancy management system [Mac88, p. 44].

For these and other reasons, the guidelines for certification of airborne software state that the degree of protection provided by software diversity "is not usually measurable" and dissimilar software versions do not provide a means for achieving safety-critical requirements, but "are usually used as a means of providing additional protection after the software verification process objectives for the software level...have been met" [RTCA92, Subsection 2.3.2].

The general idea behind the second approach to quality assurance is to hypothesize that the system has done something bad and then to analyze all the circumstances that could cause this to come about and to show that the design prevents them from happening. This approach is inspired by hazard analysis, which is a central concept in safety-critical systems; one particular method for doing it that has been adapted to software is *fault-tree analysis* (FTA) [Lev95, Section 14.3].

The property that is common to the different assurance techniques is that they provide ways to group "essentially similar" behaviors together so that fewer cases need to be considered while still providing effectively complete coverage of all possible behaviors. These techniques are very effective with systems based on mechanical, hydraulic, electrical, and other physical components: these have relatively few "essentially different" behaviors, so that relatively straightforward analysis combined with a modest number of empirical tests is sufficient to cover all possibilities. These familiar techniques are far less effective, however, with complex systems that can exhibit extremely large numbers of essentially different behaviors.

Because the complexity in modern systems is expressed in software, it follows that the software will exhibit a large number of different behaviors and that assurance will be difficult for this reason. In fact, this difficulty is compounded by another attribute of software that distinguishes it from physical systems. This attribute is considered next.

1.1.2 The Discontinuous Behavior of Software Systems

The reason that software is the focus for most of the design complexity in modern systems is its versatility: a software system can provide many different behaviors and can be programmed to respond appropriately to many different circumstances. The source of these different behaviors and responses is in the many discrete decisions that are made as software executes: each decision is discrete in that the subsequent course of execution switches from one path to another according to whether or not some condition is true. Because the relationship between the inputs and the outputs of a piece of software is the cumulative effect of these many discrete decisions, it follows that overall input/output relationship must itself be discretized, or *discontinuous*: small changes in inputs can change the outcomes at certain decision points, resulting in radically changed execution paths and correspondingly large changes in output behavior. This discontinuous relationship between inputs and outputs is the second major respect in which software differs from the physical processes considered by other engineering disciplines.

In physical systems, there is usually a (piecewise) continuous relationship between inputs and outputs: smooth changes in the inputs produce correspondingly smooth changes in the outputs. This allows the complete behavior of a physical system to be extrapolated from a finite number of tests: the continuous character of the system ensures that responses to untested input configurations will be essentially similar to those of nearby cases that have been tested. Departures from continuity are usually catastrophic breakdowns in response to inputs beyond the operating range.

An Aside on Hardware. Although this report speaks only of software, exactly the same concerns apply to many hardware components, especially custom ASICs (application-specific integrated circuits). These share all the important properties of software—notably, design complexity and discontinuity of behavior—and differ only in the technology of their implementation. Whereas software design is ultimately expressed in a programming language such as Ada and then compiled into code that is interpreted by a processor, ASIC designs are expressed in a hardware design language such as VHDL or Verilog and then transformed into hardware structures, or to gate-array configurations. The considerations for assurance described in standards and guidelines such as DO-178B should apply to ASICs as they do to software. Similarly, the techniques of formal methods can be applied to ASICs and other complex hardware designs.

But with software, this method of inferring properties of the totality of possible behaviors from tests on a selected sample is much less secure: without continuity, we cannot assume that neighboring cases are essentially similar to one another, so there is little justification for extrapolating from tested to untested cases. Now, it can be argued that although less than exhaustive testing does not allow *definitive* statements to be made about complex software, it does permit *statistical* statements of its reliability, and that such quantification of reliability is both necessary and sufficient for the certification of safety-critical systems. Sometimes this is countered by the argument that talk of reliability is meaningless when we are dealing with design faults: if design faults are present, they will cause the system to fail in specific circumstances, and the failure is *certain* whenever those circumstances arise. However, we must recognize that occurrence of those circumstances is associated with a random process—namely, the sequence of inputs to the system (or, more generally, the sequence and timing of its interactions with its environment). Thus, the manifestations of design faults behave as stochastic processes and can be treated probabilistically: to talk about a piece of software having a failure rate of less than, say, 10^{-9} per hour is to say that the probability of encountering a sequence of inputs that will cause a design fault to lead to failure is less than 10^{-9} per hour.

The problem with the experimental statistical approach to assurance for complex software is that the smallest failure rates that can be determined in this way are typically several orders of magnitude greater than those required for safety-critical systems. I explain this in somewhat more detail in the box on page 15.

An Aside on The Infeasibility of Experimental Quantification of the Reliability of Critical Software. It is perfectly reasonable to state requirements for safety-critical systems in statistical terms. For example, catastrophic failure conditions in aircraft ("those which would prevent continued safe flight and landing") must be "extremely improbable." That is, "so unlikely that they are not anticipated to occur during the entire operational life of all airplanes of one type" [FAA88, paragraphs 6.h(3) and 9.e(3)]. A little arithmetic suggests 10^7 hours as the operational lifetime of an aircraft fleet, and hazard analysis might typically reveal ten potentially catastrophic failure conditions in each of ten systems on board the aircraft, so that the maximum allowable failure rate for each is about 10^{-9} per hour [LT82, page 37]. This is indeed the number suggested as an "aid to engineering judgment to help determine compliance" with the requirement for extremely improbable failure conditions [FAA88, paragraph 10.b].[a]

For a simple physical system where breakdown or wearout is the only potential cause for a catastrophic failure condition, experience with similar systems together with testing and analysis, may yield data that can substantiate a claimed failure rate as low as 10^{-9}. Software, however, generally interacts with its environment in such a complex manner that prior experience of its behavior in similar applications may provide relatively little assurance in a new one.[b] Furthermore, most software has significant elements of novelty from one application or version to another, so that experimental determination of software reliability must examine the actual software in the context of its actual application. Furthermore, the test scenarios used to derived reliability estimates must closely approximate in type and frequency the distribution of inputs that will be encountered in operation (this is called the *operational profile*). For required failure rates on the order of 10^{-9}, this means that it will be necessary to construct many millions of the very rare scenarios that will each be encountered only one time in a billion. (Catastrophic failures usually arise in situations compounded by several rare events [Hec93].) Divergence between the test and operational profiles in these remote regions can lead to inaccurate estimates of reliability and spurious assurance of safety.

The difficulty in reproducing the operational profile for rare events, and the time required to perform fault injections and to configure other elements of "all-up" test scenarios limit the feasible failure rates that can be determined empirically to 10^{-4} or 10^{-5} [BF93]—nowhere near the 10^{-9} that is required.

[a]The probability 10^{-9} is applied to complete (sub)system failure, not to any software the system may contain. Numerical estimates of reliability are not assigned to airborne software [RTCA92, Subsection 2.2.3], but the figure gives an idea of the quality required.

[b]In flight tests of the X31, the control system "went into a reversionary mode four times in the first nine flights, usually due to disagreement between the two air-data sources. The air data logic dates back to the mid-1960s and had a divide-by-zero that occurred briefly. This was not a problem in its previous application, but the X31 flight-control system would not tolerate it" [Dor91]. Similarly, much of the software in the Therac 25 medical electron accelerator, which led to massive overdoses of radiation and the subsequent deaths of six patients, had been used in an earlier machine without accident [LT93].

The infeasibility of experimental quantification of reliability for safety-critical software means that its assurance must chiefly be provided by other means. Now, experimental evaluation is not the only means for providing assurance about the behaviors of physically engineered systems. The engineering field concerned normally provides well-validated mathematical models that allow the properties and behavior of a given design to be predicted through calculation—for example, structural engineers can calculate the behavior of a wing spar before it is built. It is one of the distinctions between engineering and craft activities that engineering uses mathematical modeling to predict behavior, whereas crafts use trial and error.

In the next section, I introduce formal methods, which provide a way to move the construction and validation of software away from experiment and adjustment, and towards prediction and calculation.

1.2 Formal Methods

The term *formal methods* refers to the use of mathematical modeling, calculation, and prediction in the specification, design, analysis, construction, and assurance of computer systems and software. The reason it is called "formal methods" rather than "mathematical modeling of software" is to highlight the character of the mathematics involved.

1.2.1 Analytic Formal Methods

Each engineering discipline develops a body of mathematical techniques that are particularly appropriate for modeling and predicting the phenomena relevant to its field. In many cases, the relevant applied mathematics uses partial differential equations to model the variations in continuous physical quantities over time or space. For software, however, the familiar methods of calculus and differential equations are inapplicable because, as noted above, we have to model discrete, rather than continuous quantities.[5] Instead of differential equations, the properties and behaviors we are concerned with are best described in terms of concepts from discrete mathematics: "sets," "graphs," "partial orders," "finite-state machines," and so on. "Calculation" in these finite domains is based on the methods of formal (or mathematical) logic rather than numerical computation. This is because the results we are interested in are logical properties, such as "this system can tolerate any single fault in any component," rather than numerical estimates for some parameter such as lift or drag. To deduce whether a certain logical property follows from descriptions of certain discrete mathematical structures, we have to start from the axioms describing those structures and manipulate their symbols according to certain rules

[5]We have to be careful here to distinguish the mathematics of the domain to which the software is applied (which may, as in the case of control applications, require the evaluation of expressions derived from differential equations) from the mathematics that describes its own operation.

of deduction. This process is more akin to proving theorems in Euclidean geometry than to ordinary numerical calculation, but it shares with calculation the characteristic that it is performed according to strict rules, so that one person can check the work of another and computers can be used to automate some of the steps. The process of manipulating symbols according to certain rules is called "formal deduction" because the legitimacy of the process depends only on the *form* of the symbolic expressions concerned and not on what they are supposed to mean. For example, the transformation of an expression of the form $x^2 - y^2$ into one of the form $(x + y) \times (x - y)$ is legitimate whenever x and y are numbers, independently of whether they represent the mass of planets or the debts of nations.

The particular importance of formal methods to safety-critical systems is that they, uniquely, though subject to caveats I will come to shortly, permit analysis of *all* the behaviors of a software system. This total exploration is the only way to provide assurance that catastrophic failure does not lie hidden among the vast number of possible behaviors. How is total exploration possible? We have already seen that in the absence of continuity there is no basis for extrapolating from tested to untested cases, so how is it that a finite procedure based on formal methods can provide assurance for all the (possibly infinite) behaviors of a software system?

Part of the explanation is that formal methods provide us with powerful tools for identifying and grouping "essentially similar" pieces of behavior together so that all members of a group can be dealt with at a single shot. In empirical testing, we examine only the external manifestations of the system, and our ability to assign these to groups that are essentially similar is very limited because of their discontinuous nature. But in formal methods we examine the internal *design* of the system, where the sources of discontinuity are visible, and we can group little "pieces" of behavior together—all those that result from following a certain path through a certain part of the design, for example. We can then characterize the properties of those pieces by mathematical expressions (i.e., formal specifications), and can deduce the properties of larger pieces of behavior by applying formal deduction to the expressions describing their component pieces. By composing small pieces of behavior together to yield larger and larger parts of the complete behavior, we eventually cover all possible end-to-end behaviors without having to enumerate them explicitly.

Another way of saying this is that formal methods let us calculate the *reasons* why a software design does its job. Software is a designed artifact, consciously constructed to achieve some goal: if we can write down what that goal is, and how the design accomplishes it, then we should be able to construct an argument that explains why we believe the software does its job. Formal methods allow this argument to be reduced to the certainty of calculation.

The "certainty of calculation" needs qualification: for example, the rules of arithmetic reduce addition to the certainty of calculation, but it is easy to make mistakes when adding a long column of figures. The calculations underlying formal

An Aside on Finite Analysis of Infinite Systems. Mathematical logic provides ways to reason about the properties of large or infinite collections of related things in a finite manner: instead of reasoning about the behavior of a system when given the input 1, or the input 2, or ..., logic provides us with methods to reason about its behavior on the symbolic input n, thereby collapsing all the separate cases into one. One method allows us to draw conclusions *for all* values of a given variable by reasoning about a single representative symbolic constant (this process is called *Skolemization*). Another method allows us to deduce properties *for all* values of some ordered domain (e.g., the natural numbers, 0, 1, 2,...) by showing (a) that the property is true for the least element(s) of the domain (e.g., 0), and (b) that when it is true for all members up to some point (e.g., n), then it is also true for the next point (e.g., $n+1$) (this process is called mathematical *induction*). Formal deduction allows properties of a complete system to be deduced by combining basic steps such as these: steps that are required to follow certain rules, and can therefore be checked by others, or by machine.

methods are similarly tedious and error-prone when done by hand, so it is often desirable to automate them. Unfortunately, these kinds of calculations are not so easily mechanized as numerical ones. Whereas the steps of a numerical calculation can be programmed as a deterministic algorithm, selection of the steps in a formal deduction requires either insight, or a heuristically guided trial-and-error search. *Theorem provers* are computer programs that attempt to automate formal deductions through a combination of heuristics and brute-force search; *proof checkers* are programs that leave selection of the steps to an insightful human and simply check that each one is carried out correctly. The most effective automated reasoning tools for formal methods generally combine elements from both theorem provers and proof checkers. For simplicity, the term "theorem prover" is generally used in the following sections to cover all forms of automated deduction.

I said that the ability to represent values symbolically, and hence to group related behaviors together is *part* of the reason why formal methods allow us to consider all the vast numbers of behaviors of a complex software system, and I also said that this claim is subject to certain caveats. Both points concern the fact that formal methods are a *modeling* activity: formal methods do not deal with actual software running on electronic computers interacting with the real environment but with mathematical models of these artifacts. This is exactly similar to the mathematical methods used in other engineering disciplines: a finite-element calculation does not calculate the stress in wing spar, it calculates a representation of the stress in a mathematical model of a wing spar.

Modeling is a source of both weakness and strength. Because a model is not the same as reality, predictions made with its aid may be incorrect. This can be because the model is insufficiently detailed, or because it is plain wrong. It is necessary to guard against these potential weaknesses by carefully *validating* models before trusting to their predictions. The fear of imprecision or inaccuracy sometimes leads to the development of complicated, highly detailed models, but this can vitiate the main purpose for developing a model in the first place: its ability to support tractable analysis. The great opportunity offered by modeling is the freedom to select and simplify the aspects of reality that are to be considered; Newtonian mechanics achieves its effectiveness because it selects for attention just a few properties of bodies and their motions: the mass of a body is considered, but not its volume, or its color. In the same way, formal methods can achieve great effectiveness by focusing on just certain parts of a complex system (e.g., those that give most difficulty, such as redundancy management), and by excluding details that are judged irrelevant (e.g., we may focus on the *algorithms* for redundancy management, and ignore the details of their execution as programs). It requires great skill, judgment, and taste to perform the abstraction necessary to create formal models that are simple enough to be computationally tractable, yet realistic enough to provide accurate predictions and credible assurance.

1.2.2 Descriptive Formal Methods

In the previous discussion, I introduced formal methods by analogy with engineering mathematics, and stated that the purpose of formal methods is to make predictions about the properties and behavior of software, based on calculations performed on a mathematical model of its design. But engineering mathematics is not used only for calculation and analysis: it also provides a vocabulary for describing and documenting designs, and a framework for thinking about them. Thus, aeronautical engineers may speak of "drag divergence" or "flow separation" as concepts independently of particular sets of equations. Formal methods can serve a similar descriptive function for software. That is to say, concepts such as "relations," "functions," "finite state machines," and "universal quantification" can supplement or replace the English prose, pseudocode, and various kinds of diagrams that are traditionally used in documenting requirements, specifications, and designs for software systems. Two benefits can follow from this use of mathematical notation: it can improve the quality of documentation and lead to better communication among those working on the system, and it can supply better ways of thinking about software.

There are two aspects to a software system: control and data. Control is concerned with the selection, timing, and sequencing of the operations performed in a software system. Data is concerned with how information is represented within the computer system and manipulated by its software. The problem with traditional ways of documenting and thinking about both aspects of software is that they are almost entirely operational: software is understood by mentally "executing" it; similarly specifications all too often describe *how* the software works, rather than *what* goals it is to accomplish.

For the control aspect, traditional operational methods can be quite effective for systems composed of a single sequential program: with practice and determination it is often possible to think through the consequences of each alternative at a branch point, and the behavior of loops can largely be understood by mentally considering the cases where each is executed zero, one, and many times. For large, or reactive, or parallel systems, however, these methods become very unreliable: it is hard to comprehend all the possibilities when external events can have an impact at almost any point and we have to consider scenarios such as "what happens if a timer interrupt occurs *here*, and suppose the other processor is in fault-recovery mode, so that it could post a 'need-service' flag at just the instant that...." The box on page 21 describes an example of the difficulties caused when one of the possible scenarios is overlooked.

As with control, the traditional way of documenting and thinking about the data aspect of software is largely in terms of its concrete representations. Thus, much software documentation is composed of pictures or programming-like notation describing how certain information is recorded in the bits of a computer word, how collections of similar items are represented in "arrays" of computer words, and how more complex structures are represented by "pointers" linking "records" together. More modern notations, derived from programming languages like Ada and ideas from object-oriented design [RBP$^+$91], have significantly raised the level of abstraction and improved the organization of data descriptions, but the orientation is still that of implementation.

In contrast to these very operational ways of describing and understanding software systems, formal methods provide ways to document and think about data and control that depend less on mentally tracing execution paths, and more on identifying properties that are to be assumed, established, or preserved. In addition, the mathematical concepts employed in formal methods assist in the construction of more abstract descriptions that state more clearly *what* is to be accomplished without getting caught up in premature details of *how* it is to be done.

Although the main purpose of descriptive formal methods is to improve communications among those working on a software project, and to facilitate informal quality control activities such as reviews and inspections, these methods are not antithetic to the analytical purposes described earlier. Rather, there is a continuum of formal methods ranging from those that are primarily intended to support mechanically checked analysis to those that are primarily intended for documentation.

An Example of the Kind of Fault That Is Hard to Find. While in orbit around Venus, the Magellan spacecraft broke contact with Earth and entered "safing" modes—preempting its scientific mission—on a number of occasions. Extensive efforts were undertaken to find the source of the problem and, after eight months, the most likely cause was identified [KP93]. Two flags determine whether a background task should be run in the otherwise unused time after the end of all the foreground tasks in the current frame and before the end-of-frame interrupt. The "scheduled" flag determines whether a particular task should be run as the background task, and the "active" flag indicates whether this task is an uncompleted activity that should be resumed at an address stored on the stack, or a new one that should start at its entry point. On very rare occasions, the end-of-frame timer interrupt would occur in the instant after one flag had been set to a new value but before the other had been. In particular, a sequence that was invoked when a background task completed could be interrupted after the "scheduled" flag had been turned off, but before the "active" flag could be turned off also. Next time this task was scheduled, the background task manager would mistakenly think it was active and would pop the stack to obtain the address at which to continue its execution. Since the task had, in fact, completed, there was no restart address on the stack, so the value that was popped and used was some random piece of data. As luck would have it, this random address sent the processor to a piece of code where it sat in a tight loop that continually reset the watchdog timer, thereby disabling the very mechanism that was intended to thwart such runaways [Coo93, pp. 209–221].[a]

Computer scientists are thoroughly familiar with the dangers of being interrupted while adjusting critical data structures and will normally arrange for such actions to take place inside a "critical section" that cannot be interrupted. These incur overhead, however, and the Magellan designers thought that in their particular circumstance it was safe to do without this protection. Cooper's book, cited above, aptly conveys the monumental task of trying to diagnose a very rare but devastating misbehavior in a reactive, real-time, parallel software system operating in the presence of faults, when the only way to understand the system is to mentally (or actually) simulate its execution under as many circumstances as possible.

[a]Fortunately, Magellan had multiple levels of redundancy and, although these were intended to cope with hardware, not design, problems, they saved the spacecraft. Specifically, the design fault described above was in the software of the attitude and articulation control system (AACS) computer and although the runaway execution reset the watchdog timer, it did not modulate the "heartbeat" pattern that is placed in memory shared with the command and data subsystem (CDS) computer. When the CDS computer saw the AACS heartbeat cease, it reconfigured the AACS systems, eventually leading to a reboot of the errant AACS computer.

2 Issues and Choices in Formal Methods

I have sketched the rationale for using formal methods in software development and assurance. In this section I describe some of the different varieties of formal methods and some considerations for their selection and use.

2.1 Selection and Abstraction in Applications of Formal Methods

Expertise in formal methods is not widespread, and can be costly to acquire. Furthermore, the resources available for any project are limited, so that effort expended on formal methods may reduce that available for other methods of analysis and assurance. For these reasons, formal methods need to be applied selectively. There are several dimensions in the use of formal methods that permit selective or partial application. I list five of the most important.

- The amount of formality can vary between occasional use of ideas and notation from discrete mathematics in a "pencil and paper" manner to "fully formal" treatments that are checked with a mechanical theorem prover.

- Formal methods can be applied to all, or only to selected, components of the system.

- Formal methods can be applied to selected properties of the system (e.g., absence of deadlock) rather than to its full functionality.

- Formal methods can be applied to all, or merely to some, of the stages of the development lifecycle. If the latter, we can choose whether to favor the earlier, or the later stages of the lifecycle.

- In all cases it is possible to include more or less detail and to choose the level of abstraction at which the formal treatment is conducted.

I examine each of these in more detail below.

2.1.1 Levels of Formality

The very notion "formal" can have different interpretations, and methods differ in the "amount" of formality they employ. An example may help explain this.

Suppose we are to produce a program that will compute the exponential n^m where n and m are integers, and m is nonnegative. One way to do this is by repeated multiplication; it is not a very efficient method, but will serve our purpose. I claim that the following program, written in a generic high-level language, computes the value n^m and leaves it in the variable r:

```
r := 1;
i := m;
while i ≠ 0 do
  r := r * n;
  i := i - 1
endwhile
```

To justify my claim, I present the following proof:

- Each time in the `while` loop, at the point just before the `i ≠ 0` test, the following relationship is true among the variables:

  $$r \times n^i = n^m.$$

  To prove this I consider two cases:

  First time into the loop. We have initialized r to 1 and i to m, so $r \times n^i$ is $1 \times n^m$ and the desired relationship is true.

  Other times around the loop. Assume the previous iteration left $r \times n^i = n^m$; after going round the loop once more, we have replaced r by $r \times n$ and i by $i - 1$, so we need to prove $(r \times n) \times n^{i-1} = n^m$. By arithmetic, the left side equals $r \times n^i$ and the result follows.

- We exit the `while` loop when $i = 0$. Since we know that $r \times n^i = n^m$ at this point, it follows by arithmetic that n^i is 1, and so $r = n^m$ as required.

- To see that the program does always terminates, note that i is initialized to m, which is a nonnegative integer. The value of i is reduced by one each time round the loop, so eventually it will reach zero and the loop will exit.

Although I used a proof here, the process was not particularly formal: I presented the argument in fairly ordinary English, and relied on our intuitive understanding of how the program executes.

A more formal approach would use logical axioms to describe the behavior of the program without requiring us to mentally execute it. As an example, I will use a method that manipulates *Hoare sentences*, which are constructs of the form $\{P\}S\{Q\}$, where P and Q are expressions describing the relationships among the program variables, and S is a piece of program text; the interpretation is that if the relationship P is true before execution of S, and if S terminates, then the relationship Q will be true afterward.

The behavior of a `while` loop is specified by the following axiom.[6]

$$\frac{\{P \wedge B\}S\{P\}}{\{P\}\,\texttt{while}\ B\ \texttt{do}\ S\ \texttt{endwhile}\,\{P \wedge \neg B\}}.$$

[6]In this formula, the symbol $\wedge$ means "and," and $\neg$ means "not."

This says if the Hoare sentence above the line is true, then the Hoare sentence below the line will be true also. If we substitute "`i` $\neq$ `0`" for B, "`r := r * n; i := i - 1`" for S, and "$r \times n^i = n^m$" for P, then we obtain

$$\frac{\{r \times n^i = n^m \wedge i \neq 0\}\texttt{r := r * n; i := i - 1}\{r \times n^i = n^m\}}{\{r \times n^i = n^m\}\,\texttt{while}\ i \neq 0\ \texttt{do r:=r*n; i:=i-1 endwhile}\,\{r \times n^i = n^m \wedge i = 0\}}.$$

Now the expression $r \times n^i = n^m \wedge i = 0$ at the bottom right gives us $r = n^m$, which is what we want to prove, so the next step is to prove the Hoare sentence above the line, which involves the sequential composition of two assignment statements.

The axiom for sequential composition is the following, where both Hoare sentences above the line must be true in order to conclude the one below the line.

$$\frac{\{P\}S_1\{Q\}, \{Q\}S_2\{R\}}{\{P\}S_1; S_2\{R\}}.$$

We substitute "$r \times n^i = n^m \wedge i \neq 0$" for P, "`r := r * n`" for S_1, "$r \times n^i = n \times n^m$" for Q, "`i := i - 1`" for S_2, and "$r \times n^i = n^m$" for R to obtain

$$\frac{\begin{array}{c}\{r \times n^i = n^m \wedge i \neq 0\}\,\texttt{r := r * n}\,\{r \times n^i = n \times n^m\},\\ \{r \times n^i = n \times n^m\}\,\texttt{i := i - 1}\,\{r \times n^i = n^m\}\end{array}}{\{r \times n^i = n^m \wedge i \neq 0\}\ \ \texttt{r := r * n; i := i - 1}\ \ \{r \times n^i = n^m\}}.$$

I won't go into the details, but the axioms that specify assignment statements do allow us to prove the two Hoare sentences above the line. To complete the proof of our program, we need to establish that the initialization statements establish the relationship assumed at the start of the while loop, that is

$$\{m \geq 0\}\,\texttt{r := 1; i := m}\,\{r \times n^i = n^m\}.$$

This follows from the axioms for sequential composition and assignment in a manner similar to that of the previous step, thereby completing the proof of *partial* correctness for our program. Partial correctness establishes that the program gives the right result, *provided that it terminates*. To establish *total* correctness, we must show that the program does indeed terminate; a formal version of the argument for termination can be based on the informal argument that i is decremented each time round the loop, so that the loop termination condition must eventually be satisifed.

This second way to analyze our program using Hoare logic was much more formal than the first: instead of appealing to intuition about how a program executes, we used axioms and substituted program text and mathematical expressions into them. Although we used insight and intuition to decide which axioms to use and what to substitute into them, the subsequent manipulations were quite mechanical and it should be clear that each of the steps we performed could have been checked by computer.

There are advantages and disadvantages to the different levels of formality employed in these two treatments of our example program. In discussing them, it is useful to have a simple scale for identifying levels of formality. I use a three-point scale as follows.

Level 1 formal methods use ideas and notation from discrete mathematics and logic, but within a loose framework, where mathematics, English, diagrams, and other notations are used together. Proofs are careful arguments that are evaluated by whether they persuade reviewers. This is the way most mathematics is done. My first treatment of the exponentiation program was an example of Level 1 formalism.

Level 2 formal methods employ a fixed specification language for documenting requirements and designs. A specification language generally blends concepts from logic, discrete mathematics, and programming into a single notation. Often, the language is supported by tools that check specifications for certain types of errors, and that provide useful functions such as cross-referencing or typesetting. Analyses and proofs are performed by hand and recorded with pencil and paper, but make use of explicit axioms and proof rules that describe the semantics of the languages and methods used. My second treatment of the exponentiation program was an example of Level 2 formalism.

Level 3 formal methods stress mechanized analysis. Their specification languages are generally closer to standard logic than those of type 2 formal methods, and are supported by tools that include proof checkers, theorem provers, or model checkers. The tools that support a Level 3 formal method are often referred to collectively as a *verification system*; such a system could mechanize either the first or the second of the approaches I presented for the exponentiation program. Most current systems use an approach closer to the first than the second. Typically, a program verification system would require that the program is annotated with assertions and a loop invariant as follows:

```
entry assertion $m \geq 0$
r := 1;
i := m;
while i $\neq$ 0 do
  loop invariant $r \times n^i = n^m$
  r := r * n;
  i := i - 1
endwhile
exit assertion $r = n^m$
```

Using the method of *inductive assertions* (the fully formal version of the method I used in the first example), the verification system would then generate the following three *verification conditions*, which correspond to the paths in the program from the entry assertion to the loop invariant, from the loop invariant around the loop and back to the invariant, and from the loop invariant to the exit assertion, respectively.[7]

VC1: $m \geq 0 \wedge r = 1 \wedge i = m \supset r \times n^i = n^m$.

VC2: $m \geq 0 \wedge r \times n^i = n^m \wedge i \neq 0 \wedge r' = r \times n \wedge i' = i - 1 \supset r' \times n^{i'} = n^m$.

VC3: $m \geq 0 \wedge r \times n^i = n^m \wedge i = 0 \supset r = n^m$.

These verification conditions are expressions in ordinary logic (plus arithmetic), and can be proved quite easily by the theorem provers of most verification systems.

The advantage of Level 1 formal methods is the flexibility that is available: notations and techniques can be selected, or invented, to suit the particular problem at hand. These methods can be very effective when used by individuals or small teams possessed of skill and judgment, but the lack of standardized notation and methods can make communication and training difficult across larger groups.

Level 2 formal methods address the problems of communication and training by providing fixed specification notations (Z [Spi93] and VDM [Jon90] are well-known examples) and, usually, a methodology for using them. Individual Level 2 methods are well suited to some types of applications (e.g., data processing), and less well suited to others (e.g., concurrent systems); users must be careful not to stretch their chosen method beyond its limits.

In general, the Level 2 notations are optimized for descriptive, rather than analytic, purposes. If the goal is to use formal methods to calculate properties of a design for the purpose of analysis, then a Level 3 method equipped with appropriate tools will probably be more suitable. It generally requires considerable skill and experience to use Level 3 tools effectively, but they can provide a very high degree of assurance.

2.1.2 Selected Components

Formal methods are generally advocated because it is felt that they can improve quality control and assurance for software. If this is so, then the greatest benefits will be seen when formal methods are applied to the most critical components, and to those for which traditional methods have been found least effective.

[7] The symbol $\supset$ in these formulas means "implies." Also, a prime indicates the value of a variable in the "new" state following execution of a path, and an unprimed name indicates the value in the "old" state at the beginning of the path.

An Aside on Practicality. Readers who found their eyes glazing over at the formulas used to verify the trivial exponentiation program may wonder whether these formal techniques really are practical, and might ask "how am I going to get my engineers to use this stuff?" Privately, they may also wonder "if this stuff is so good, why isn't it used more?"

Before we attribute the slow industrial takeup of formal methods to ineffectiveness, we should remember the comparative youth of the field, and should recall the history of other engineering subjects (see, for example, [Vin90]). Most of the disciplines that we now regard as engineering started as crafts, practiced by experimentalists who learned what would work and what would not by trial and error, and who passed their lore on to their successors by on the job example. Gradually, methods based on science and mathematics started to appear, but they did not immediately displace the traditional methods. Among the reasons for the slow acceptance of mathematical techniques were the conservatism of the traditional practioners[a], the arcane difficulty of the new methods (to those lacking the necessary training), and their initially narrow range of application. The new methods displaced the old as concrete evidence of their superiority accumulated, as good textbooks became available, and as new generations of engineers, trained in the mathematical methods, joined the field. For example, when Donald Douglas, then recently graduated from MIT, started work for the Martin company in 1914, he found Glenn Martin bouncing up and down inside a seaplane supported on wooden trestles "to see if it was strong enough." Douglas noted that "this didn't represent any load that the plane bore in flight." Aided by proper engineering analysis, his first design (the Model S) had almost twice the range and payload of Martin's previous products [Bid91, pp. 85, 86].

It is inconceivable today that an aeronautical engineer could be ignorant of aerodynamics or structural mechanics, still less argue that such mathematical modeling is irrelevant to the practical business of designing airfoils. I expect formal methods eventually to play a similar role in software engineering.

The practicality and cost/benefit of formal methods are heavily dependent on the type of applications considered. Program verification of the kind illustrated in my examples is undeniably tedious and expensive (see, for example the figures quoted in [GH90]), and must compete with traditional methods that are quite effective. My opinion is that the greatest benefits are likely to be found when formal methods are applied to the *hardest* and *most difficult* problems—where traditional methods are ineffective or unavailable. Examples of hard problems are those involving distributed and concurrent execution and, especially, redundancy management [ORSvH95]. These problems can be considered practical because, though hard, they are not large and can therefore be undertaken by a few highly skilled people. It is not necessary to train every programmer to get valuable returns from formal methods. Another opportunity lies in problems where formal methods can be massively automated. Example include certain kinds of protocols [HK90, CGH^{+}95] and hardware designs [MS95].

[a]For example, Tesla quit Edison's laboratory after less than a year complaining of Edison's preference for empirical methods "knowing that a little theory and calculation would have saved him 90% of his labor" [Bur93].

Standards and guidance documents for safety-critical systems generally rank software components by criticality according to the severity of the consequences that could result from their malfunction. For example, DO-178B identifies software criticality levels A through E according to the severity of their potential failure conditions (i.e., Level A is software whose malfunction could contribute to a catastrophic failure condition) [RTCA92, Subsection 2.2.2]. Software criticality level determines the amount of effort and evidence required to show compliance with certification requirements. It provides a natural criterion for selecting components for which formal methods should be considered.

Another criterion that should be considered is the likely effectiveness of formal methods *versus* traditional methods for quality control and assurance. It is to be expected, and there is some evidence to support the expectation [Lut93], that the intrinsically hard design problems tend to be the most prone to faults, and the most resistant to traditional means of assurance. These intrinsically hard problems generally involve complex interactions, such as the coordination of distributed, concurrent, or real-time computations, and redundancy management. It requires great skill to address these problems using formal methods, but the number and size of these problems may not be large. Hence, as noted in the box on page 28, the greatest return on formal methods may be obtained when relatively few, very highly skilled people apply formal methods to the hardest and most critical problems.

2.1.3 Selected Properties

Just as some components of a system may be more suitable for formal methods than others, so different *properties* of those components may be more suited than others to formal treatment. As with components, suitability may be determined by criticality or by the effectiveness of formal methods compared with other methods. For example, the important property of a particular component may not be that it does its job (there may be backups to accomplish that), but that it is free of "specific anomalous behaviors" [RTCA92, Section 2.6]. Negative properties such as this (i.e., properties that state what must *not* happen) are particularly difficult to test and can be good candidates for formal analysis.

2.1.4 Lifecycle Stages

The example shown earlier involving the exponentiation program illustrated the use of formal methods in the late lifecycle. That example was one of *program verification*, where an executable program is proved to satisfy its detailed requirements. Other applications of formal methods focus on activities of the early lifecycle, such as the documentation and analysis of requirements, and on those of the intermediate lifecycle stages such as the documentation of interfaces and the systematic refinement of requirements into designs, or designs into implementations.

Late-lifecycle applications of formal methods such as program verification were among the earliest to be developed and are now widely known and well understood. But precisely because this part of the lifecycle is well understood, informal methods and engineering practice have achieved a considerable degree of practical effectiveness: sequential programming and gate-level design are not major sources of difficulty or faults today (at least, not in those industries that practice stringent software quality control and assurance). For example, Lutz [Lut93] reports on 197 significant software faults detected during integration and system testing of the Voyager and Galileo spacecraft. Only three of these faults were programming errors; the vast majority were requirements problems. Similarly, Keutzer [Keu91] reports that fully half of all ASICs are faulty on first fabrication, and that these faults are invariably due to errors in requirements or high-level design: no errors are reported in implementation below the register-transfer level.

It is now generally recognized that faults introduced in the early lifecycle are among the most difficult and expensive to detect and eradicate later; furthermore, the most serious failures are often traced to undetected faults that were introduced early in the lifecycle. One explanation for the intractability and persistence of faults introduced in the early stages of development is that there are few good methods for validating the products of these stages: requirements and early design descriptions do not lend themselves to execution and tests. Formal methods can help overcome this difficulty by allowing early specifications to be challenged and explored through theorem proving: a challenge of the form "if this specification says what it should, then the following ought to follow" can be formulated as a putative theorem that should be provable from the specification. Rapid prototyping can serve some of the same ends, but it is not always straightforward to distinguish those properties that are truly entailed by the requirements or design descriptions being validated from those that are accidental to the prototype. Unlike a prototype, a formal requirements specification can be validated experimentally without necessarily being executable.

2.1.5 Abstraction

Abstraction is one of the most powerful tools for gaining intellectual mastery of complex systems: it allows us to ignore the irrelevant and simplify the relevant so that the essential matter of concern is exposed to scrutiny in its clearest and most tractable form. Abstraction is also a crucial factor in controlling the size of a formal development, and the effort required for its analysis.

One example of abstraction considers the *algorithms* that underlie a design, rather than their expression as *programs*. For example, the repeated multiplication that underlies the exponentiation program considered earlier can be abstracted to an algorithm represented by the following recursive function.[8]

$$n^m \overset{\text{def}}{=} \textbf{if } m = 0 \textbf{ then } 1 \textbf{ else } n \times n^{m-1} \textbf{ endif}$$

This specification can be analyzed using ordinary logic (i.e., without the special machinery for program verification, such as Hoare sentences) and it is comparatively easy to establish its properties, even when using Level 3 formal methods and mechanized theorem proving. The transition from this abstract algorithm to the concrete program can be justified either informally, or using formal methods of any of the three levels.

When formal methods are applied to algorithms, there is further scope for abstraction in the choice of how much detail to include. For example, one of the important algorithms in fault-tolerant systems is one for distributing sensor samples consistently in the presence of faults [LSP82]. This is a distributed algorithm, and if we are concerned with issues of the timing and transfer of the messages that are communicated in the algorithm, then it is necessary to model these mechanisms in some detail, and the analysis will be correspondingly detailed and lengthy [LM94]. But if we are mainly concerned with the fault masking properties of the algorithm, then the mechanisms of distributed computation and communication can be ignored and the algorithm can be modeled as a recursive function, in which form its analysis is quite straightforward. Certain details of behavior, and therefore the opportunity to detect some potential faults, are missing in the more abstracted representation. On the other hand, the economy provided by ignoring details of communication can allow us to *increase* detail elsewhere, and this may be a useful tradeoff. In this particular example, it is possible to increase the number of different types of faults that are considered in the most abstracted representation, and this allows the fault tolerance of the algorithm to be analyzed in greater detail (and reveals a bug in a published algorithm) [LR93].

As this example makes clear, abstraction is closely related to the modeling activity that is inherent in formal methods. The whole basis of formal methods is to create mathematical models of certain physical and computational phenomena and to make predictions about these phenomena through analysis of the models. Abstraction is concerned with how much, and what, detail to include in the model, and how to represent it. Validity of the predictions made through use of formal methods requires that the abstraction retains all salient details and that their formal representation is faithful to reality. Tractability of formal analysis, on the other hand, requires that the abstraction is ruthless in expelling all irrelevant detail. Ability to resolve this tension between too much and too little abstraction is the most

[8]Actually, this specification is generally taken as the axiomatic *definition* of exponentiation and is used to establish lemmas such as $n^1 = n$ and $n^{m_1+m_2} = n^{m_1} \times n^{m_2}$ and the correctness of more efficient algorithms.

important, and rarest, of the skills required to make effective use of formal methods.

2.2 The Varieties of Formal Specifications

Formal methods embrace a variety of approaches that differ considerably in techniques, goals, claims, and philosophy. The previous section discussed some of the important differences, such as whether formal methods are used primarily for descriptive or for analytic purposes, the level of formality employed, and the stage(s) of the software development lifecycle to which formal methods are applied. The different approaches to formal methods tend to be associated with different kinds of specification languages. Conversely, it is important to recognize that different specification languages are often intended for very different purposes and therefore cannot be compared directly to one another. Failure to appreciate this point is a source of much misunderstanding. In this section I briefly introduce some of the main varieties of specification languages and indicate their applications.

2.2.1 Model-Oriented Specifications

If specification or annotation of programs is the goal, then the formal notation employed should generally be close to, though more abstract than, that of programming, with operations changing values "stored" in an implicit system "state," with data structures described fairly concretely, and with control described in operational terms.

Formal notations with these characteristics are often described as *model oriented*, meaning that desired properties or behaviors are specified by giving a mathematical model that has those properties. For data structures, these models are often constructed from the notions of set theory using sets, functions, relations, and so on. A *pushdown stack*, for example, can be modeled by a pair consisting of a natural number (the *pointer*) and a function (the *stack*) from natural numbers to the type of value being stacked. (This can be thought of as an array with the contents of the pushdown stack occupying positions $1 \ldots pointer$; the empty stack is indicated by $pointer = 0$.) The "top" of the stack is the value of the function at the argument indicated by the pointer; the stack is "popped" by decrementing the pointer, and a value x is "pushed" on to the stack by incrementing the pointer and modifying the function so that it takes the value x at the argument indicated by the pointer.

To describe control, model-oriented notations for sequential programs generally provide sequential composition and *if-then-else* selection. Explicit loop constructs are not needed since their effects can usually be specified more abstractly using quantification. For example, whereas in programming we would use a loop to search for the least value stored in a pushdown stack, we can formally specify this value as the l such that (a) *for all* ($\forall$) natural numbers up to the pointer, the value of the

An Aside on Notation. Representative notation for a model-oriented specification of a pushdown stack is the following, where a prime indicates the value of a variable in the "new" state following the operation and an unprimed name indicates the value in the "old" state prior to the operation. The $\oplus$ operator indicates function modification (also called "overriding").

top	is	$stack(pointer)$, provided $pointer > 0$
pop	is	$pointer' = pointer - 1$, provided $pointer > 0$
push	is	$pointer' = pointer + 1 \wedge stack' = stack \oplus \{pointer' \mapsto x\}$

stack at that point is no less than l, and (b) *there exists* ($\exists$) a natural number less than or equal to the pointer such that the value of the stack at that point equals l. Typical notation for specifying this is the following:

$$(\forall p : 1 \leq p \leq pointer \bullet stack(p) \geq l) \wedge (\exists p : 1 \leq p \leq pointer \bullet stack(p) = l).$$

A disadvantage of model-oriented specifications is that they can be overly prescriptive: suggesting how something is to be implemented, rather than just the properties it is required to have. For example, even though the specification of the *least* function does not prescribe an algorithm, it is stated in terms of the pointer and array model, and so it would be fairly difficult to use this specification to establish correctness of an implementation that used linked lists instead.

2.2.2 Property-Oriented Specifications

In contrast to the model-oriented style of specification that is often preferred for program-level descriptions, specifications of early-lifecycle products such as requirements commonly use *property-oriented* notations. These notations use an axiomatic style to state properties and relationships that are required to hold of the component being described, without suggesting how it is to be achieved. To specify a pushdown stack, for example, a property-oriented notation would state the relationships that are required to hold among the operations "top," "pop," and "push": namely, that a push followed by a pop leaves the stack unchanged, and a top following a push returns the value that was pushed onto the stack.

Sequential control in property-oriented specifications is generally modeled by functional composition. For example, a push followed by a pop is specified by $pop(push(s, x))$ rather than by the more operational $push(x); pop$ (where the semicolon indicates sequential composition and the state of the stack is implicit in the operational specification). Iteration is generally modeled by quantification or recursion. For example, the least element in a stack can be specified by the following

An Aside on Notation. Representative notation for a property-oriented specification of a pushdown stack is the following. Note that the value of the stack is supplied as an explicit argument (here s) to the operations, rather than being the implicit value of a program "state."

$$\begin{array}{lll}\text{axiom } \mathit{pop\_push} \text{ is } & \mathit{pop}(\mathit{push}(s,x)) & = s \\ \text{axiom } \mathit{top\_push} \text{ is } & \mathit{top}(\mathit{push}(s,x)) & = x\end{array}$$

recursive function:

$$\mathit{least}(s) = \textbf{if } \mathit{empty}(s) \textbf{ then } \infty \textbf{ else } \mathit{min}(\mathit{top}(s), \mathit{least}(\mathit{pop}(s))) \textbf{ endif.}$$[9]

Notice that although the stack operations are specified in a property-oriented style, this specification of the *least* function has an algorithmic flavor; the presentation in the previous section used a model-oriented specification for the data structure, but the specification of the *least* function was property-oriented (relative to the model of the data structure), rather than algorithmic. These mixed modes of expression are not uncommon.

An advantage of property-oriented over model-oriented specifications is that it is possible merely to *constrain* certain relationships or values, without having to define them exactly. On the other hand, it is very easy to write conflicting constraints that cause the specification to become inconsistent; inconsistent specifications are unimplementable, and are very dangerous because they can be used to prove anything. Some Level 3 specification languages provide ways to ensure or demonstrate that property-oriented specifications are consistent (see section 2.3.1, and page 36 in particular).

2.2.3 Specifications for Concurrent Systems

Concurrent and distributed systems can be specified in a variety of styles. One style takes some form of communication as primitive and has programming-like features for sending and receiving values. This style has a model-oriented flavor and is often referred to as *process algebra*. Another style takes shared variables as the primitive means of communication and often uses *temporal logic* to allow specification that a

[9]The treatment here of the empty stack (setting its *least* value to infinity), is a little suspect, though it can be made rigorous. Also, I have not explained how the *empty* predicate is defined. Taking care of these difficulties in a fully satisfactory manner would require more space than I wish to allocate, but they give a hint of the technical details that must be dealt with in a formal specification notation.

property should hold "henceforth" or "eventually" on some or all execution paths. This style has a property-oriented flavor. Methods associated with a kind of analysis known as *model checking* use one type of description (a kind of state machine) to specify the system concerned, and another (a kind of temporal logic) to specify the properties required of it.

Further distinctions concern whether concurrent activities are considered to occur simultaneously ("true" concurrency) or alternately ("interleaving" concurrency), and whether consideration of time is restricted to the order in which events happen, or whether duration is considered ("real-time" logics).

2.3 The Varieties of Formal Analysis

Earlier, I observed that one of the most significant differences among formal methods concerns whether their primary purpose is description or analysis. In fact, this distinction is too coarse: we have to ask what *kind* of analysis. The strongest kind of analysis is one that takes a formal description and predicts the behavior of a system satisfying that description. This is the kind of analysis that most closely corresponds to the use of applied mathematics and calculation in traditional engineering fields. If this is the goal desired of formal methods, then the modeling and notational techniques employed should favor efficient deduction, whereas more weight should be given to the "readability" of the notation when descriptive purposes are paramount. In an ideal world, one technique would serve both ends but, in the present state of the art, those notations that are considered most "readable" are much less tractable for automated reasoning than notations designed for that purpose—conversely, notations designed for automated reasoning tend to have a rather austere and forbidding appearance.[10]

In between the purely descriptive uses of formal specifications and those that use automated deduction to make general predictions of behavior, there are many intermediate kinds of analysis that perform formal calculations to establish limited properties of a specification.

2.3.1 Consistency Analysis and Typechecking

One kind of analysis does not attempt to deduce specific properties of the system described by a formal specification; instead it attempts to deduce whether such a system could *exist*. In other words, it checks whether the specification is sufficiently well formed to be a description of *some*thing.

One very important well-formedness property is consistency: if a specification states two contradictory things, then it cannot describe a real system and is therefore

[10]The influence of notation can be illustrated by comparing Arabic with Roman numerals: for small numbers, at least, Roman numerals are more "readable" (e.g., III is more suggestive of the concept "three" than is 3), but they are much less effective for calculation than are Arabic numerals.

useless as a specification. Certain types of specification lend themselves to systematic checks for consistency but, as always, there are tradeoffs involved. For example, specifications that allow new concepts to be introduced only by definition in terms of existing concepts can easily be checked for consistency; however, such specifications are purely constructive (i.e., strongly model oriented), and are unattractive for some purposes. In particular, constructive definitions are an unnatural way to state assumptions about the environment in which a system is to operate; axioms are more natural for this purpose, since our goal is to describe the environment, not to implement it. Consistency for axiomatic specifications can be established by showing that the axioms are true of some constructively defined "implementation." This implementation need not be efficient or realistic, it just has to exist.

It is shockingly easy to write formal specifications that are inconsistent; consequently, any specifications offered in support of certification for safety-critical systems should be furnished with evidence for their consistency.

Some specification languages allow "types" to be given for entities appearing in specifications. Types are familiar from programming languages such as Ada, where variables can be declared as `integer` or `boolean` and the compiler will generate an error message if an attempt is made to multiply an integer by a boolean. The types in a specification language can be more sophisticated (since they do not have to have a direct implementation), and the checks that are performed can be more elaborate. A computer program that checks specifications to ensure that entities are always used in ways compatible with their types is called a *typechecker*; it can be seen as a tool that performs a specialized kind of formal deduction (it attempts to prove the theorem "this specification is type-correct"). If the typechecker is allowed to use general-purpose theorem proving, rather than just perform algorithmic checks like a programming-language compiler, then the type system can become very sophisticated, and typechecking becomes a very powerful way to detect errors in a specification.

Specification languages based on certain kinds of mathematical logic (notably, *higher-order* logic) have to use types to keep the logic consistent; types are technically optional for other kinds of logics (where they are sometimes called *sorts*). Ordinary set theory is untyped; when types are added (as in Z, for example), the result is a little awkward in that either some of the conveniences of set theory (e.g., nonhomogeneous sets) must be given up, or typechecking must be rather weak. Within these constraints, selection of a typed or an untyped specification language is often considered a matter of personal preference: some people value the early error detection of typechecking, others find the restrictions imposed by types to be irksome. However, those who prefer to forsake types should be expected to provide other, equally strong, evidence for the properties that would be established by typechecking.

In my opinion, strong typechecking (the stronger the better) should always be required for formal specifications offered in support of certification for safety-critical systems.

2.3.2 Validating Formal Specifications

Predictions are based on a mathematical model of the system; if the model is inaccurate, the predictions may not be true of the real system. It is therefore necessary to *validate* the accuracy of the model very carefully. Formal methods are no different in this regard than the mathematical methods used in any other engineering field. Consistency analysis and typechecking provide evidence that a formal specification means something, but additional evidence is required to establish that it means what is intended. Reviews are one way to develop this evidence, but formal specifications can also support more analytical methods.

Animation

One way to gain confidence in a formal specification is to "test" it on a few small examples. This kind of examination of a specification is sometimes called *animation*. Some kinds of formal specification can be "executed" using highly efficient forms of deduction, so that test cases can be run directly against the specification. Model-oriented specifications tend to lend themselves more naturally to direct execution than do property-oriented specifications. Note that executability may be at odds with other desirable properties of a specification (such as abstractness, and nonprescriptiveness) [HJ89]. For specifications that cannot be executed directly, it may be desirable to construct a simulator or rapid prototype for testing purposes.

Formal Challenges

Formal specifications can also be explored by posing and proving putative theorems that I call *challenges*: "if this specification says what it should, then the following ought to follow." For example, suppose we had specified the operation of *sorting* a sequence; we might then ask whether sorting an already sorted sequence leaves the sequence unchanged (i.e., whether sorting is idempotent). That is, we might ask whether

$$sort(sort(x)) = sort(x)$$

is a theorem of the specification (assuming *sort* is a function that takes a sequence as argument and returns the sorted sequence as its value). Gerhart and Yelowitz [GY76] describe how early formal specifications of sorting were deficient in that they required the output of the operation to be ordered, but neglected to stipulate that it should also be a permutation of the input. An attempt to prove the theorem above would reveal such inadequacies.

Animation could examine the putative theorem for a few representative values of the input x, but a formal challenge would force consideration of all inputs. For some specifications of sorting, this could lead to the discovery that the putative theorem is unprovable; further examination might then lead to the notion of a *stable* sort (one that does not reorder elements of the sequence that are equivalent with respect to the ordering criterion, but distinguishable in other ways). We could then decide whether stability was important to our application and, if so, could adjoin it as an additional requirement of the sorting specification.

Notice how this process of subjecting specifications to formal challenges probes the completeness as well as the correctness of specifications. Data from the Jet Propulsion Laboratory indicates that two-thirds of the defects in requirements specifications are omissions [KSH92], so that systematic methods of exploring completeness are highly desirable.

Challenges can be undertaken at any level of formality, but I believe that all those who write or review formal specifications should have experience in challenging specifications at Level 3 using a mechanized proof checker or theorem prover. Those who learn a formal specification language from textbooks or training courses, but who do not experiment with mechanically checked challenges, are in a position similar to those who would learn a programming language without the opportunity to execute programs. In fact, their position is worse because experience with other programming languages is likely to help them learn a new one, whereas many of those learning a formal specification language are receiving their first exposure to formal methods, as well as to abstract and axiomatic forms of expression. Just as the failure of an "obviously correct" program teaches us that programming is difficult, so the discovery through dialog with a theorem prover that an expected property is not entailed by an "obviously correct" formal specification teaches us that specification is no easier than programming. In my experience, we all have to learn this for ourselves: only the personal shock of discovering egregious errors in our own specifications teaches us the requisite humility.

2.3.3 Predicting Behavior and Verifying Refinement

Formal challenges probe a formal specification by asking whether it entails certain expected properties. Generally, these properties are special cases, or fragments of the overall requirements. Once a specification has been sufficiently validated in this way, it is possible to examine it for the properties of real interest, and to verify some steps in its refinement toward implementation.

The behavior of a system generally has many aspects and formal methods are usually not used to examine every aspect, but only those that are important to a particular analysis: for example, we may want to know whether a system can deadlock, or whether it can survive any single fault, or whether a response is always delivered within a certain time. Some formal methods are especially well suited to the calculations necessary to predict certain kinds of properties, others are well suited to certain kinds of systems.

State Exploration

Certain kinds of formal methods allow automatic, brute force exploration of all possible behaviors, provided there are not too many of them (the maximum number depends on many factors and increases as technology improves, but the typical range is from tens of thousands to tens of millions). Hardware, and distributed algorithms such as protocols are particularly suitable for this kind of examination through exhaustive *state exploration* [DDHY92] (related technologies are known as *model checking* [CGL94] and *language inclusion* [HK90]).

A specification may admit too many behaviors for state exploration to succeed, but it may be possible to develop a "downscaled" version that can be examined in this way. For example, a communications protocol may be designed to move arbitrary data reliably over a faulty channel using sequence numbers that cycle through the range $0 \ldots 255$. For state exploration, we could downscale the protocol to consider just one or two different data values, and with sequence numbers restricted to 0 and 1. Experience indicates that examining *all* behaviors of a downscaled design is often a more potent validation and assurance technique than examining *some* of the behaviors of the full design. Downscaling is closely related to abstraction; the difference is that abstraction is generally used in verification, whereas dowscaling is generally used for debugging and can be useful even if crude. (Generally, we require that if a property is verified of an abstraction, then it should also be true of the full specification, whereas for a downscaled specification we only require that bugs in the full specification should be likely to show up in the downscaled version.)

Verifying Desired Properties

Formal methods provide the most searching examination and the strongest assurance when proofs are used to verify significant application properties. The process is essentially similar to that used in challenging specifications, except that the properties verified are of external significance, and the proofs are usually more difficult and longer. The basic idea is that we construct a formal specification of the requirement, design, or algorithm concerned, and also a formal statement of the property it is desired to satisfy, and then try to prove that the one implies the other. In practice, the first proof attempt seldom succeeds; instead it usually reveals the need for adjustments to the specification, or to the statement of the desired property or the assumptions under which it is expected to hold. Generally, this process of adjustment is iterated several times as renewed proof attempts reveal additional subtleties. My experience is that the process is always very enlightening, particularly when conducted with the full rigor of Level 3 formal methods, and leads to greatly enhanced understanding of the specifications and properties under examination.

If all goes well, the adjustments will converge and we will finally obtain a satisfactory proof. Subject to caveats on the fidelity of the modeling employed (these were mentioned at the end of Section 1.2.1 on page 18 and must be ensured by reviews and by validation as explained in the previous section), and on the proof being performed without error (this is where mechanized proof checking is valuable, but the mechanization is intended to enhance, not replace, human judgment and responsibility), the proof provides strong assurance that the specified artifact indeed satisfies the desired property. In addition, the process of formal modeling and proof generally provides other, incidental benefits: the discovery and correction of faults, complete enumeration of assumptions, sharpened statements of properties assumed or satisfied, streamlined arguments, and an enhanced understanding that can lead to improvements in design or assurance. Furthermore, formal specifications and proofs are a reusable intellectual resource that can be particularly valuable—potentially, a corporate asset—when design changes on the larger scale require modifications to the component under consideration: highly automated Level 3 verifications, in particular, can often verify slightly modified designs or properties with very little extra effort, and with the same degree of rigor as the original case—something that is very difficult to achieve with reviews.

An Aside on Mechanized Proof Checking. The effectiveness of mechanized proof checkers and theorem provers for formal methods is advancing very rapidly. For example, mechanized verification of the microcode for a simple processor called "Tamarack" represented a significant challenge just five years ago [Joy89], whereas it can now be done completely automatically in about five minutes [CRSS94]. Progress is uneven however, and the amount of human time and effort required to undertake a Level 3 analysis can vary by two orders of magnitude or more from one verification system to another. Potential users should be skeptical of developer's own assessments of where their verification system stands in the "power rankings," and of impressions gained from small examples; instead, they should evaluate candidate systems on full-size examples representative of the intended application.

Verifying Design Refinement

Verification of requirements, designs, and algorithms against desired properties is typically an activity of the early lifecycle. In the later lifecycle, the task of verification is generally to demonstrate that (the specification of) a design at one level is implemented correctly by another at a more detailed level. Generally an *abstraction function* is used to translate the terms of the lower-level specification into those of the upper-level and the task is then to prove that this function has the properties of a homomorphism. The task becomes more complicated when the two levels operate at different granularities of time, and especially when the timing relationship between them is variable (as, for example, in a pipelined microprocessor, where the implementation may take a variable number of cycles to complete an instruction, depending on whether the pipeline stalls, and other details that are hidden at the upper level).

Design refinement can be verified using any level of formality but, once the specifications get reasonably large, it is difficult to construct all the proof obligations to establish the homomorphism without mechanized assistance. On the other hand, the required proofs can be rather repetitive, so that investment in automation is often very worthwhile.

3 Formal Methods and Certification

This section is concerned with the use of formal methods in support of certification for critical systems. First, I present my general recommendations; these are followed by more specific recommendations presented in the context of a commentary on the guidelines for the certification of software for civil aircraft.

3.1 General Recommendations

Formal methods should be part of the education of every computer scientist and software engineer, just as the appropriate branch of applied mathematics is a necessary part of the education of all other engineers. Formal methods provide the intellectual underpinnings of our field; they can shape our thinking and help direct our approach to problems along productive paths; they provide notations for documenting requirements and designs, and for communicating our thoughts to others in a precise and perspicuous manner; and they provide us with analytical tools for calculating the properties and consequences of the requirements and designs that we document.

However, it will be many years before even a small proportion of those working in industry have been exposed to a thorough grounding in formal methods, and it is simply impractical to demand large scale application of formal methods in critical software—and unnecessary too, since industry seems to be doing a mostly satisfactory job using nonformal methods.[11]

Nonetheless, I believe industry should be strongly encouraged to develop and apply formal methods that will permit more complete analysis and exploration of those aspects of design that are least well covered by present techniques. These arise in redundancy management, partitioning, and the synchronization and coordination of distributed components, and primarily concern fault tolerance, timing, concurrency, and nondeterminism. Scrupulous informal reviews, massive simulation, near-complete unit testing of components, and extensive all-up testing do not provide the same level of assurance in these cases as they do for sequential programs—because the properties of interest are not manifest in individual components, and because distributed execution is susceptible to subtle variations in timing and fault status that are virtually impossible to set up and cover adequately in tests.

These formal analyses should be additional to those presently undertaken and can increase assurance without necessarily being totally comprehensive: the value of formal methods lies not in eliminating doubt but in circumscribing it. For example, in addition to all the other assurance techniques that may be applied, it will be valuable to *prove* that mode-switching logic does not contain states with no escape,

[11]The appalling safety record of the Airbus A320 aircraft [BCAG95] seems attributable to poor human factors rather than to specific software faults [Mel94].

and that sensor data is distributed consistently despite the presence of faults. These are not the only properties required of mode switching and sensor distribution, but they are among the most crucial and among the most difficult to assure using traditional methods. To deal with such problems using current technology for formal methods it will often be necessary to abstract away irrelevant detail, and possibly to simplify even relevant detail. Doing so while continuing to model the issues of real concern in a faithful way requires considerable talent and training. On the other hand, since we will be dealing only with relatively small, albeit crucial, elements of the system, the number of people required to possess that talent and training in formal methods will be small.

The benefit provided by these formal analyses is a *complete* exploration of a *model* of possible behaviors. Subject to the fidelity of the modeling employed (which must be established by extensive challenge and review), we will be assured that certain kinds of faults are not present at the level of description and stage of the lifecycle considered. One source of doubt will have been eliminated, and others posed more sharply. Admittedly, this does not guarantee that the implementation will not reintroduce the very faults that have been excluded by the formal analysis, but current practices seem effective at tracing implementations. As resources and capability permit, it will be worth seeing if formal methods can increase assurance for these aspects also, but initially we should focus on cases where current practice seems weakest, not where it seems effective. By that measure, other promising applications for formal methods are in the general area of requirements specification and analysis—where current processes, though fairly effective, are ad-hoc and unstructured.

3.2 Interpretation for DO-178B

The RTCA ("Requirements and Technical Concepts for Aviation, Inc.") document known in the USA as DO-178B [RTCA92] and in Europe as EUROCAE ED-12B provides industry-accepted guidelines for meeting certification requirements for software used in airborne systems and equipment, and is incorporated by reference into European and United States regulatory and advisory documents. DO-178B provides guidelines and does not lay down specific certification requirements: those are based on existing regulations or special conditions decided by the certification authority in consultation with the applicant.

Formal methods are not specifically endorsed by DO-178B (in contrast to certain other guidelines and standards that do recommend or require their use [MOD91]), but are included among the "alternative methods" discussed in its section 12.3.

12.3. Alternative Methods: *Some methods were not discussed in the previous sections of this document because of inadequate maturity at the time this document was written or limited applicability for airborne software. It is not the* *intention of this document to restrict the implementation of any current or future methods. Any single alternative method discussed in this subsection is not considered an alternative to the set of methods recommended by this document, but may be used in satisfying one or more of the objectives of this document.*

...

12.3.1. Formal Methods: *Formal methods involve the use of formal logic, discrete mathematics, and computer-readable languages to improve the specification and verification of software. These methods could produce an implementation whose operational behavior is known with confidence to be within a defined domain. In their most thorough application, formal methods could be equivalent to exhaustive analysis of a system with respect to its requirements. Such analysis could provide:*

- *Evidence that the system is complete and correct with respect to its requirements.*
- *Determination of which code, software requirements or software architecture satisfy the next higher level of software requirements.*

The goal of applying formal methods is to prevent and eliminate requirements, design and code errors throughout the software development processes. Thus, formal methods are complementary to testing. Testing shows that functional requirements are satisfied and detects errors, and formal methods could be used to increase confidence that anomalous behavior will not occur (for inputs that are out of range) or unlikely to occur.[12]

Section 12.3.1 of DO-178B recognizes different levels of formality and rigor in applications of formal methods.

Formal methods include these increasingly rigorous levels: [13]

- *formal specification with no proofs.*

[12]I find this sentence difficult to interpret. I think what is intended is that testing provides assurance that the normal behavior of the software is satisfactory; formal methods can extend that assurance by considering all possible behaviors, including those induced by rare or anomalous combinations of inputs and other circumstances.

[13]These levels do not correspond to those I call 1, 2 and 3: DO-178B does not identify methods corresponding to my Level 1 (semiformal mathematical notation and proofs), and subdivides my Level 2 in two according to whether manual proofs are performed; we agree on level 3. The reason I do not subdivide my Level 2 is that, as explained in Section 2.3.2 on page 37, I attach little credibility (or utility) to specifications whose consequences have not been challenged by proof. For safety-critical applications, I believe that my interpretation of Level 1 rigor is preferable to DO-178B's.

- *formal specifications with manual proofs.*
- *formal specifications with automatically checked or generated proofs.*

> *The use of formal specifications alone forces requirements to be unambiguous. Manual proof is a well understood process that can be used when there is little detail. Automatically checked or generated proofs can aid the human proof process and offer a higher degree of dependability, especially for more complicated proofs.*

Section 12.3 of DO-178B provides guidance in using an alternative method.

> *An alternative method cannot be considered in isolation from the suite of software development processes. The effort for obtaining certification credit of an alternative method is dependent on the software level and the impact of the alternative method on the software lifecycle processes. Guidance for using an alternative method includes:*
>
> **a.** *An alternative method should be shown to satisfy the objectives of this document.*
>
> **b.** *The applicant should specify in the Plan for Software Aspects of Certification, and obtain agreement from the certification authority for:*
>
> **(1)** *The impact of the proposed method on the software development processes.*[14]
>
> **(2)** *The impact of the proposed method on the software lifecycle data.*[15]
>
> **(3)** *The rationale for use of the alternative method which shows that the system safety objectives are satisfied.*
>
> *The rationale should be substantiated by software plans, processes, expected results, and evidence of the use of the method.*

The effort required to satisfy these guidance items will depend on the extent to which formal methods replace, rather than merely supplement, traditional lifecycle processes and data. Note, however, that even if formal methods are truly supplementary to the traditional processes, there will still be some impact on the lifecycle processes and data that the applicant should discuss and explain. For example, if descriptive formal methods are used to supplement traditional documentation, then we have to ask which is the primary description, how is consistency established

[14]Software development processes, discussed in Section 5 of DO-178B, include software requirements analysis, design, coding, and integration.

[15]Software lifecycle data, which are discussed in Section 11 of DO-178B, are produced "to plan, direct, explain, define, record, and provide evidence of activities throughout the software lifecycle."

between them (and maintained through changes), and which is the target of the reviews and analyses performed for verification?

Oddly, the guidance for alternative methods in Section 12.3 of DO-178B does not explicitly call for an assessment of the impact of the proposed method on verification activities. In my opinion, however, this is implicit in the requirement (quoted in paragraph 12.3.b(2), above) to consider the impact of the method on lifecycle data—since these data record the results of verification activities. Verification is one of what DO-178B calls the “integral processes” which

> *...support the development processes by ensuring the correctness and quality of all processes and the delivered software.*

The integral processes comprise software verification, software configuration management, software quality assurance, and certification liaison. They are discussed in Sections 6 through 9 of DO-178B; in particular, technical guidance on verification is found in Section 6.[16]

6.0. Software Verification Process:

> ...
>
> *Verification is not simply testing. Testing, in general, cannot show the absence of errors. As a result, the following subsections use the term “verify” instead of “test” when the software verification objectives being discussed are typically a combination of reviews, analyses, and tests.*

Section 6.3 makes a distinction between reviews and analyses that is pertinent when considering formal methods.

6.3. Software Reviews and Analyses: *Reviews and analyses are applied to the results of the software development processes and software verification process. One distinction between reviews and analyses is that analyses provide repeatable evidence of correctness and reviews provide a qualitative assessment of correctness.* (A draft version of DO-178B said “*The primary distinction between reviews and analyses is that analyses provide repeatable evidence, and reviews provide a group consensus of correctness.*”)

The significant attribute of formal methods is that they allow certain questions to be settled by calculation—that is, by analysis—which informal methods must resolve by means of reviews.

An applicant who offers descriptive formal methods might argue that formal analysis is not his goal and that reviews provide an adequate means of verification

[16]The guidance on software quality assurance found in Section 8 is chiefly concerned with monitoring and recording the processes recommended in the other sections.

for his purpose. This could be acceptable if formal methods are offered only as a supplement to traditional documentation, and if the traditional verification processes are applied to that documentation (though it would then be natural to question the purpose of offering formal methods); it could also be acceptable if formal methods are used for limited purposes, such as describing data structures. I am skeptical, however, of the reliability of reviews when they are applied to substantial formal specifications involving, for example, significant numbers of axioms, or operations specified by complex pre- and post-conditions, or constructions with subtle semantics (e.g., schemas in the language Z). The problem is that it is difficult to provide objective evidence that the authors of a specification can reliably express themselves in such forms, and that its reviewers can interpret them correctly; the problem is compounded by the fact that such specifications often contain technical errors (the equivalent of "coding bugs") that can render them inconsistent or meaningless.

It is my opinion that an essential step in ensuring an effective review process for formal specifications is to require that they are subjected to stringent (and preferably mechanized) analysis *before* they are submitted to reviews. The purpose of the analysis is to eliminate as large a class of potential faults as possible by purely formal means (i.e., by calculational processes), so that the review process may concentrate on the intellectual substance of the specification. Some specific forms of analysis that should be considered (in ascending order of stringency) are

- Parsing.
- Typechecking (there are many degrees of stringency possible here; the most stringent generally require use of theorem proving).
- Well-formedness checking for definitions (i.e., assurance of conservative extension).
- Demonstration of consistency for axiomatic specifications (i.e., exhibition of models).
- Animation (i.e., construction of an executable prototype from the formal specification, so that it can be subjected to experiment). This form of analysis has a rather different character than the others listed here, and should be used for specific purposes that are defined beforehand—otherwise it can degenerate into hacking.
- State exploration (i.e., exploring *all* the behaviors of a possibly downscaled version of the specification).
- Formal challenges (i.e., posing and proving putative theorems that should be entailed by the specification).

My experience is that mechanically supported analyses of the kinds suggested above are extremely potent forms of fault detection for formal specifications. I expect that in many projects it will also be worthwhile to develop additional forms of mechanized analysis to check for specific classes of faults. By these means, it can be ensured that the formal specifications submitted for review are free of gross defects and the reviewers can focus on deeper issues.

The question remains: how much confidence can we have in reviews of formal specifications by personnel who may not be experts in formal methods? It seems to me that we must trust the integrity of the review process to decide this. Currently, reviews are conducted using checklists with items such as "do you consider the requirements are complete?" and it will be necessary to add items such as "do you consider that you have been able to fully comprehend the formal specification?" The assurance that participants do fully comprehend a formal specification may be enhanced if the suggestions of Parnas and Weiss [PW85] are followed: for example, someone other than the author of a specification should be expected to explain it during the review, and the author should pose questions to the reviewers (rather than vice versa).

The rationale submitted to satisfy Section 12.3.b(3) of DO178B, should clearly state the analyses that are required to be completed prior to reviews, and should describe the class of faults that are detected by means of these analyses, and whether the detection is certain, or merely likely. The number and stringency of the analyses performed may be determined by the criticality and sophistication of the formal specifications considered. The rationale should also provide evidence that the applicant's process for reviewing formal specifications is effective.

Although descriptive formal methods have value, it is when formal methods exploit the power of calculational, and especially automated, forms of analysis that their singular potential is best realized. For example, in its Section 6.3.1 (Reviews and Analyses of High-Level Requirements), DO-178B requires

> **b.** (Accuracy and Consistency): *The objective is to ensure that each high-level requirement is accurate, unambiguous and sufficiently detailed and that the requirements do not conflict with each other.* (Similar considerations apply to lower-level requirements described in DO-178B Section 6.3.2.)

Some aspects of consistency and nonconflict can be established mechanically for formal specifications by strong typechecking and related analyses.

At a more detailed level, the verification objectives stated in Section 6.3.1 of DO-178B include

> **g.** (Algorithm Aspects): *The objective is to ensure the accuracy and behavior of the proposed algorithms, especially in the area of discontinuities.*

Suitable verification objectives for analytic formal methods would be to demonstrate that certain fundamental algorithms and architectural mechanisms involving complex behavior (i.e., behavior with large numbers of "*discontinuities*") satisfy their corresponding high-level requirements. Other suitable verification objectives for strong kinds of formal analysis may be to discharge the requirements of DO-178B Section 6.2.d.

> *When it is not possible to verify specific software requirements by exercising the software in a realistic test environment, other means should be provided and their justification for satisfying the software verification process objectives [should be] defined in the Software Verification Plan.*

In my opinion, circumstances "when it is not possible to verify specific software requirements by exercising the software in a realistic test environment" are likely to include those where complex interactions produce very large numbers of possible behaviors (e.g., in coordination of real-time or concurrent processes, and in redundancy management), and analytic formal methods may be the most effective "other means" of verification in these cases. DO-178B seems to agree that formal methods are well suited to such "complex behaviors," and in its Section 12.3.1 states that

> *Formal methods may be applied to software requirements that:*
>
> - *Are safety-related.*
> - *Can be defined by discrete mathematics.*
> - *Involve complex behavior, such as concurrency, distributed processing, redundancy management, and synchronization.*

The specific benefit provided by formal methods is that they allow "*complex behaviors*" to be analyzed (by means of proofs or state exploration), rather than merely reviewed—and analyzed in their totality, rather than merely sampled as by testing or simulation. Thus, the benefit derives from formal analysis, not from formal specification alone: formally specifying the individual state machines at either end of a protocol, for example, adds little to our understanding—we need to calculate their *combined* behavior to ensure that they accomplish the desired goal.

My recommendation is that those aspects of design that "*involve complex behavior*" should be provided with at least the level of formal description and analysis that would be found in a refereed computer science journal. That is, for the mechanisms of, say, redundancy management, a specification of the relevant algorithms should be provided, together with the fault assumptions, the fault masking or recovery objectives, and a proof that the algorithms satisfy the objectives, subject to the assumptions. This level of rigor of presentation is what I earlier called a "Level 1" formal method; it is less rigorous than any of the levels of formality contemplated in Section 12.3.1 of DO-178B. Nonetheless, I believe this level of rigor would be a distinct improvement on current practice.

Beyond this modest step, we should consider the extent to which quality control and assurance might be further enhanced by increasing the level of formal rigor employed, or the number of stages of the lifecycle subjected to formal analysis.

When the concern is to establish that certain tricky or crucial aspects of design are correctly handled by the algorithms and architectural mechanisms employed, I see little advantage to Level 2 formal methods over Level 1: it is the proofs that matter, and both levels employ the traditional kind (presented and checked informally "by hand"); all that Level 2 would add is a fixed syntax for the specification language and possibly some built-in models of computation and concurrency. These last may be a mixed blessing: useful if they match the needs of the problems considered, otherwise an obstacle to be overcome.

But if Level 2 formal methods add very little in this domain, Level 3 may add a great deal. The focus will be on difficult problems, where a large number of potential behaviors must be considered—that is why the applicant has decided to use formal methods—and the proofs may be expected to be replete with boundary conditions and case analyses. These are precisely the kinds of arguments where informal reasoning may be expected to go astray—and go astray it does: for example, the published proof for one synchronization algorithm [LMS85] has flaws in its main theorem and in four of its five lemmas [RvH93]. The flaws in this example were discovered while undertaking formal analysis at Level 3 and suggest the benefits that may be derived from this level of rigor.

The value of undertaking mechanically checked proofs is that the dialog with a theorem prover forces examination of all cases to the argument. On it own, a mechanically checked proof is not a "*means of compliance*" with a certification basis and concern that "the theorem prover has not itself been proved correct" is not an obstacle to deriving great benefit and additional assurance by applying Level 3 formal methods in this domain. The analysis produced through dialog with any adequately validated proof-checker will be considerably more complete and reliable (and repeatable) than one produced without such aid—it is the ultimate walkthrough—but the "certificate" that comes from a mechanized proof checker should not be accepted as unsupported evidence of fitness any more than should other computer-assisted calculations, such as those of aerodynamic properties, or of mechanical stress. In my opinion, the analysis developed with the aid of a theorem prover should also be rendered into a clear and compelling semiformal (i.e., Level 1) argument that is subjected to intense human review, and it is the *combination* of stringent mechanical and human scrutiny (and other evidence, such as tests) that should be considered in certification.[17]

[17]For this reason, I do not endorse the requirement in UK Interim Defence Standard 00-55 [MOD91, paragraph 32.2.3] that a second mechanically checked proof using a "diverse tool"

The construction of a mechanically checked proof that certain algorithms and architectural mechanisms accomplish certain goals subject to certain assumptions addresses only part of the problem: it is also necessary to validate the modeling employed. That is to say, the applicant needs to provide evidence that the model of computation employed, and the statements of the assumptions made and of the goals to be achieved, are all true in the intended interpretation. It is also necessary to provide evidence that the algorithm and architectural mechanisms considered in the proof are correctly implemented. There is a tension between these concerns: it is generally easier to validate models that make a few broad and abstract assertions (e.g., "it is possible for a nonfaulty processor to read the clock of another nonfaulty processor with at most a small error ϵ") than those that make many detailed ones (e.g., that talk about specific mechanisms for reading clocks and the behavior of particular interface registers), but the "gap" between the verified specification and its implementation will be greater in the former case. Since the assurance objective of this analysis is to ensure that there are no conceptual flaws in the basic algorithms and mechanisms, my opinion is that credibility of validation should take precedence over proximity to implementation. This argues for performing the analysis early in the lifecycle and using abstract modeling (i.e., suppressing all detail judged irrelevant). Validation should be accomplished by peer review, supported by analyses that demonstrate, for example, that axiomatic specifications are consistent (i.e., have a model), that intended models are not excluded (e.g., that clocks that keep perfect time satisfy the axioms for a "good clock"), that definitions are well formed, and that expected properties (i.e., "challenges") can be proven to follow from the specification. Concern that implementations are faithful to their verified specifications is a separate problem, and can be handled using either formal methods, or traditional techniques for V&V. My personal opinion is that traditional techniques are likely to be adequate: the evidence seems to be that it is the basic mechanisms and algorithms that have been flawed, not their implementations.

3.3 Conclusion

The recommendations presented above may seem modest to those who believe that formal methods should be used more extensively (for example, in the manner required by UK Interim Defence Standard 00-55). They may also seem a retreat from the traditional goals of formal verification: there would no claims of "proving correctness," and no ambition to apply formal methods from "top to bottom" (i.e., from requirements down to code or gates). Rather, the goal would be to establish that certain properties hold, and certain conceptual faults are absent, in formal models of some of the basic mechanisms necessary to safe operation of the system.

should be required. The resources required would be better expended on diverse *analysis*, and on human scrutiny of the argument and modeling employed.

These may seem small claims in the total scheme of things, but they are the claims that I think are least well supported by current practice and that cause the most concern, since they are the most fundamental.

Those who argue that more should be required—that formal methods should be carried down to code or gates, or that formal specifications should be used as part of the software engineering process—need to provide evidence that this will increase assurance in an industry that has an excellent record of accomplishment using traditional methods. They also need to provide evidence that resources expended on formal methods would not be better spent on other forms of assurance.

On the other hand, these recommendations may seem excessive to some readers: I propose that the most stringent kinds of formal methods should be applied to the hardest problems of design. These pose tough challenges, to be sure, but how could anything less challenging be expected to improve a process that is already very effective? And notice that although these challenges are tough, they are relatively few in number and small in scale, and can therefore be undertaken by a small (though highly skilled) team of people. The tools that are currently available to support these ambitious applications of formal methods are not ideal, but are adequate to the task.

Finally, I would like to observe that using all the techniques at our disposal, even including formal methods, I do not believe we can provide assurance that software of any significant complexity achieves failure rates on the order of 10^{-9} per hour for sustained periods, and we should not build systems that depend on such undemonstrable properties. System-level reliability and safety analyses must not be predicated on software failure rates that cannot be substantiated by experiment or analysis. To achieve a credible probability of catastrophic system failure below 10^{-9}, software must generally be buttressed by mechanisms depending on quite different technologies that provide robust forms of diversity. In the case of flight control for commercial aircraft, this probably means that stout cables should connect the control yoke and rudder pedals to the control surfaces.

Acknowledgments

I wish to acknowledge my deep appreciation for the support and encouragement (and patience!) of Pete Saraceni of the Flight Safety Research Branch of the FAA Technical Center, and Ricky Butler of the NASA Langley Research Center. I would also like to thank my colleagues Judy Crow, Jack Goldberg, Pat Lincoln, Sam Owre, Peter Neumann, Natarajan Shankar, and Mandayam Srivas for numerous discussions on these topics, and for comments on drafts of this report.

References

[BCAG95] *Statistical Summary of Commercial Jet Aircraft Accidents, Worldwide Operations, 1959–1994*. Published annually by: Airplane Safety Engineering (B-210B), Boeing Commercial Airplane Group, Seattle, WA, March 1995.

[BF93] Ricky W. Butler and George B. Finelli. The infeasibility of experimental quantification of life-critical software reliability. *IEEE Transactions on Software Engineering*, 19(1):3–12, January 1993.

[Bid91] Wayne Biddle. *Barons of the Sky: From Early Flight to Strategic Warfare, the Story of the American Aerospace Industry.* Simon and Schuster, New York, NY, 1991. Paperback edition by Henry Holt, 1993.

[BJ94] J. Brazendale and A. R. Jeffs. Out of control: Failures involving control systems. *High Integrity Systems*, 1(1):67–72, 1994.

[Bur93] R. W. Burns. Genius at work. *IEE Review*, 39(5):187–189, September 1993.

[CGH$^+$95] Edmund M. Clarke, Orna Grumberg, Hiromi Haraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, and Linda A. Ness. Verification of the Futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6(2):217–232, March 1995.

[CGL94] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency*, pages 124–175, REX Workshop, Mook, The Netherlands, June 1994. Volume 803 of *Lecture Notes in Computer Science*, Springer-Verlag.

[Coo93] Henry S. F. Cooper Jr. *The Evening Star: Venus Observed.* Farrar Straus Giroux, New York, NY, 1993.

[CRSS94] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In Ramayya Kumar and Thomas Kropf, editors, *Theorem Provers in Circuit Design (TPCD '94)*, pages 203–222, Bad Herrenalb, Germany, September 1994. Volume 910 of *Lecture Notes in Computer Science*, Springer-Verlag.

[DDHY92] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992. Cambridge, MA, October 11-14.

[Dor91] Michael A. Dornheim. X-31 flight tests to explore combat agility to 70 deg. AOA. *Aviation Week and Space Technology*, pages 38–41, March 11, 1991.

[ECK+91] Dave E. Eckhardt, Alper K. Caglayan, John C. Knight, Larry D. Lee, David F. McAllister, Mladen A. Vouk, and John P. J. Kelly. An experimental evaluation of software redundancy as a strategy for improving reliability. *IEEE Transactions on Software Engineering*, 17(7):692–702, July 1991.

[FAA88] *System Design and Analysis.* Federal Aviation Administration, June 21, 1988. Advisory Circular 25.1309-1A.

[FAA89] *Digital Systems Validation Handbook–Volume II.* Federal Aviation Administration Technical Center, Atlantic City, NJ, February 1989. DOT/FAA/CT-88/10.

[FB92] John H. Fielder and Douglas Birsch, editors. *The DC-10 Case: A Case Study in Applied Ethics, Technology, and Society.* State University of New York Press, 1992.

[GH90] G. Guiho and C. Hennebert. SACEM software validation. In *12th International Conference on Software Engineering*, pages 186–191, Nice, France, March 1990. IEEE Computer Society.

[GY76] S. L. Gerhart and L. Yelowitz. Observations of fallibility in modern programming methodologies. *IEEE Transactions on Software Engineering*, SE-2(3):195–207, September 1976.

[Hec93] Herbert Hecht. Rare conditions: An important cause of failures. In *COMPASS '93 (Proceedings of the Eighth Annual Conference on Computer Assurance)*, pages 81–85, Gaithersburg, MD, June 1993. IEEE Washington Section.

[HJ89] I. J. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *IEE/BCS Software Engineering Journal*, 4(6):320–338, November 1989.

[HK90] Zvi Har'El and Robert P. Kurshan. Software for analytical development of communications protocols. *AT&T Technical Journal*, 69(1):45–59, January/February 1990.

[Jon90] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, second edition, 1990.

[Joy89] Jeffrey Joyce. *Multi-Level Verification of Microprocessor-Based Systems*. PhD thesis, University of Cambridge, December 1989.

[Keu91] Kurt Keutzer. The need for formal verification in hardware design and what formal verification has not done for me lately. In Phillip Windley, editor, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, pages 77–86, Davis, CA, August 1991. IEEE Computer Society.

[KL86] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, January 1986.

[KP93] Rick Kasuda and Donna Sexton Packard. Spacecraft fault tolerance: The Magellan experience. In Robert D. Culp and George Bickley, editors, *Proceedings of the Annual Rocky Mountain Guidance and Control Conference*, pages 249–267, Keystone, CO, February 1993. Volume 81 of *Advances in the Astronautical Sciences*, American Astronautical Society.

[KSH92] John C. Kelly, Joseph S. Sherif, and Jonathan Hops. An analysis of defect densities found during software inspections. *Journal of Systems Software*, 17:111–117, 1992.

[Lev95] Nancy G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[LM94] Leslie Lamport and Stephan Merz. Specifying and verifying fault-tolerant systems. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 41–76, Lübeck, Germany, September 1994. Volume 863 of *Lecture Notes in Computer Science*, Springer-Verlag.

[LMS85] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[LR93] Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Fault Tolerant Computing Symposium 23*, pages 402–411, Toulouse, France, June 1993. IEEE Computer Society.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[LT82] E. Lloyd and W. Tye. *Systematic Safety: Safety Assessment of Aircraft Systems.* Civil Aviation Authority, London, England, 1982. Reprinted 1992.

[LT93] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[Lut93] Robyn R. Lutz. Analyzing software requirements errors in safety-critical embedded systems. In *IEEE International Symposium on Requirements Engineering*, pages 126–133, San Diego, CA, January 1993.

[Mac88] 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.

[Mel94] Peter Mellor. CAD: Computer-aided disaster. *High Integrity Systems*, 1(2):101–156, 1994.

[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.

[MS95] Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 2–16, Boca Raton, FL, 1995. IEEE Computer Society.

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[PW85] David L. Parnas and David M. Weiss. Active design reviews: Principles and practices. In *8th International Conference on Software Engineering*, pages 132–136, London, UK, August 1985. IEEE Computer Society.

[RBP$^+$91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[RTCA92] *DO-178B: Software Considerations in Airborne Systems and Equipment Certification.* Requirements and Technical Concepts for Aviation, Washington, DC, December 1992. This document is known as EUROCAE ED-12B in Europe.

[Rus93] John Rushby. Formal methods and the certification of critical systems. Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Also issued under the title *Formal Methods and Digital Systems Validation for Airborne Systems* as NASA Contractor Report 4551, December 1993.

[RvH93] John Rushby and Friedrich von Henke. Formal verification of algorithms for critical systems. *IEEE Transactions on Software Engineering*, 19(1):13–23, January 1993.

[Spi93] 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.

[Vin90] Walter G. Vincenti. *What Engineers Know and How They Know It: Analytical Studies from Aeronatical History.* Johns Hopkins Studies in the History of Technology. The Johns Hopkins University Press, Baltimore, MD, 1990.

Index