Formal Methods and the Certification of Critical Systems[1]

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-93-7
December 1993

[1]Also issued under the title *Formal Methods and Digital Systems Validation for Airborne Systems* as NASA CR 4551. 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 was prepared to supplement a forthcoming chapter on formal methods in the FAA Digital Systems Validation Handbook[1]. Its purpose is to outline the technical basis for formal methods in computer science, 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 report assumes a serious interest in the engineering of critical systems, and a willingness to read occasional mathematical formulas and specialized terminology, but assumes no special background in formal logic or mathematical specification techniques. (An appendix provides a rapid introduction to formal logic for those to whom this topic is new.) The discussion should be accessible to most people with an engineering background. It may also be of use to those who develop or advocate formal methods and are interested in their use to support certification of critical systems.

The particular focus of this report is computer systems used on-board aircraft. Some background on these systems is provided so that those concerned with critical computer systems in other contexts may be able to reinterpret the airplane-specific material to suit their own field.

---

[1] *Digital Systems Validation Handbook–Volume II.* Federal Aviation Administration Technical Center, Atlantic City, NJ, February 1989. DOT/FAA/CT-88/10. The chapter on Formal Methods will probably form part of Volume III.

Contents

How to Read this Report

This report was written primarily for FAA certifiers and assumes no previous exposure to formal methods. However, it is not a tutorial on formal methods: it contains a great deal of discussion on strengths, weaknesses, and technical issues in formal methods that should be considered in certification, but does not explain how to *do* formal methods. There are a few simple examples scattered through the report, but these are intended only to give some feel for the topics discussed, and are not representative of specific techniques or notations that might be offered in support of certification. Those who wish to offer formal methods in support of certification must obviously become expert in the methods they select, and certifiers will likewise need to gain expertise in the specific methods adopted. Some reading suggestions for those embarking on such studies are provided in Section 2.8.

It is difficult to discuss some of the issues in formal methods without occasionally using technical terms and concepts that require a certain knowledge of the field, and of mathematical logic in particular. I have tried to keep the discussion suitable for the nonspecialist, but have not excluded technical terms and concepts where these are necessary for precision. An appendix provides some technical background in a relatively brief and accessible form, explaining the main ideas of mathematical logic, and introducing most of the technical terms that a nonspecialist might encounter.

The organization of this report is as follows. Chapter 1 provides an introduction and adumbrates some of the main themes. These are covered in detail in Chapter 2, which adopts a fairly general perspective. Specialization of the perspective towards digital systems validation for aircraft, and discussion of issues bearing on certification credit and compliance with the guidelines in DO-178B [RTCA92], make up Chapter 3.

A reader new to formal methods should probably concentrate on Chapters 1 and 3, and should only skim Chapter 2 the first time through, perhaps taking a little more time over its later sections, where the benefits, fallibilities, and some applications of formal methods are described. In order to evaluate the credibility of any claims involving formal methods that may be offered in support of certification, rather greater familiarity with the topics examined in the earlier sections of Chapter 2 will be required. In particular, the certifier and applicant will need to reach agreement on the technical basis for validating formal specifications.

Although this report is written primarily for certifiers, I hope it will also be useful to those involved in developing airborne systems, and I have included some information (for example, on the selection of methods and tools) that is specifically addressed to that audience. In order to make the discussion accessible to those concerned with applications other than on-board aircraft, and also in the hope that developers and advocates of formal methods may wish to contribute to their application in such systems, I include descriptions of some of the characteristics of airborne systems.[2]

A Note on Terminology

Although the topic of this report is formal methods, the context for its discussion is the certification of embedded software for systems deployed in civil aircraft. This necessarily involves some discussion of safety-critical systems and of fault tolerance. Both these fields employ very careful terminology in order to distinguish concepts that are important to their analyses. Unfortunately, each field is rather careless about the terminology of the other. I will strive to use terms in the way they are used in the specialized field concerned. In this section I briefly define and explain these usages.

IFIP Working Group 10.4 (Dependability and Fault Tolerance) has established a standard framework for discussing reliable and fault-tolerant systems. A book presents the basic concepts and preferred terminology in five languages [Lap91]. In this framework, a *dependable system* is one for which reliance may justifiably be placed on certain aspects of the quality of service it delivers. The quality of a service includes both its correctness (i.e., conformity with requirements, specifications, and expectations) and the continuity of its delivery.

A departure from the service required of a system constitutes a *failure*—so that a dependable system can also be described as one that does not fail. Failures are attributed to underlying causes called *faults*. Faults can include mistakes in specifications or design (i.e., bugs), component failures, improper operation, and environmental anomalies (e.g., electromagnetic perturbations). Not all faults produce immediate failures: failure is a property of the external behavior of a system—which is itself a manifestation of internal states and state transitions. Suppose a system

[2]For more background, consult Anderson and Dorfman [AD91], who provide a good collection of papers on software engineering for aerospace, and Spitzer [Spi87], who gives a general introduction to digital avionics. Current developments can be followed through the Digital Systems Avionics Conference (DASC) and National Aerospace Electronics Conference (NAECON), as well other specialized conferences and journals of the AIAA, IEEE, and SAE. For an excellent account of the basic approach to safety assessment of aircraft, see the book by Lloyd and Tye [LT82] (though unfortunately this predates the introduction of fly-by-wire). For a discussion of the certification challenges posed by new technology, including fly-by-wire, see the paper by Holt [Hol87]; for an assessment of how current certification practices address these challenges, see the report by the United States General Accounting Office [GAO93]. For a critical examination of ethical and regulatory issues in safety-critical systems, see the compendium edited by Fielder and Birsch [FB92]. The document DO-178B [RTCA92] (known as EUROCAE ED-12B in Europe), which is referred to frequently in the following pages, provides industry-accepted guidelines for meeting certification requirements for software used in airborne systems and equipment. It is incorporated by reference into European and United States regulatory and advisory documents.

progresses through a sequence of internal states $s_1, s_2, \ldots, s_n$ and that its external behavior conforms to its service specification throughout the transitions from s_1 to s_{n-1} but that on entering state s_n the external behavior departs from that required. Is it reasonable to attribute the failure to state s_n? Clearly not, since there could have been something wrong with an earlier state s_j, say, that, while it did not produce an immediate failure, led to the sequence of transitions culminating in the failure at s_n. In this case, the state s_j is said to contain a *latent error* that persists through the states that follow and becomes *effective* in state s_n when it affects the service delivered and failure occurs. In general, an *error* is that part of the system state that has been damaged by the fault and (if uncorrected) can lead to failure. Fault tolerance is based on detecting latent errors before they become effective, and then replacing the erroneous component of the state by an error-free version. (Latent errors are discussed in a separate chapter of the FAA Handbook [FAA-L].)

The crucial distinctions here are those between fault, error, and failure. Unfortunately, as noted, other fields do not always preserve these important distinctions; in particular, they often use fault and error interchangeably. DO-178B [RTCA92] is guilty of this, generally using the term *error* to describe programming mistakes or bugs that are properly called *faults*. In this report, I use the proper term, except where quoting from DO-178B.

Safety-critical systems in general, and aircraft systems in particular, make use of the terminology and techniques of hazard analysis. The following paragraphs introduce this terminology for the benefit of those who are reading this report for general interest and who are not familiar with aircraft certification. Weapons systems provided some of the early impetus to safety engineering and hazard analysis, so military standards such as 882B [DoD84] (United States) and 00-56 [MOD91b] (United Kingdom) are good alternative sources on these topics. A standard text is that by Roland and Moriarty [RM90].

While fault tolerance is concerned with reducing the incidence of failures, safety concerns the occurrence of accidents (or *mishaps*)—which are unplanned events that result in death, injury, illness, damage to or loss of property, or environmental harm. Whereas system failures are defined in terms of system services, safety is defined in terms of external consequences.

The general approach to development and certification of safety-critical systems is grounded in hazard analysis; a *hazard* is a condition that can lead to an accident. *Damage* is a measure of the loss in an accident. The *severity* of a hazard is an assessment of the worst possible damage that could result, while the *danger* is the probability of the hazard leading to an accident. *Risk* is the combination of hazard severity and danger. The goal in safety engineering is to control hazards. During requirements and design reviews, potential hazards are identified and analyzed for risk. Unacceptable risks are eliminated or reduced by respecification of requirements, redesign, incorporation of safety features, or incorporation of warning devices.

For example, if the concern is destruction by fire, the primary hazards are availability of combustible material, an ignition source, and a supply of oxygen. If at all possible, the preferred treatments are to eliminate or reduce these hazards by, for example, substitution of non-flammable materials, elimination of spark-generating electrical machinery, and reduction in oxygen content (cf. substitution of air for pure oxygen during ground operations for Project Apollo after the Apollo 1 fire). If hazard elimination is impossible or judged only partially effective, then addition of a fire suppression system and of warning devices may be considered. The effectiveness and reliability of these systems then becomes a safety issue, and new hazards may need to be considered (e.g., inadvertent activation of the fire suppression system).

Software *criticality* is determined by hazard analysis; DO-178B considers five levels from A (most critical) to E. Level A is "software whose anomalous behavior...would cause or contribute to a failure of system function resulting in a catastrophic failure condition for the aircraft" [RTCA92, Subsection 2.2.2]. Catastrophic failure conditions are "those which would prevent continued safe flight and landing" [FAA88, paragraph 6.h(3)] and include both loss of function, malfunction, and unintended function. Failure condition severities and probabilities must have an inverse relationship; in particular, catastrophic failure conditions must be "extremely improbable" [FAA88, paragraph 7.d]. That is, they must be "so unlikely that they are not anticipated to occur during the entire operational life of all airplanes of one type" [FAA88, paragraph 9.e(3)]. "When using quantitative analyses...numerical probabilities...on the order of 10^{-9} per flight-hour[3] may be used...as aids to engineering judgment...to...help determine compliance" with the requirement for extremely improbable failure conditions [FAA88, paragraph 10.b]. An explanation for this figure can be derived [LT82, page 37] by considering a fleet of 100 aircraft, each flying 3,000 hours per year over a lifetime of 33 years (thereby accumulating about 10^7 flight-hours). If hazard analysis reveals ten potentially catastrophic failure conditions in each of ten systems, then the "budget" for each is about 10^{-9} if such a condition is not expected to occur in the lifetime of the fleet. An alternative justification is obtained by projecting the historical trend of reliability achieved in modern jets. From 1960 to 1980, the fatal accident rate for large jets improved from 2 to 0.5 per 10^6 hours, and was projected to be below 0.3 per 10^6 hours by 1990 [LT82, page 28].[4] This suggests that less than 1 fatal accident per

[3] "Based on a flight of mean duration for the airplane type. However, for a function which is used only during a specific flight operation; e.g., takeoff, landing etc., the acceptable probability should be based on, and expressed in terms of, the flight operation's actual duration" [FAA88, paragraph 10.b].

[4] Between 1982 and 1991, there were 163 "hull loss" accidents. Causes of these accidents have been officially identified in 120 cases; of these, 15 were attributed to failure of the aircraft's design or systems [BCAG94]. Since the 1980s there has been a large increase in on-board safety-critical software in systems such as engine control and monitoring, displays, autopilot, and flight management. Among aircraft of this generation manufactured in the United States, the Boeing 757,

10^7 hours is a feasible goal, and the same calculation as above then leads to 10^{-9} as the requirement for individual catastrophic failure conditions.

Note that the probability 10^{-9} is applied to (sub)system failure, not to any software the system may contain. Numerical estimates of reliability are not assigned to software in safety-critical systems [RTCA92, Subsection 2.2.3], primarily because software failure is not random but systematic (i.e, due to faults of specification, design, or construction), and because the rates required are too small to be measured; discussion of these points consumes much of the early part of Chapter 3.

Acknowledgements

Many people have contributed to my understanding of the topics discussed in this report. In addition to my obvious debt to the founders, educators, and practioners of the various disciplines that are covered here, I am grateful to those who have taken time over the last several years to explain and argue their points of view with me. In formal methods these particularly include my colleagues Friedrich von Henke, Sam Owre, and Shankar; and in safety-critical systems and their assessment, Ricky Butler of NASA Langley Research Center, Nancy Leveson of the University of Washington, Bev Littlewood of the City University in London, and Brian Randell of the University of Newcastle upon Tyne.

John Kelly and his colleagues at the Jet Propulsion Laboratory helped me with data on faults found during inspection and test, and David Hamilton of IBM Federal Systems Company (Houston) and Steve Miller of Collins Commercial Avionics helped me understand the requirements definition and assurance processes on large aerospace projects.

I am grateful for comments received on earlier drafts from Ricky Butler, Ben DiVito, Michael Holloway, and Sally Johnson of NASA Langley Research Center, and for the support of Pete Saraceni of the Flight Safety Research Branch of the FAA Technical Center.

747-400, and the McDonnell Douglas MD-11 have had no hull loss accidents. (However, inadequate design of the wing flap/slat control handle of the MD-11 has led to 12 unintentional activations of the flap/slat system in flight, including the China Eastern Airlines accident of April 1993 which injured 160 passengers and killed two [Phi93].) The Boeing 767 has had one: the Lauda Air crash over Thailand in May 1991 that killed 223 and was caused by in-flight activation of an engine thrust reverser.

Only one airplane in current service—the Airbus A320—uses full fly-by-wire. It has had four fatal, hull loss, crashes since entering service. This appalling record is far worse than any other contemporary airplane on an hours-flown basis, but none of these accidents has implicated the reliability of its software. The first three crashes were officially attributed to pilot error; the cause of the fourth (the 14 September 1993 Lufthansa accident on landing at Warsaw-Okocie airport) has not yet been officially determined, though informed sources indicate that in this case pilot error will not be implicated.

This version of the report contains some changes to that issued as NASA CR 4551. The appendix, in particular, incorporates several improvements suggested by Bob Riemenschneider, who refereed it for the CSL Technical Report series.

Chapter 1

Introduction

> *"When software fails, it invariably fails catastrophically. This awful quality is a reflection of the lack of continuity between successive system states in executing software. If the preceding state is correct, there is no inherent carry over of even partial correctness into the next succeeding state."* [Roy91, page 11]

> *"Primarily, mathematics is a method of inquiry known as postulational thinking. The method consists in carefully formulating definitions of the concepts to be discussed and in explicitly stating the assumptions that shall be the basis for reasoning. From these definitions and assumptions conclusions are deduced by the application of the most rigorous logic man is capable of using."* [Kli90, page 20]

"Formal methods" are the use of mathematical techniques in the design and analysis of computer hardware and software; in particular, formal methods allow properties of a computer system to be predicted from a mathematical model of the system by a process akin to calculation. Engineers in established disciplines are fully familiar with the benefits that mathematical modeling and analysis can provide: for example, computational fluid dynamics (CFD) allows safe and relatively inexpensive exploration of alternative airfoil designs, and accurate prediction of the behavior of the selected design. Wind-tunnel and flight tests are performed primarily to validate the accuracy of the CFD model, and to measure final performance parameters. With computer software, however, prototyping and testing remain the principal methods for exploring designs and validating implementations. These are expensive and possibly dangerous, and can provide only partial coverage of the range of behaviors that a piece of software may exhibit.

The problem, of course, is that computer hardware and software are discrete systems: their behavior is determined by a succession of discrete state changes. The succession of states need not produce behavior that varies smoothly or continuously; instead it can exhibit discontinuities and abrupt transitions. The ability to select among many alternative courses of action is a source of the power and flexibility provided by computer systems, but it is also the reason why their behavior is hard to predict and to validate. Tests provide information on only the state sequences actually examined; without continuity there is little reason to suppose the behavior of untested sequences will be "close" to tested ones, and therefore little justification for extrapolating from tested cases to untested ones.

Formal methods confront the discrete behavior of computer systems by using *discrete mathematics* to model it. Discrete mathematics builds directly on mathematical logic, and proofs of theorems take the place of the numerical calculations that are familiar in most other engineering mathematics. That is, instead of using a mathematical model to calculate a value for some numerical quantity such as lift or drag, in formal methods for computer science we prove theorems about the modeled behavior—such as "for any integer input x in the range $0 \le x < 2^{16}$, this program produces an integer output y satisfying $y^2 \le x < (y+1)^2$." In formal methods, the problem of discontinuity and the unsoundness of extrapolating from a finite number of tests are overcome using methods of proof based on mathematical induction, so that an infinite (or finite but very large) number of possible behaviors is fully covered in a finite proof.[1]

Formal methods offer much more to computer science than just "proofs of correctness" for programs and digital circuits, however. Many of the problems in software and hardware design are due to imprecision, ambiguity, incompleteness, misunderstanding, and just plain mistakes in the statement of top-level requirements, in the description of intermediate designs, or in the specifications of components and interfaces. Some of these problems can be attributed to the difficulty of describing large and complex artifacts in natural language. Many notations and techniques have been proposed to overcome this difficulty and it is important here to distinguish formal methods from what we might call *formalized* methods. The latter include many CASE methodologies, diagrammatic techniques, pseudocode, and other systematic ways for describing the requirements, specification, or operation of computer systems. The distinction is that formal methods are provided with an explicit method for deriving, by a systematic process that resembles calculation, the properties and consequences of a specification: it must be possible, at least in principle, to *calculate* whether one specification correctly implements another, or whether certain requirements are satisfied by a given specification. Formalized methods, on the other hand, continue to rely on intuitive understanding of the notations and concepts employed: they may replace a possibly woolly natural language

[1]Formal methods based on "state-exploration" can sometimes examine all the behaviors in a very large, but still finite, space of possibilities using cleverly-implemented brute force techniques.

description with, say, an apparently precise diagram—but the precision is illusory if there is no underlying semantics giving a strict meaning to the diagram. Formalized methods also often force a premature commitment to design decisions; for example, pseudocode, dataflow diagrams, and many other CASE techniques allow a statement of what is required to be given only in terms of a mechanism to achieve it. This is not to deny the utility and merit of formalized methodologies; the theoretical advantages of truly formal methods may be unimportant in some contexts, and unrealizable (because of the skill and costs that may be required) in others.

Formal methods provide for the construction of specifications whose interpretation is less reliant on human intuition by using techniques based, mainly, on the axiomatic method of mathematics. The basic idea of the axiomatic method is to specify properties required or assumed[2] as axioms in some formal language. These axioms, plus the rules of inference associated with the chosen language, supply everything that can be used in reasoning about the specified artifact. To show that the specification has some property not explicitly mentioned in the axioms, we must prove that that property is a logical consequence of the axioms; similarly, to show that a design meets its requirements, we must prove that each requirement is derivable from the axioms that specify the design. Formal specifications are tested and explored by posing and proving putative theorems that I call *challenges*: "if this specification says what it should, then the following ought to follow."

The word "formal" in formal methods derives from formal logic and means "to do with form." The idea in formal logic is to avoid reliance on human intuition and judgment in evaluating arguments by requiring that all assumptions and all reasoning steps be made explicit, and further requiring that each reasoning step be an instance of a very small number of allowed rules of inference. Assumptions, theorems, and proofs are written in a restricted language with very precise rules about what constitutes an acceptable statement or a valid proof. In their pure, mathematical form, these languages and their associated rules of manipulation are called *logics*. In formal methods for computer science, the languages are enriched with some of the ideas from programming languages and are called *specification languages*, but their underlying interpretation is usually based on a standard logic.

In a formal specification, everything that can be assumed about the specified entity must follow, by the rules of inference for the language concerned, from the

[2]Whether the specification is of requirements or assumptions depends on whether one is specifying an artifact to be constructed, or some external entity (such as a device, or the environment) with which it is to interact. Usually both are required simultaneously, as in formalizations of statements such as "this design is required to satisfy the following requirements, assuming its environment behaves in the manner described..." The formal version of this statement is usually expressed as

$$\mathit{assumptions} \supset (\mathit{design} \supset \mathit{requirements})$$

where $\supset$ is the symbol for implication.

axioms that make up its specification. Hilbert, who established the modern axiomatic method with his treatment of geometry at the turn of the century, asserted that one had to be able to say "tables, chairs, and beer mugs" in place of "points, lines, and planes" [Rei86, page 60]. By this, he meant to stress that intuitions associated with familiar names should be ignored; the terms used must be treated as marks on paper with no properties assumed but those written down as axioms or deducible, via formal proof, as theorems. The benefit claimed for this approach in computer science is that it forces us to make things explicit: instead of relying on intuition about the meaning of English terms (which can be unreliable and can differ from person to person), everything is expressed in a precise form that has an unambiguous interpretation. In addition, all assumptions are written down and placed in the open, where they can be subjected to independent scrutiny, and all arguments are reduced to calculations that can be checked independently (or even mechanically).

In a formal proof, everything we need to know about the particular problem area is encoded in the statement of the theorem and the axiomatization of its premises, and the truth of the theorem can be established by simply "pushing the symbols around" according to the rules of inference; the *form* of the proof determines its validity—it does not depend on unrecorded knowledge of the problem area, nor on intuition about what the theorem says: checking a formal proof is a purely mechanical exercise that can, in principle, be done by a computer program, and with complete accuracy.

This is not to say that knowledge and intuition are superfluous in formal methods—on the contrary, they are essential to the invention of useful axiomatizations and theorems, to their interpretation and application in the real world, and to the *discovery* of proofs. Use of purely formal reasoning to *check* proofs, however, can be beneficial because it forces a complete enumeration of assumptions and inferences, and can catch faulty reasoning that would otherwise be overlooked. Even without mechanical checking, the "symbol-pushing" character of formal proofs can help prevent faults due to flawed intuition, jumping to conclusions, or simple carelessness. Above all, it is a *repeatable* exercise: others can check our work in greater detail and with greater reliability than is feasible for informal reviews.

Some of these ideas are familiar from use of mathematics in traditional engineering disciplines. Intuition about the behavior of fluids is necessary in order to derive the Navier-Stokes equations, and understanding of the application is needed in order to interpret and apply their solution in a particular instance. But in order to manipulate the equations and develop numerical solutions, it is the rules for treating partial differential equations that are important, not the particular interpretation of the equations as the flow of air over a wing.

Partial differential equations and most of conventional mathematics are not fully formal systems, so the rules for their manipulation are not spelled out as explicit axioms and rules of inference, although, in principle, they could be. In just the same way, formal methods can be applied to computer systems in varying degrees of formality. At their most formal, specifications are written in a specification language with a very direct interpretation in logic, and the proofs of theorems are performed, or checked, by computer. Less rigorously formal are methods that use a true specification language with a strict mathematical interpretation, but with limited mechanical support: perhaps a parser and typechecker,[3] and either no, or only rudimentary, theorem-proving capability. With less commitment to full mechanization, these approaches can provide richer specification languages, but proofs and arguments are expected to be conducted mostly with pencil and paper in the style of a normal "mathematical presentation." The least formal of formal methods are those that employ the concepts and notations of discrete mathematics to develop specifications and proofs in the style of traditional mathematics and engineering. The only tools used are pencil, paper, and a wastebasket, but this limitation also confers a freedom to invent notations at will.

"More formal" is not necessarily better, and a machine-checked proof is not necessarily superior to an intuitively compelling sketch. Use of the term "proof" in formal methods must generally be treated with great care, since in common parlance it carries the connotation of complete certainty. But in formal methods and logic, "proof" is a technical term, that describes a certain type of symbolic manipulation. It can be helpful to mentally substitute the term "logical calculation," with deliberate analogy to "numerical calculation," whenever "proof" is used in the context of formal methods. Just like numerical calculations of solutions to partial differential equations, the logical calculations that constitute a formal proof can fail us in several ways: we can make a mistake in the calculation, we can use a specification (or system of equations) that does not accurately model the real world, and our requirements (or interpretation of the answer) may be mistaken. Depending on the problem and the methods used, different levels of formality can assist or hinder our attempts to minimize these potential failings; sometimes formalism may be altogether inappropriate. Responsible use of formal methods and of formal proofs requires an understanding of these limitations, so that the methods are used in a balanced way to improve and augment existing processes for the development of safety-critical systems.[4]

The various degrees of formal methods may be related to the "nonformal" methods used in traditional software and hardware engineering. Software and hardware

[3] A typechecker performs static semantic analysis (rather like the front-end of the compiler for a modern programming language) with special emphasis on the compatibility of the "types" of the constants, variables, and functions involved. For example, a typechecker will detect errors such as adding an integer to a string. Type systems for specification languages can be much richer than those for programming languages, allowing greater precision of expression and stricter checking.

[4] MacKenzie's very readable account of the British VIPER controversy is salutary [Mac91]; see also Section 2.5.2.

engineering generally follow a very formal development *process*, in which requirements are documented and elaborated (possibly through several levels), and designs are similarly elaborated through several levels of specification down to the final code or circuit. Requirements and specifications are usually documented in natural language, possibly augmented by diagrams, equations, flowcharts, data dictionaries, and pseudocode. For safety-critical and many other kinds of systems, the development process includes components known as "verification and validation" (V&V).

Verification is the process of determining whether each level of specification, and the final code itself, fully and exclusively implements the requirements of its superior specification. Formal methods do not replace this process, but rather augment it by providing formal notations and concepts to be used in writing requirements and specifications, and varying degrees of formal proof that can be used in verification.

Validation, the other component of V&V, is the process by which delivered code is directly shown to satisfy the original user requirements by testing it in execution.[5] Modern development processes recognize the value of doing extensive validation earlier in the lifecycle—so that faults can be caught and corrected sooner. Rapid prototyping, simulation, and animation are all techniques that help validate satisfaction of the customer's expectations before the full system is built. Formal methods can also augment this process, by allowing the properties and consequences of nonexecutable specifications to be explored via theorem proving at the earliest stages of the lifecycle. Hybrid methods are also possible: some formal specifications can be directly executed, or converted into rapid prototypes; alternatively, theorem proving can sometimes be replaced (or, rather, achieved by) completely automatic methods based on "state-exploration" techniques. Whereas a rapid prototype can be used to probe selected test cases, state exploration covers all possible behaviors (though often of only a simplified model of the system concerned).

There are advantages, difficulties, and costs associated with the use of formal methods. The balance of advantage varies with the nature and criticality of the application, with the stages of the lifecycle in which formal methods are used, with the degree of formality employed, with the quality of the method and of any mechanized tools that support it, and with the experience and skill of the practitioners. The general advantages claimed for all formal methods are that they enable faults to be detected *earlier* than would otherwise be the case. This is because they allow greater precision and explicitness to be achieved earlier in the lifecycle than

[5] DO-178B defines validation as "the process of determining that the requirements are the correct requirements and that they are complete,"and verification as "the evaluation of the results of a process to ensure correctness and consistency with respect to the inputs and standard(s) provided to that process" [RTCA92, Annex B]. FAA staff informally characterize validation as "showing that you got the requirements right" and verification as "showing that you built the system according to the requirements." This is similar to the well-known informal characterization that verification is concerned with showing that the product is built right, while validation is concerned with showing that the right product has been built.

otherwise, and because they can support more systematic analyses than informal methods. Stronger forms of this claim are that formal methods cause faults to be detected with greater *certainty* than would otherwise be the case, and that in certain circumstances, subject to certain caveats, they can *guarantee* the absence of certain faults. These stronger claims are generally associated with formal methods that undertake "rigorous" or mechanically-checked proofs.

The advantages claimed for formal specifications are that they reduce the ambiguity and imprecision of natural-language specifications, that explicit use of mathematical concepts (such as sets, mappings, and predicates) contributes to clarity of thought and expression, and that use of quantification, higher-order constructs, and other concepts from logic allow clear statement of *what* is to be done, without imposing a premature commitment (as, for example, pseudocode does) to *how* it is to be done. The advantages claimed for formal proof are that it is a potent method for revealing faults, that it covers all cases, and that the reasoning can be checked by others (or by machine in fully formal cases). Arguments in favor of the more completely formal methods are that mechanical checking eliminates faults that can survive less rigorous scrutiny, that the intensity of the scrutiny leads to improved understanding and to better *informal* presentations, and that complete formality forces an explicit enumeration of all assumptions. Arguments against formal methods (these are particularly leveled at the more intensely rigorous interpretations) are that it is expensive, that activity tends to be focused on the formalism and not on the real problem at hand, that no guarantees are possible unless it is performed from "top to bottom" (which is generally infeasible), and that formal verification is ultimately useless, since the behavior of the finished product will depend on real-world physical processes (e.g., the behavior of a computer or of an actuator) whose behavior may not be modeled with complete accuracy, and is judged against real-world expectations that may not be captured accurately or completely in formal requirements statements.

In the following chapters I examine these and related issues in more detail and provide certifiers with information that I hope will assist them in evaluating the potential contributions of formal methods to digital systems used on aircraft, and to make decisions on certification credit for use of formal methods.

Chapter 2

Issues in Formal Methods

> *"Deduction, as a method of obtaining conclusions, has many advantages over trial and error or reasoning by induction and analogy. The outstanding advantage is ... that the conclusions are unquestionable if the premises are. Truth, if it can be obtained at all, must come from certainties and not from doubtful or approximate inferences. Second, in contrast to experimentation, deduction can be carried on without the use or loss of expensive equipment. Before the bridge is built and before the long-range gun is fired, deductive reasoning can be applied to decide the outcome.*
>
> *"With all its advantages, deduction does not supersede experience, induction, or reasoning by analogy. It is true that 100 per cent certainty can be attached to the conclusions of deduction when the premises can be vouched for 100 per cent. But such unquestionable premises are not necessarily available."* [Kli90, page 45]

As mentioned in the introduction, formal methods can be applied in varying degrees of rigor. There are also other axes along which formal methods can vary in the thoroughness of their application: they can be applied to selected, rather than to all, stages of the lifecycle, and to some or all of the components and properties of the system. I begin this chapter with a classification of formal methods according to the rigor of their application, and then consider issues in the extent and thoroughness of their application through the lifecycle, across components, and with respect to different system properties. Later, I examine techniques for validating formal specifications, and then consider benefits that may accrue from use of formal methods, the fallibilities of those methods, and tools to support them. I close the chapter with brief descriptions of some applications of formal methods in industrial settings.

2.1 Levels of Rigor in Formal Methods

In the introduction, I suggested that varying levels of rigor can be discerned in the application of formal methods. Here, I systematize that classification, and outline issues in the choice of level for a particular application. The word "rigor" is not a completely accurate term for the characterization that I have in mind, but no other term seems better. What I am concerned with is the extent to which a method is "truly formal": that is, formulates specifications in an axiomatic style, explicitly enumerates all assumptions and reduces proofs to explicit applications of elementary rules of inference. Increasing formality allows examination of the products of formal methods (i.e., specifications and proofs) to be less dependent on *reviews* (i.e., processes that depend on consensus judgment) and more amenable to *analyses* (i.e., systematic forms of examination whose results are *repeatable*). The distinction between reviews and analyses is from DO-178B [RTCA92, Section 6.3].

Because it requires almost superhuman discipline to be truly formal with pencil and paper, increasing formality is usually associated with increasing dependence on mechanical support (conversely, since computers can do nothing but "push symbols around," it is difficult to mechanize the reasoning processes of less than truly formal methods). Note that it is possible to be "rigorous," in the sense of being painstakingly careful and serious, without being truly formal.

I classify formal methods into the following four levels of increasing rigor.

Level 0: *No use of formal methods.* This corresponds to current standard practice in which verification is a manual process of review and inspection applied to documents written in natural language, pseudocode, or a programming language, possibly augmented with diagrams (and equations in the case of control laws). Validation is based on testing, which can be driven by requirements and specifications (i.e., functional or black-box testing) or by the structure of the program.

Although the standard practice makes little use of formal methods, it may employ very thorough and highly formalized development *processes*. For example, "structured walk-throughs" and "formal inspections" are highly structured methods for the manual review of program designs and code. As first described by Fagan [Fag76] (see [ABL89] and [Fag86] for more recent treatments, and [Bus90, Wel93] for case studies), four participants are required—the *Moderator*, the *Designer*, the *Coder/Implementor/Specifier*, and the *Tester*. If a single person performs more than one role in the development of the system, substitutes from related projects are impressed into the review team. The review team scrutinizes the specification, design or program in considerable detail: typically, one person (usually the Coder) acts as a "reader" and describes the specification or the workings of the design or program to the others, "walking through" it in a systematic manner so that every piece of logic is covered at least once, and every or branch (in the case of programs) is taken at least once. Intensive questioning is encouraged, and it is the Moderator's responsibility to ensure that it remains constructive and purposeful (managers are excluded and the process is not used to evaluate the performance of the participants). As the design and its implementation become understood, the attention shifts to a conscious search for faults. A checklist of likely mistakes may be used to guide the fault-finding process. Considerable effectiveness is reported for formal inspections: removal of 50% of design and implementation errors "across a wide span of applications," and 70% to 80% error removal when inspections are practiced with greater rigor and frequency [Dye92, pp. 9–10]. Recently, it has been suggested that the process can be made more reliable if N inspection teams (coordinated by a single Moderator) are used [SMT92].

One of the main advantages of the structured walk-through over other forms of verification and validation is that it does not require an executable program; it can therefore be applied early in the design cycle to help uncover faults and oversights before they become entrenched.

Just as verification can be supported by a very formalized process, so can testing in support of validation. There are numerous metrics for assessing the coverage of a series of tests (e.g., visiting all statements, or taking all branches, and more sophisticated measures based on data flow criteria [RW85, Wey88]), and tools are available to monitor coverage against chosen metrics and to administer regression tests. Note, however, that these automated processes mainly concern structural test criteria, whereas DO-178B rightly also stresses the importance of requirements-based testing [RTCA92, Section 6.4].[1]

Notwithstanding their labor and test-intensive character, it is important to recognize that established processes for the development of critical software and hardware have proved fairly effective in practice. Clearly, these processes should not be changed gratuitously, and a good case needs to be made for the value added by innovations such as formal methods.

Level 1: *Use of concepts and notation from discrete mathematics.* The idea here is to replace some of the natural language used in requirements statements and specifications with notations and concepts derived from logic and discrete mathematics. Examples include the notions and terminology of set theory, and the various special kinds of relations and functions (such as transitive relations,

[1] Other criteria for test adequacy are based on reliability growth models; testing is stopped when the model predicts that the software has become "reliable enough" [CDM86, MA89b]. These are of questionable utility for critical systems, because of the extreme reliabilities required: see Section 3.1.

or one-to-one functions). Generally speaking, applications at this level are not too concerned with the precise details of the formal system employed (e.g., no-one is going to worry about whether the set theory employed is Zermelo-Fraenkel or that of Gödel, von Neumann, and Bernays), and proofs, if any, are performed in the informal style typical of a mathematics textbook. This informal style matches the way mathematics is used in most other engineering disciplines and, indeed, the way most of mathematics itself is performed. Used in this way, discrete mathematics contributes to software and hardware engineering in three ways:

- It provides an economical but versatile collection of mental building blocks that can help in the development and presentation of clear, precise, and systematic statements of requirements, assumptions, specifications, and designs.
- It provides a compact notation that allows these statements of requirements, assumptions, specifications, and designs to be written down and communicated with less ambiguity than natural language, and in the reasonable expectation that author and reader will share the same understanding of the notation.
- It may provide some "laws" (i.e., theorems or derived rules of inference) that can systematize or guide the elaboration of the specification or design from one level to another.

Level 1 formal methods usually augment an existing development method and are not normally part of a wholesale revision to the process. An exception may be the Cleanroom methodology pioneered by Mills [MDL87] and Dyer [Dye92].[2] Less revolutionary is the A7 (also known as the "Software Cost Reduction" or SCR) methodology pioneered by Parnas and others at the Naval Research Laboratory [Hen80] (see [vS90, vSPM93] for more recent treatments).

Level 2: *Use of formalized specification languages with some mechanized support tools.* Simply enjoining staff to "use discrete mathematics" is probably not an effective management technique for projects involving more than a few members; standards and conventions are generally desirable, and these may be

[2]The Cleanroom uses formalized specifications, a limited and structured set of primitives for design, correctness arguments as the basis for formal inspections, no unit or other testing by developers, and statistical process control and reliability measurement. "If the total Cleanroom process is adopted for software development, it represents a radical departure from current software development practice... The process introduces new controls for software development, imposes new roles and responsibilities on the various engineering disciplines, eliminates some seemingly core methods from the development process, and raises the level of training and proficiency required in the engineering disciplines" [Dye92, pp. 4–5].

systematized as an informal "specification language." It may then become attractive to fix a concrete notation for the language and to provide mechanized support tools. Formal methods on the lower rungs of Level 2 may provide tools, such as syntax checkers and prettyprinters, that examine only the surface features of specifications; better supported methods may provide deeper tools, such as typecheckers, that examine rather more of their content. All Level 2 applications of formal methods should retain or enhance the benefits of Level 1 applications, and may provide additional benefits:

- Specification languages generally provide more than just a standardized notation for discrete mathematics: they usually also address software engineering concerns and allow specifications to be structured into units (e.g., modules, abstract datatypes, or objects) with explicitly specified interfaces. Some specification languages form part of a fully developed software engineering methodology (e.g., VDM [Jon90]).
- Mechanized checking tools allow accurate detection of certain types of faults; other tools may make it simple to generate up-to-date documentation and summary reports, and to trace dependencies through large specifications.
- It may be possible to quickly generate a simulation, "animation," or prototype implementation from a specification, either automatically, or with some human guidance [AJ90, HI88]. These can be very useful ways to explore certain properties of a specification. Some languages used for specification are directly executable (e.g., OBJ [FGJM85]), but this rather compromises their status as vehicles for specification and renders them closer to high-level programming languages [HJ89].

Standardization and mechanization bring limitations as well [Nau82]: it is usually no longer possible to invent new or modified notations—instead all specifications must be built from the constructs provided by the specification language concerned. Some specification languages may be optimized for a particular class of applications, or for a particular form of analysis, and may prove cumbersome or inapplicable in contexts other than those intended. Generally speaking, the richer and more widely applicable a specification language, the harder it is to build support tools for it. Thus, the richest and most convenient specification languages tend to have relatively limited mechanical support.

Mechanized theorem proving is the dividing line between Level 2 and Level 3 applications of formal methods in my classification. Proofs in Level 2 applications are usually performed informally, as in Level 1. Level 2 methods often, however, provide explicit formal rules of deduction, so that these proofs could, in principle, be done formally, albeit by hand. The term *rigorous* (as opposed to *formal*) is often used to describe the style of proof employed in Level 2 applications.

Level 3: *Use of fully formal specification languages with comprehensive support environments, including mechanized theorem proving or proof checking.* The most truly formal of methods are those that employ a specification language with a very direct interpretation in logic, and with correspondingly formal proof methods. Once proof methods are completely formalized (i.e., reduced to symbol manipulation), it becomes possible to mechanize them. Mechanization can take the form of a *proof checker* (i.e., a computer program that checks the steps of a proof proposed by the human user), or of a *theorem prover* (i.e., a computer program that attempts to discover proofs without human guidance) or, most commonly, of something in between.

The advantages of this fully formal approach are that requirements, specifications, and designs can be subjected to searching examination, and that mechanization eliminates faulty reasoning with almost complete certainty; its disadvantages are that fully formal specifications and machine-checked proofs can be expensive to develop, and that many of the specification languages with full mechanical support have been rather impoverished notations based on restricted logics. This means that specifications may have to be contorted to fit the restrictions of the language, and can therefore be difficult to write and to read.

The costs of developing fully formal verifications are often exacerbated by the need to develop formalizations of the supporting theories or mathematical "background knowledge" (e.g., the properties of the arithmetic mean, or of permutations) that are taken for granted in less formal approaches. As formal verification becomes more widely practiced, I would expect verified libraries of such theories to become available, thereby reducing the cost of subsequent developments.

Higher levels of rigor in the application of formal methods are not always superior to lower levels: depending on the benefits desired from use of formal methods, the criticality of the application, and the resources available, any of the four levels (including Level 0) can be the appropriate choice. For example, if formal methods are used simply as documentation, then Level 1 may be the best choice; but if they are used to justify the design of a novel and critical component, then Level 3 may be preferred; Level 2 will fall somewhere between these extremes, and Level 0 may be the sensible choice for routine applications that have been handled adequately in the past by standard practices. I discuss factors that should influence the choice of rigor in more detail in Chapter 3, when I have prepared more of the groundwork.

Notice that it is feasible to use a formal method at a lower level of rigor than that for which it is primarily intended. For example, a system capable of supporting Level 3 could be employed in a Level 2 development by using just its specification language, and not its theorem-proving capabilities. There can be advantages and disadvantages to such a choice. On the one hand, methods intended for the more rigorous levels may embody solutions to problems whose existence might not even be recognized at lower levels (e.g., a mechanically-checked definitional principle which guarantees that definitions do not introduce inconsistencies). On the other hand, and as I noted earlier, increasing formality and mechanization impose demands on a specification language that often (but need not) reduce its expressiveness and hence its convenience as a notation.

Many individual formal methods are migrating upwards through the levels I have enumerated. For example, a notation that starts in Level 1 (e.g., Z, which in its early days was referred to as a "style" rather than a language [Abr80a, Sør81]) might be embellished and standardized and provided with elementary tools, thereby moving it into the lower reaches of Level 2 [Spi93]. Later, the tool support might be increased, moving it upward in Level 2, and theorem-proving support could then be added to move it into Level 3.[3] Conversely, developments of some of the formal methods at Level 3 are aimed at reducing the overheads associated with full rigor and mechanization, thereby broadening their appeal and range of application downwards into Level 2. (I describe some of the evolution of mechanical support for formal methods, and speculate on some of the likely courses of development, in Section 2.6.) In the future, therefore, individual formal methods may be less restricted than at present in the levels of rigor for which they are best suited.

Developments such as these blur some of the distinctions in my classification of formal methods. Nonetheless, I consider the classification valuable because the unadorned term "formal methods" covers a wide range of techniques that have very different characteristics, costs, benefits, and drawbacks. While it will be necessary to evaluate the individual attributes of any formal method that may be offered in support of certification (see Section 3.5), my classification serves to highlight the gross distinctions. Notice in particular that in Europe, and especially in the United Kingdom, many uses of the term "formal methods" refer to methods (e.g., RAISE [RAISE92], VDM [Jon90], Z [Spi93]) that I would classify as Level 2, whereas North American usage normally refers to the most rigorous and powerfully mechanized end of Level 3 (e.g., the Boyer-Moore prover "Nqthm" [BM88],

[3] However, the theorem-proving capabilities of "proof assistants" such as Balzac (a related system is known as Zola) [Har91], the B-Tool [LS91], and *mural* [JJLM91], are so limited at present that I hesitate to describe them as Level 3.

Eves [CKM+91], PVS [ORS92], or SDVS [CFL+91]).[4] Failure to recognize these different usages can be a source of great misunderstanding.

2.2 Extent of Application in Formal Methods

Just as it is possible to vary how rigorously formal methods are applied, so we can vary the *extent* of their application. There are at least three dimensions to the notion of extent:

- 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.
- Formal methods can be applied to all, or only to selected, components of the system. More generally, we can vary the level to which formal methods are applied according to the component.
- Formal methods can be applied to system properties other than full functionality. Traditionally, formal methods have been associated with "proof of correctness," that is, with ensuring that a system component meets its functional specification. For the purposes of verification, not all its functional properties may be equally important; we may then apply formal methods only to the most important functional properties. In flight-critical systems, it may sometimes be more important to be sure that a component does *not* exhibit certain kinds of catastrophic failures, rather than that it has certain positive properties.

In the following subsections, I examine each of these notions of extent in more detail.

[4]This distinction between North American and European approaches is a very coarse one; for example, the British system called HOL [GM93] is generally used for Level 3 applications, whereas the American Larch notation [GH93b] is generally used in Level 2 applications. The British tool SPADE [COCD86] (MALPAS [MC90] is somewhat similar) is sometimes used with mechanized proof checking and therefore has some of the characteristics of a Level 3 formal method, but does not fit my classification very well. Primarily, SPADE supports the analysis of executable programs (written in a variety of languages), including static code analysis (e.g., checking for variables that are not used, and for those that are used but not set, as suggested in DO-178B [RTCA92, Subsection 6.3.4.f]), and sophisticated flow analysis [BC85]. SPADE also allows programs to be annotated with assertions expressed as pre- and post-conditions that can either be checked at run-time, or used to generate verification conditions that are submitted to the SPADE proof checker [Car89]. Although it uses formal methods in its analysis of programs, SPADE does not have a fully expressive general-purpose specification language and is therefore somewhat different from other techniques that I would classify on Levels 2 and 3. This is not a criticism of SPADE, which has been used quite successfully in some aircraft projects (e.g., [OCF+88], where it was used to verify consistency between Z8002 assembler code implementing the primitive elements of the LUCOL language—used in the full authority digital fuel control system of the Rolls Royce RB211-524G engine—and the "fact sheets" that constitute their specification), but a limitation in my classification scheme.

2.2.1 Formal Methods in the Lifecycle

Some of the earliest formal methods were associated with "proofs of correctness" of small programs represented as flowcharts [Flo67].[5] The basic idea is to attach a logical assertion to each segment of straight-line code in the program and to prove that these assertions will be true for every execution path through the program. This can be done by considering just the pairwise combinations from one segment of straight-line code to another. These pairwise combinations give rise to lemmas (small theorems) called "verification conditions" (VCs); an induction principle ensures that the program is correct if all the individual VCs are true (hence this method is sometimes known as that of "inductive assertions").

An example of this approach can be based on the flowchart shown in Figure 2.1, which describes a program to calculate the positive integer square root z of its input x by the method of linear search (i.e., counting up from 0 until the square root is found). This program is obviously trivial and very inefficient, but serves to illustrate the technique. The specification for this program is that, given an integer x satisfying $x \geq 0$, it should produce an integer z satisfying $z^2 \leq x \wedge (z+1)^2 > x$.[6]

The program is described by the statements enclosed in boxes, and by the "flow of control" that is implicit in the lines connecting them. The logical assertions used in specifying and reasoning about the program are the three formulas not enclosed in boxes. Notice that similar identifiers and expressions appear in both the assertions and the program statements, but that they use different fonts and are written slightly differently. The letters `x` and `z` appearing in the program statements represent *program* variables, whereas the x and z appearing in the assertions are *logical* variables. Logical variables represent integer values, so that an equation such as $z = z + 1$ makes no sense: there is no integer that equals itself plus 1. Program variables, on the other hand, represent "storage locations," so that the expression `z := z + 1` does make sense: it means take the value stored in location `z`, increment it by 1 and store the result back in location `z`. The relationship between z and `z` is that the former captures the value stored in the latter at the instant the flow of control passes the assertion in which z appears. The assertions therefore document the instantaneous relationships among the values stored in the program variables.

Three straight-line segments of code connect the assertions: from the entry assertion (at the top) to the assertion in the loop (this latter kind of assertion is

[5] The idea that one could reason about the correctness of programs is almost as old as programming; for example, Babbage wrote about "Verification of the Formulae Placed on the [Operation] Cards" of his analytical engine in the 19th century [Ran75b, pp. 45–47] and Turing proved the correctness of a program in 1949 [MJ84, Tur92].

[6] The infix symbol $\wedge$ means "and"; I also use $\vee$ for "or," $\supset$ (or sometimes $\Rightarrow$) for "implies," $\equiv$ (or sometimes $\Leftrightarrow$) for "if and only if," and the prefix symbols $\neg$ for "not," $\forall$ for "for all," and $\exists$ for "there exists."

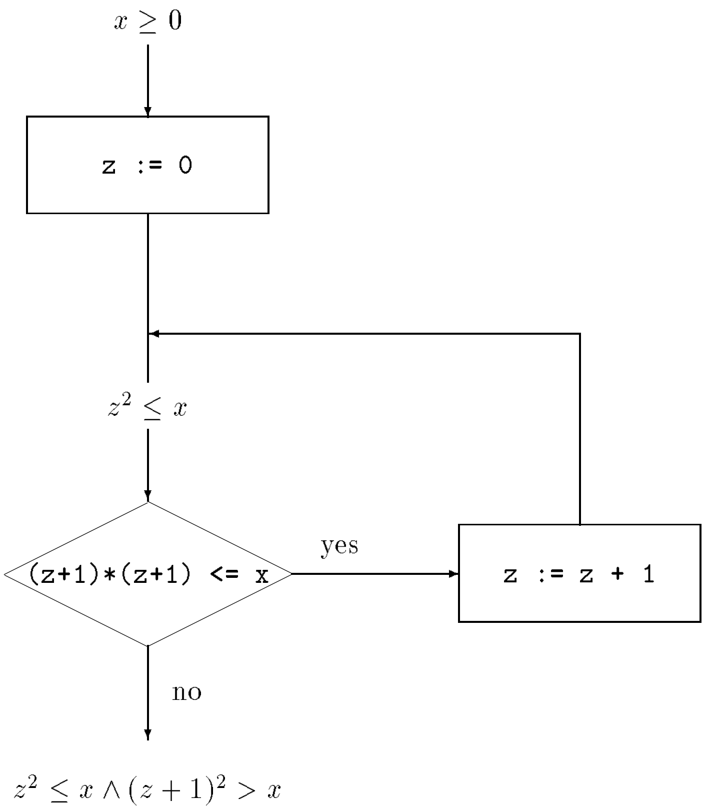

Figure 2.1: An Annotated Flowchart

called a "loop invariant"), from the loop invariant around the loop and back again, and from the loop invariant to the exit assertion (at the bottom). Each of these gives rise to a verification condition: the first requires that if we assume $x \geq 0$ and then set `z` to zero (the effect of executing the statement `z := 0`), we will satisfy $z^2 \leq x$; the second requires that if we assume $z^2 \leq x$ (the loop invariant) and $(z+1)^2 \leq x$ (from taking the "yes" branch of the test `(z+1)*(z+1) <= x`), and increment `z` by one (the effect of the statement `z := z+1`), then the *new* value of `z` will also satisfy the loop invariant $z^2 \leq x$; the final segment requires that if we assume $z^2 \leq x$ (the loop invariant) and $(z+1)^2 \not\leq x$ (from taking the "no" branch of the test), then we will satisfy the exit assertion $z^2 \leq x \wedge (z+1)^2 > x$.

In the notation of logic, these VCs are:

1. $x \geq 0 \wedge z = 0 \supset z^2 \leq x$,
2. $z^2 \leq x \wedge (z+1)^2 \leq x \supset (z+1)^2 \leq x$, and
3. $z^2 \leq x \wedge (z+1)^2 \not\leq x \supset z^2 \leq x \wedge (z+1)^2 > x$.

It is clear that the VCs for this example are true, and so we can conclude that the program represented by the flowchart satisfies the specification for integer square root—if it terminates. Termination can be established by arguing that the value of `z` increases by one each time round the loop, and so the branch condition `(z+1)*(z+1) <= x` must eventually become false and cause the loop to terminate.

As I have presented it here, this verification is an example of a Level 1 application of formal methods in my classification scheme. Early mechanizations of this approach to program verification took programs written in an Algol-like language and annotated with logical assertions, generated the corresponding VCs with a program called a "Verification Condition Generator" (VCG), and used the theorem-proving technology of the time to prove the VCs under human guidance [Goo70, ILL75, Kin69]. A difficulty with this approach is that users' intuition usually rests in the program, but the VCs that they are asked to prove are logical formulas that appear divorced from that context. Consequently, most presentations of program verification in textbooks, and most applications of program proofs done by hand, use an alternative approach introduced by Hoare [Hoa69]; instead of transforming programs into VCs in an ordinary logic and doing the reasoning there, Hoare's approach extends the logic to include program elements and makes it possible to reason about programs directly.

In Hoare's approach, properties of program fragments are described by triples (sometimes called "Hoare sentences") of the form $\{P\}\mathcal{S}\{Q\}$ where P and Q are assertions about the program state (called the precondition and postcondition, respectively), and $\mathcal{S}$ is a program fragment; the interpretation is that if the program fragment starts off in a state satisfying P then, if it terminates, it will do so in a state satisfying Q. Program fragments are composed using rules of inference such as the following, which is a simplified form of the rule for a *while* loop:

$$\frac{\{P \wedge B\}\ \mathcal{S}\ \{P\}}{\{P\}\ \texttt{while}\ B\ \texttt{do}\ \mathcal{S}\ \{P \wedge \neg B\}}. \tag{2.1}$$

Rules of inference written in this way are interpreted as saying that if the formula above the line is true, then we may conclude that the formula below the line is true also. This particular rule says that if the property P is preserved by the program fragment $\mathcal{S}$ (assuming B is true on entry), then if P is true on entry to the schematic *while* loop "`while` B `do` $\mathcal{S}$," both P and the negation of B will be true on exit (if the loop terminates). For our integer square root example, P is $z^2 \leq x$, B is $(z+1)^2 \leq x$,[7] and $\mathcal{S}$ is `z := z+1`, so that as the formula above the line we have (from the rule for assignment, which I will take as given)

$$\{z^2 \leq x \wedge (z+1)^2 \leq x\} \ \texttt{z := } z+1 \ \{z^2 \leq x\}$$

(notice that the z in the postcondition is the value of the program variable `z` *after* it has been incremented) and can therefore conclude the corresponding formula below the line:

$$\{z^2 \leq x\} \ \texttt{while (z+1)*(z+1) <= x do z := z+1} \ \{z^2 \leq x \wedge (z+1)^2 > x\}.$$

When combined with the rule

$$\{true\} \ \texttt{z := 0} \ \{z = 0\}$$

for the initialization statement, and the rule for sequential composition (I will pass over the details), this allows us to conclude that the program

```
z := 0;
while (z+1)*(z+1) <= x do z := z+1
```

indeed satisfies our specification for integer square root. If I had explicitly given all the rules for the programming language logic concerned, and filled in all the steps that were elided, then this would be an example of a Level 2 application of formal methods in my classification scheme. Notice that as well as supporting program verification, Hoare's "proof rules" can also be used to *define* the semantics of the program language concerned [HW73] (e.g., (2.1) can be taken as a definition of the semantics for a *while* loop).

Although Hoare's approach and its descendents (notably Dijkstra's weakest preconditions [Dij76]) are attractive and are the ways program derivation and verification are treated in textbooks, most mechanized program verification systems continue to use VCGs, rather than mechanize Hoare's method directly.

A rather different, and older, tradition in programming language semantics and program verification derives from the work of McCarthy [McC60]. This approach is based on functional programming (and originally was closely identified with the LISP programming language) in which the notion of an implicit program "state" is not required. In this approach, programs are functions and can be understood in

[7] As in the flowchart, this B occurs as an expression of ordinary logic in the pre- and postconditions, where I write it as $(z+1)^2 \leq x$, and as a corresponding expression in the program, where I write it as `(z+1)*(z+1) <= x`.

a fairly conventional mathematical framework (although it took researchers many years to take care of all the theoretical details).

The application considered in all these early styles of formal methods was chiefly program verification; the programs concerned could be quite intricate [MP67] and sometimes implemented very clever algorithms, but they typically made little use of datatypes other than booleans, integers, and arrays of these, and were essentially stand-alone programs rather than systems. Then in 1972, Hoare [Hoa72] presented a method for verifying correctness of data representations, and ideas for specifying the modules and abstract datatypes of larger systems were suggested by Parnas [Par72]. The Hierarchical Development Methodology (HDM) was an early attempt to provide tool support for these ideas [RLS79, SLR78]. Meanwhile, most notably in Europe, very rich formal specification notations [Bjø81, Sør81] were being developed and applied to the requirements and top-level specifications of larger objects such as operating systems [Abr80b] and programming languages [BBH$^+$74].

The integer square root example does not lend itself to demonstration of these later techniques, so I will introduce a new (but still very elementary) example. The paradigmatic abstract datatype is a pushdown stack—an object provided with three operations: *push*, *top*, and *pop*. The *push* operation allows an element to be stored on the "top" of the stack, *top* returns the element stored on top of the stack, and *pop* returns the stack with the top element removed; sequences of *pop* operations remove elements in a last-in/first-out order. A formal specification in a functional style for the stack operations might declare the *signatures* (i.e., the number and types of the arguments, and the type of the value returned) of the three operations as follows:

- *push*: $[elem,\ stack \rightarrow stack]$,
- *pop*: $[stack \rightarrow stack]$, and
- *top*: $[stack \rightarrow elem]$,

where *elem* denotes the type of elements that may be stored in the stack. The behavior of the operations would be specified by means of the following two axioms:

- $pop(push(e, s)) = s$, and
- $top(push(e, s)) = e$,

where s and e are variables standing for an arbitrary *stack*, and an arbitrary *element*, respectively.[8] The points worth noting here are the functional style (i.e., the

[8] There are a number of deficiencies with this specification: for example, it is silent on the existence of an empty stack (and therefore on the results of applying *pop* or *top* to such a stack), and on the possibility of exceeding the number of elements that can be pushed onto a stack. I will return to these issues later (in Subsection 2.3.1.1).

stack is an explicit argument to the operations) and the use of axioms to specify the interrelationships among the three stack operations without suggesting an implementation. This style of specification is called *property-oriented*, because it specifies (some aspects of) a component in terms of the properties it is required to possess.

A more concrete specification style might specify a stack in terms of a linked list of elements, or of an array and a pointer. For example, it might specify that a stack is represented by a record:

$$
\begin{array}{ll}
\mathit{stack}\colon & \textbf{type} = \ \textbf{record} \\
& \quad \mathit{array} : [\ \mathit{nat} \rightarrow \ \mathit{elem}], \\
& \quad \mathit{pointer}\colon \ \mathit{nat} \\
& \textbf{end record}
\end{array}
$$

and that the functions *push*, *pop*, and *top* manipulate these components as follows (the contents of the stack occupy positions $1 \ldots \mathit{pointer}$ in *array*).

$$
\begin{array}{l}
\mathit{push}(e,\ s) \ = \ s \ \ \textbf{with} \ \ [\ \mathit{pointer} := \ s.\mathit{pointer} + 1, \\
\qquad\qquad\qquad\qquad\quad \mathit{array} := \ s.\mathit{array} \ \ \textbf{with} \ \ [\ s.\mathit{pointer} + 1 := \ e]]^{9}
\end{array}
$$

$$\mathit{pop}(s) \ = \ s \ \ \textbf{with} \ \ [\ \mathit{pointer} := \ \ \mathit{pred}(s.\mathit{pointer})]^{10}$$

$$\mathit{top}(s) \ = \ \ s.\mathit{array}(s.\mathit{pointer})$$

Specifications of the kind illustrated in this example are called *model-oriented* (because they specify a component by giving a mathematical model that has the desired behavior). The disadvantages of model-oriented specifications are that they suggest a particular implementation (and thereby discourage use of, and complicate the argument for the correctness of, an implementation different from the one suggested),

[9] The **with** construct is a notation for function or record modification (also called overriding): for a function f of one argument, the construction f **with** $[x := y]$ is another function having the same signature as f that satisfies the constraint

$$f \ \textbf{with} \ [x := y](z) = \textbf{if} \ z = x \ \textbf{then} \ y \ \textbf{else} \ f(z) \ \textbf{endif}.$$

The case where f is a record is similar, except that here x and z will be field names. This notation is used to describe the behavior of function and record "updates," without requiring the notion of a "state" whose elements can be modified "in place."

[10] *pred* is the predecessor (i.e., subtract 1) function on the natural numbers: $\mathit{pred}(0) = 0$. The expression $s.\mathit{pointer} - 1$ would be type-incorrect here since it could yield a negative result, whereas a natural number is required. This specification implicitly admits the empty stack (one that has $\mathit{pointer} = 0$); *pop*ping such a stack leaves it unchanged; applying *top* to such a stack returns the arbitrary value $\mathit{array}(0)$.

and they support deductive reasoning less readily than the more abstract property-oriented style. For example, it is easy to prove the theorem

$$pop(pop(push(e, push(e, s)))) = s$$

from the two property-oriented axioms given earlier, but would be more tedious to do it from the model-oriented description. Conversely, a disadvantage of property-oriented specifications is that they are remote from implementation concerns.

Verification can reconcile these two approaches: using formal verification techniques we can establish that the model-oriented specification is consistent with the property-oriented version (i.e., the axioms of the property-oriented specification are satisfied by the model-oriented specification). Thus, a property-oriented axiomatization can be used as a top-level specification, and a provably consistent model-oriented description can be used as an implementation-level specification. In larger examples, there may be several layers of specifications going from property-oriented requirements statements through increasingly detailed (and more model-oriented) levels to the final implementation; a chain of formal verifications can then establish that the implementation satisfies the requirements.

Both the property- and the model-oriented specifications for stack shown above are *functional*. The functional style of specification means that we carry the stack along explicitly as an argument s to the functions representing the operations. In an implementation of this abstract datatype, the stack would generally be "stored" as the *state* of the abstract datatype and would be implicit to the routines that implement the operations. Hoare sentences can provide the link between a functional style of specification and program-level descriptions of routines having an implicit state. For example, we could specify the implementation of a stack as an abstract datatype with state $\mathcal{S}$ (of type *stack*), and routines *pushop*, and *popop* satisfying the Hoare sentences:

$$\{\mathcal{S} = s\}\ pushop(e)\ \{\mathcal{S} = push(e, s)\}$$ [11]

and

$$\{\mathcal{S} = s\}\ popop(e)\ \{\mathcal{S} = pop(s) \wedge e = top(s)\}.$$

[11] Another way to write this is

$$\{true\}\ pushop(e)\ \{\mathcal{S} = push(e, \mathcal{S}')\}$$

where the prime indicates the value of the state $\mathcal{S}$ in the precondition. VDM uses this style, stating the postcondition as a "predicate of two states." One advantage of doing things this way is that it allows a specification to say what doesn't change. For example, the ordinary Hoare-style specification of an integer square root program $\mathcal{P}$:

$$\{x \geq 0\}\ \mathcal{P}\ \{z^2 \leq x \wedge x < (z+1)^2\}$$

Notice that to match the way things are usually implemented, I have chosen to combine the effects of the functions *top* and *pop* into the single *popop* routine: this returns the value on top of the stack, and pops the stack as a side-effect. In order to prove a theorem such as

$$\{\mathcal{S} = s\} pushop(e); popop(e)\ \{\mathcal{S} = s\},$$

we first use the Hoare sentences above and the rule for sequential composition to reduce the problem to that of showing

$$pop(push(e, s)) = s$$

and then use the appropriate axiom from the functional specification to conclude that this is valid.

The argument in favor of separating specifications into a set of abstract operations described functionally (i.e., taking a variable encoding the state as an explicit argument), and a set of routines (operating on an implicit state) described in terms of these operations, was first argued forcefully by Guttag and Horning in 1980 [GH80].[12] Not all specification styles follow these precepts; for example, Z [Spi93] and VDM [Jon90] introduce program state and operations specified by pre- and postconditions from the very beginning. A disadvantage of this approach is that it can encourage an overly concrete specification, resulting in a state with many components and complex interrelationships. A secondary disadvantage of this approach is that it can complicate mechanized theorem proving.

The examples I have shown so far illustrate some of the techniques for applying formal methods to sequential programs. Concurrent or distributed programs raise many new issues, as do real-time properties, and the variety of essentially different techniques for addressing these issues is much greater than for sequential systems. Model-oriented specifications for concurrent and distributed systems are generally based on *process algebras* (e.g., the LOTOS specification method for telecommunications protocols [DV89, ISO88] is based on Hoare's Communicating Sequential Processes (CSP) [Hoa85] and Milner's Calculus of Communicating Systems (CCS) [Mil80]), while property-oriented specifications often use some form of temporal logic [Pnu81, Lam91, MP92].

can be satisfied by the program that ignores the input value of x and always sets x to 4 and z to 2. The VDM-style specification

$$\{x \geq 0\}\ \mathcal{P}\ \{z^2 \leq x' \wedge x' < (z+1)^2 \wedge x' = x\}$$

rules this out. How to specify what does *not* change is called the "frame problem."

[12] This excellent paper is full of insights that remain pertinent more than a decade later. Among other things, it recommends that specification of exception conditions belongs to the routines—abstract operations should deal with the ideal case. Many of these ideas are embodied in the Larch family of specification languages [GH93b].

Even an outline of some of the range of formal methods for concurrent and distributed systems would require far more space than is feasible here, so I will give just a single example to illustrate the subtlety of the reasoning that can be required for even tiny concurrent programs.

The example is known as Fischer's real-time mutual exclusion algorithm [Lam87] and is something of a test piece for formal methods intended to support reasoning about concurrent, real-time programs. The idea is that several processes, each identified by a unique number, share a state variable x, which is initially 0. Whenever any process wishes to enter its critical section, it waits until it sees that $x = 0$. Within at most *max-delay* time units, it then sets x to its own number and proceeds to the "check" action. In the check action, it waits for at least *min-pause* units and then looks to see if the value of x is still its own number (some other process could have previously seen $x = 0$ and be in a race with this one); if it is, the process immediately enters its critical section (otherwise it goes back to waiting for x to return to 0). On leaving its critical section, the process resets x to 0. The claim is that this algorithm achieves mutual exclusion (i.e., no two processes are ever in their critical sections simultaneously), provided *max-delay* < *min-pause*. Notice that the specification for this algorithm (i.e., mutual exclusion) does not involve real time. Real-time reasoning is needed only to show correctness of the algorithm.

Despite its small size, and the simple nature of its individual operations, it is not at all obvious to most people that this algorithm works, and no amount of testing is likely to increase their confidence. I will give an informal presentation of one argument for its correctness in which I consider a simplified version of the algorithm that neglects the looping behavior: that is, processes simply stall at the check action if they find x different to their own number. The three steps in the algorithm can then be described as follows, where i is the number of the process concerned.

init: if $x = 0$, goto try;

try: $x := i$, goto check;

check: if $x = i$, enter critical section.

I write $try(i)$ to indicate that the i'th process is executing the *try* action, and so on for the other actions; $cs(i)$ means that process i is in its critical section. The correctness requirement is then

$$cs(i) \wedge cs(k) \supset i = k.$$

To express real-time properties, I will use a "since" operator (denoted by vertical bars) that is due to Shankar [Sha93]: at any point in the program's execution, $|p|$ records the time that has elapsed since the last state in the past at which p was true. If p was never true in the past $|p| = \infty$. Since $|p|$ is concerned only about the past, its value is unaffected by whether p is true *now*. Using the *since* operator, the two timing constraints on the program can be expressed as follows.

1. $\forall i\colon try(i) \supset |init(i)| \leq max\text{-}delay$.
   When process i is executing its *try* action, the time since it was in its *init* action is at most *max-delay*.

2. $\forall i\colon cs(i) \supset |check(i)| + min\text{-}pause \leq |try(i)|$.
   When process i is in its critical section, the time since it was in its *try* action exceeds the time when it was last in its *check* action by at least *min-pause*.

The argument for correctness follows from four lemmas.

1. Whenever $cs(k)$, the time since $try(k)$ is at most the time since $x = 0$. This follows because the *try* action sets x to k at the same time it enters the *check* action.

2. Whenever $try(i)$, the time since $x = 0$ is at most the time since $init(i)$. This is because $x = 0$ is a precondition for the action at *init*.

3. Whenever $cs(k)$, then $\neg try(i)$. Otherwise the time since $x = 0$ is at most *max-delay* (by Lemma 2 and Timing Constraint 1) and at least *min-pause* (by Lemma 1 and Timing Constraint 2), contradicting $max\text{-}delay < min\text{-}pause$.

4. Whenever $cs(k)$, then $x = k$. Only the action $try(i)$ could falsify $x = k$, but Lemma 3 rules this out.

Shankar has formalized the *since* operator using the PVS verification system [ORS92] and has mechanically checked the correctness argument given above for Fischer's mutual exclusion algorithm and for some other simple problems [Sha93].

This example should not be taken as representative of the formal techniques that have been proposed and applied to the problems of reasoning about concurrent and distributed execution and about real-time properties, but it is representative of their main focus: that is, to find ways to limit the extent to which the behaviors of other processes need to be considered when contemplating the behavior of "this" process.

Formal methods have developed considerably since the pioneering studies of McCarthy, Floyd, and Hoare, and very many people have contributed many important techniques (Jones [Jon92] gives a partial history). One trend in these developments has been to consider more complex models of computation (incorporating concurrent and distributed execution) and more difficult properties of those models (such as real-time properties). Another trend, and the one that is perhaps most responsible for increased interest in formal methods, has been a shift in focus from the later to the earlier stages of the development lifecycle: from program verification to requirements capture and system specification.

The system lifecycle model is one of the key concepts in modern software and hardware engineering. Its premise is that development and implementation are carried out in several distinguishable, sequential phases, each performing well-defined tasks. There have been many refinements to the basic model: Royce's *Waterfall* model [Roy70], for example, recognized the necessity for feedback between phases and recommended that it should be confined to adjacent phases; Boehm's *Spiral* model [Boe88] (see also Brooks [Bro87]) advocates a more iterative approach, including rapid prototyping.

In all cases, one of the outputs of each phase is a document that serves as the basis for evaluating the outcome of the phase and that forms a specification for subsequent phases. The traceability of requirements through the lifecycle is particularly important in critical systems: we must be sure that every requirement is satisfied by the implementation, and that the implementation does not provide functions other than those required. Verification is an element in detailed requirements tracing: as already noted, it is the process of determining whether the output of each phase "fully and exclusively" implements the specification of its preceding phase. Formal methods substitute formal specifications for the informal design documents of some of the phases of the development lifecycle and, possibly, formal verification for the manual examination of documentation and specifications that constitutes conventional verification.

The motivation for a disciplined lifecycle model is to try to catch faults of design or requirements as reliably and as early as possible. The more phases of the lifecycle that separate the commission and detection of a fault, the more expensive it is to correct, and potentially the more dangerous the consequences if it is not detected. For example, it is usually cheap and simple to correct a coding bug caught during unit test, and it is usually equally simple and cheap to insert a missed requirement that is caught during system requirements review. But it can be ruinously expensive to correct such a missed requirement if it is not detected until the system has been coded and is undergoing integration test. Data presented by Fairley [Fai85, pp. 48–50] show that it is 5 times more costly to correct a requirement fault at the design stage than during initial requirements, 10 times more costly to correct it during coding, 20 to 50 times more costly to correct it at acceptance testing, and 100 to 200 times more costly to correct the problem once the system is in operation. For critical systems, the motivation for disciplined lifecycle processes is assurance: use of systematic methods at every stage for preventing the introduction of faults, and for finding and eliminating faults, is the principle means for instilling confidence that the final system will be free of critical defects.

Verification is joined by validation to make up the methodology of "verification and validation" (V&V) that is used to provide quality control and assurance within a disciplined software lifecycle. Validation is the process by which the product of a development phase is shown to satisfy the original user requirements. Traditionally, this has been done by testing the final product but, as described in the first chapter, modern development practices recognize the value of also doing validation earlier in the lifecycle, in order to check satisfaction of the user's requirements before the full system is built.

Realistic demonstrations of these early-lifecycle activities are necessarily rather large, so I will illustrate the ideas with near-trivial examples.

Suppose a requirements specification contained the following sentences (adapted from [Inc88, pp. 77–78]):

1. If the switch is on or the monitoring computer is in a ready state, then both a recognition signal and a functioning message are sent to the control computer.
2. If a functioning message is sent to the control computer or the system is in a normal state, then pilot commands will be accepted.

We might want to validate these requirements by checking that they entail some expected additional properties, such as:

1. If the switch is on, then pilot commands will be accepted.

This example is so simple we can formalize it within propositional logic (the simplest of all logics) as follows.

Given

$$\mathit{on} \vee \mathit{ready} \supset \mathit{recognition} \wedge \mathit{functioning}, \text{ and}$$

$$\mathit{functioning} \vee \mathit{normal} \supset \mathit{accepted},$$

prove

$$\mathit{on} \supset \mathit{accepted}.$$

This particular theorem is easily proved by propositional reasoning. Realistic examples of requirements and design specifications generally require richer specification notations and more powerful theorem-proving methods than this.

In the example just given, the formal "method" is simply the use of formal notation to replace English phrases. Particular classes of problems admit more specific methods that can provide systematic ways to approach and organize their requirements specifications. One such method uses state machines to model the behavior of a required system: the description consists of a specification of the components of the state data to be recorded by the system, a logical expression called an "invariant" that specifies how the components of the state are related, an initialization operation, and the various operations that access and modify the state of the system (specified by means of pre- and post-conditions). The proof obligations incurred by such a specification require that each operation (including initialization) be shown to leave the system in a state that satisfies the invariant.

State machine modeling is suitable for systems and system components that have the character of a database or an abstract datatype. There are also many methods particular to other kinds of applications—for example, process-control systems (the A7 methodology [vSPM93]) and sequential hardware [Gor86].

We now need to ask where formal methods should be introduced into the software and hardware development lifecycle. A purist might answer "everywhere," and it is certainly technically feasible to apply formal methods all the way from requirements capture through to code or circuit verification. However, formal methods are relatively expensive to apply at present and must compete on cost and effectiveness with other methods for quality control and assurance, so some selectivity in their application is inevitable in most projects. The crudest form of selection amounts to a dichotomy: we can prefer to apply formal methods late in the lifecycle, or early. Both positions have strong advocates.

Late-lifecycle advocates observe that it is the running program code (or gate layout in the case of hardware) that determines the behavior of the system; unless the code (or gate layout) has been verified, formal methods haven't done anything that is real. It is easy to see the flaw in this argument: what good does it do to verify the code against its detailed specification if that specification could be wrong? Shouldn't the earlier stages of the lifecycle be subjected to at least the same degree of scrutiny as the final code? Early-lifecycle advocates take this argument to its conclusion and claim that the early lifecycle is the source of the most dangerous and expensive faults, and that the later stages of the lifecycle are adequately served by conventional methods.

In the next subsection I examine the case for formal methods in the late lifecycle; the early lifecycle is considered in the section after that; the discussion is then summarized in the final subsection of this section. These discussions on formal methods in the lifecycle are mostly theoretical (concerning the kinds of faults that might occur and might be eliminated). Data on system failures in aerospace and similar applications is considered much later, in Section 3.3.

2.2.1.1 Formal Methods in the Late Lifecycle

In examining where in the lifecycle formal methods may best be applied, we need to consider the *effectiveness* and the *cost* of formal methods relative to other methods. In both these dimensions (cost and effectiveness), there is a strong case against applying formal methods late in the lifecycle, especially in the form of program verification, and especially at the higher levels of rigor.[13] Formal verification is the process of demonstrating, by formal means, the consistency of two specifications; in the case of *program verification*, one of the specifications is executable program text. It is reasonable to suppose that the cost of a formal verification increases with the size of the two specifications that are be shown consistent, and it is an observable fact that specifications get bigger as they get more detailed and closer to executable code. Program verification is therefore generally more expensive to perform than formal verification between specifications generated earlier in the lifecycle simply because the program text and its detailed specifications are large objects. In addition, program verification is expensive because it must deal with the notion of an implicit program "state," whereas specifications and verifications at earlier stages of the lifecycle can model the system in purely functional terms, and thereby remain in the realm of ordinary logic, where theorem proving is generally more straightforward and efficient.[14]

For these reasons, the costs of mechanically-checked (i.e., Level 3) formal program verification are such that it has seldom been attempted for programs of more than a few hundred lines (the maximum achieved is probably around a few thousand lines in certain computer security applications). Program verification at Level 2 may be even less practical: the attention to detail required in formal program verification is such that its rigorous application without mechanical assistance is all but impossible for examples beyond those that appear in textbooks. Program verification at Level 1, however, corresponds to techniques such as the "verification-based inspections" used in the Cleanroom methodology. Published data suggest that these methods are very cost-effective ("early error-detection rates in the 90% range" [Dye92, page 10]). Certain computer security applications check "specification to code correspondence" at a similar level [Sol82, YM87]. Even at Level 0 (no use of formal methods), the scrutiny applied to program text in life-critical applications is very thorough, if informal (e.g., structured walk-throughs). And, of course, testing is performed very directly and thoroughly at the program code level.

[13] At least for the purposes of quality control (i.e., eliminating bugs); the case for program verification in quality assurance (i.e., showing that bugs have been eliminated) is examined a little later.

[14] The Hoare-style verification of the stack routines performed earlier in this section illustrate the transition between a purely functional external specification and operations that manipulate a program state.

Now the purposes of V&V at the later (coding) stages of the lifecycle are: firstly, to detect and eliminate faults introduced at those stages and, secondly, to provide assurance that no such faults remain. As I have noted, the evidence from current experience in safety-critical systems seems to be that conventional methods of review, together with testing, are very effective in achieving the first of these purposes. When applied sedulously, as in aerospace applications, these conventional methods of V&V have been shown to be capable of achieving very low incidences of failure due to faults introduced late in the lifecycle. Similar observations seem to apply to hardware design, where extensive simulation and symbolic testing techniques are very effective at eliminating faults from the later stages of the lifecycle.[15]

But for safety-critical systems, it is not enough to eliminate almost all faults introduced in the later stages of the lifecycle; we need *assurance* that they have been eliminated (or, rather, that there will be extremely few critical failures). Formal methods may not be particularly cost-effective at eliminating bugs in the late lifecycle, but what about providing assurance that no bugs remain?

The assurance provided by conventional V&V combines review with testing. By abstracting appropriately, reviews can consider all cases—whereas test coverage is necessarily partial. On the other hand, testing provides objective and repeatable evidence, whereas review is based on human judgment and consensus. Formal methods retain the advantages of reviews, but render the process more objective and repeatable. It is a matter of judgment whether this extra assurance is deemed worthwhile for particular applications. It seems plausible that program verification to Level 3 could be advantageous when the arguments for correctness are unavoidably difficult and intricate, or particularly crucial, or when test coverage is unusually limited.[16] Conversely, formal methods may offer little for applications programs with relatively few discontinuities, such as those that evaluate control laws.

Examples where correctness arguments are difficult and intricate include those concerned with concurrency and asynchrony (e.g., interrupt handling, and coordination of multiple processors), those concerned with real-time constraints, those concerned with redundancy management and fault tolerance, and those concerned with manipulation of low-level hardware features (e.g., the use of memory-management and protection hardware to provide partitioning and fault containment across tasks sharing the same processor). Examples where the correctness argument is singularly crucial are those where a single fault could lead to system failure, as in the basic

[15] According to Keutzer [Keu91], more than half of all VLSI designs are found to be defective on first fabrication, but *none* of these defects are attributable to the later stages of the design lifecycle.

[16] It could also be worthwhile when the process can be largely automated. This can sometimes be achieved when properties other than functional correctness are considered. For example, it is possible to check programs for certain information flow properties (this is done in computer security applications [DD77a]), or for the absence of certain run-time exceptions [Ger78], using specialized methods of formal analysis and theorem proving.

synchronization, communication, and voting mechanisms between the primary flight computers. Examples where test coverage is limited are those that require fault injection (e.g., fault tolerance and redundancy management) or depend on timing (e.g., communications, and synchronization).

In all these circumstances, formal methods have the potential to offer worthwhile increases in assurance. The factor that complicates realization of this potential is the intricacy of the modeling required. If we tackle these difficult or intricate arguments at the late stages of the lifecycle, we are considering not just the broad arguments for the correctness of a chosen scheme for redundancy management or synchronization, but the correctness of a particular implementation of the scheme. This will necessitate formal modeling of a host of low-level mechanisms, such as the communication and synchronization primitives of the programming language concerned, the interface between its run time support system and the process-management features of the operating system, and the characteristics of certain hardware devices. For formal methods to contribute to assurance, we must have great confidence in the fidelity of the formal modeling employed, and this is not easy to achieve with highly detailed, language-, machine-, and device-specific models. For these reasons, late-lifecycle applications of formal methods to intricate problems may be considered more of a technical tour-de-force than a practical way of increasing assurance.

So far, this discussion has focussed on formal methods applied to verification, but there are many who advocate use of formal methods purely for specification, and without the rigors of formal verification. The claim is that formal specifications provide a better starting point for coding than do informal specifications, and that in some cases specifications can be refined into programs that are guaranteed "correct by construction." The problem with this argument is that if proofs are not performed then the overall process is essentially the traditional one, but with formal specifications substituted for informal ones, and systematic refinement possibly substituted for ad-hoc development; assurance will continue to derive from traditional reviews and analyses. Thus, if quality control and assurance are to be improved through use of formal specifications, then it must be because these notations render the traditional review processes more reliable. It is plausible that this may be so, but I do not think it likely that any improvement will be all that significant, since the basic assurance processes stay the same. (In industries that do not employ the disciplined lifecycle processes used for flight software, it is possible that formal specifications and systematic refinement could have a larger impact—but this might be as much due to their impact on the development process as to their intrinsic merit.) If a case for using formal specifications during the late lifecycle in industries with effective quality control and assurance processes can be made at all, then in my opinion it is better made on the grounds of reduced costs than improved assurance. Reduced costs through use of formal specifications have been claimed in some commercial software developments (e.g., the IBM CICS example described in section 2.7), but denied in others (e.g., other parts of IBM [End93]).

2.2.1.2 Formal Methods in the Early Lifecycle

I have noted that, for current designs, conventional V&V is very effective for quality control in the later stages of the lifecycle, and adequate for quality assurance in most cases. It is the earlier stages of the lifecycle that are generally considered the most problematical. For example, Bloomfield and Froome [BF86] mention experiments "in which conventional good practice is shown to be effective at removing (at a price) implementation errors, but to leave residual errors arising from misinterpretation of the natural language specification." Other evidence indicates that fault densities are highest in the early stages of the lifecycle, and that the processes for detecting and removing these faults are very imperfect. Using formal inspections, Kelly, Sherif, and Hops [KSH92] found (in 203 formal inspections of six projects at the Jet Propulsion Laboratory) that requirements documents averaged one major defect every three pages (a page is 38 lines of text), compared with one every 20 pages for code. Two-thirds of the defects in requirements were omissions. Even in organizations where extremely thorough requirements analysis is performed, it seems less reliable than the analyses that can be performed on the products of the later lifecycle. For example, a quick count of faults detected and eliminated during development of the space shuttle on-board software indicates that about 6 times as many faults "leak" through requirements analysis, than leak through the processes of code development and review. Similar data (described in more detail in section 3.3) are provided by Lutz [Lut93a], who reports that of 197 "safety-critical" faults detected during integration testing of the Voyager and Galileo spacecraft, only 3 were programming bugs; of the remainder, half were attributed to flawed requirements, a quarter to incorrect implementation of requirements (i.e., design faults), and the rest to misunderstood interfaces.

Faults that have their origin early in the lifecycle, during the specification of requirements, or high-level design, are very costly to correct later in the lifecycle (and may be corrected by kludges that promote unreliability), can be difficult to detect in validation, and are often the source of serious failures. Leveson [Lev91], for example, states that almost all accidents involving computerized process-control systems are due to inadequate design foresight and requirements specification, including incomplete or wrong assumptions about the behavior or operation of the controlled system, and unanticipated states of the controlled system and its environment.

It is fortunate, then, that formal methods seem to find their most effective application early in the lifecycle, where conventional methods are apparently weakest. I discuss the general benefits claimed for formal methods later (in Section 2.4), but the special benefits of formal methods early in the lifecycle can be summarized as follows.

First, the early stages of the lifecycle are those when software development is most intimately connected to overall system development, and the need for effective communication between software engineers and those from other engineering disciplines is greatest. Engineers from different disciplines often construct different mental models for the same entity, and this can result in faulty or misunderstood requirements.[17] Attempts by systems and other engineers to communicate their needs to software engineers in a precise fashion often result in overly prescriptive statements of requirements (e.g., pseudocode), or in ad-hoc notations based on those of their own field (e.g., analog control diagrams) that are quite unsuited to the new demands made upon them. Formal methods have the potential to bridge this gap and to provide an effective and precise means of communication between software and other engineers.

Second, formal methods provide a repertoire of mental building blocks that assist and encourage the development of specifications that are precise yet abstract. For example, the notion of *set* allows requirements writers to describe a collection of objects without worrying about how it is represented, and quantification allows them to describe properties of the members without having to specify a search procedure or a loop.

Third, formal specifications assist early discovery of incomplete or ambiguous specifications. With natural language specifications, it is easy to overlook special cases, initializations and so on. These missing or vague elements are discovered only when a programmer realizes they are needed to complete the specification of a piece of code. Formal specifications have some similarity with programs in that they encourage a systematic elaboration of cases, and may thereby reveal incompletenesses and ambiguities at an earlier stage.

Fourth, formal specifications can be subjected to various forms of mechanical analysis that are rather effective in detecting certain kinds of faults. Simple syntax analysis identifies many clerical errors, and typechecking is a very potent debugging aid. Critics will argue that these tools merely detect faults due to the petty constraints of formal specification languages. This overlooks how rich the type-systems can be for modern specification languages: typechecking in these systems is a very strong check on internal consistency (see Subsection 2.3.1.1).

Fifth, formal specifications can support a form of validation early in the lifecycle—when few other validation methods are available. This form of validation is conducted by posing "challenges" in the form of putative theorems that

[17] In one case, the requirement that a function should "abort" in a particular circumstance was interpreted by the implementor as meaning that it should perform the "abort" operation (a jump to the enclosing block) in the programming language being used. The requirements writer meant "abort the mission."

should be valid consequences of a properly formulated specification (e.g., "if this specification says what it should, then the following ought to follow"). The putative theorems are examined by the techniques of formal verification. Other illuminating, though less conclusive, challenges can take the form of putative non-theorems: "if the following is provable, then I must have overconstrained the specification." A variant is demonstration of the consistency of a specification through exhibition of a model,[18] and a related method for demonstrating "reasonableness": for example, "if I've defined the concept of a 'good clock' correctly, then a clock that keeps perfect time should satisfy the axioms."[19] Still another variation is the analysis of specifications for certain restricted, though nonetheless important, notions of completeness such as those of Jaffe, Leveson, and Melhart [JL89, JLHM91] (see Section 2.3.2).

Sixth, formal specifications can increase the effectiveness of review-based verification in the early lifecycle. One of the reasons that verification based on reviews or inspections is effective in the late lifecycle may be that a program is necessarily a formal text. Thus, at least one end of the verification is anchored in precise terms, even if the verification process itself is informal. Verification in the early lifecycle using informal specifications is a much less effective tool. By formalizing the specifications, we may increase the precision of even informal verification—and we also make formal (Level 2 or 3) verification feasible.

Seventh, validation of the formal modeling employed is likely to be much simpler in early- than in late-lifecycle applications. For example, the correctness of an algorithm for clock synchronization rests on a few broad assumptions, such as "a processor can read another's clock with at most a small error ϵ," whereas the correctness of an implementation of the algorithm will rest on assumptions about specific mechanisms for reading clocks. It is clearly easier to validate the fidelity of models that make a few broad assumptions than those that make many highly detailed ones. So, if our main concern is whether or not the basic algorithm works, then an early-lifecycle investigation using abstract modeling and robust assumptions will generally be simpler and more useful than scrutiny of a particular implementation in the late-lifecycle.

Finally, the expense of formal methods applied early in the lifecycle is likely to be considerably less than those applied late. In particular, the size of the formal text is likely to be much smaller earlier in the lifecycle, and it is usually possible to write early-lifecycle specifications in a functional style that is much more convenient for formal verification than specifications involving an implicit state. Furthermore, the

[18] The notion of models is explained in Subsection 2.3.1.3, and in more technical terms in Appendix Sections A.2, A.3, and A.4.

[19] Young [You92] found a mistake by just this means in a formal specification that Friedrich von Henke and I constructed for the Interactive Convergence Clock Synchronization Algorithm [RvH91]. The fault is corrected in the revised version of our report and in the published version [RvH93].

early elimination of faults will save later redesign, and this may more than recoup the costs of formal methods.

In addition to the specific benefits described above and the general benefits described later, which may all be considered improvements in the design *process*, use of formal methods may also lead to an improved *design*. This can happen in two, almost contradictory, ways.

First, the cost and difficulty of formal verification is very sensitive to the complexity of the design being verified. Formal verification is therefore very effective at revealing unsuspected complications in a design, and anticipation of formal verification usually encourages renewed interest in simple designs. Other things being equal, simple designs are generally to be preferred to complicated ones, and formal methods provide useful guidance in this regard. Often, too, the improved understanding that comes from undertaking formal verification can actually suggest simpler designs.

The second way in which formal methods can assist design is through opening up new areas in the design space by allowing novel approaches to be fully explored and validated at an early stage. Many of the most difficult problems in modern systems design concern coordination of distributed activities executing in parallel. These difficulties are compounded in most safety-critical system by the need also to satisfy real-time constraints and to continue operation despite the occurrence of faults. Consequently, design and validation of the core coordination and recovery protocols is one of the most important—and difficult—steps in the early lifecycle. It is also one of the riskiest: to take an example from a different field, it is reliably reported that certain computer design companies are not working on shared-store multiprocessor architectures because they lack confidence in their ability to develop and implement the crucial cache-coherence algorithms correctly; others were still finding bugs in their cache controllers after several design iterations. Another, and more relevant, example concerns whether the redundant channels of a flight-control system should be synchronized or not. Asynchronous designs seem to contain an intuitively robust source of fault tolerance in that the separate channels work on slightly different data (due to sampling skew). Nonetheless, several systems of this kind have revealed anomalies in flight-test and have proved hard to validate (I describe the AFTI-F16 flight tests in some detail in Section 3.3). On the other hand, the alternative, synchronized designs require extremely sophisticated treatment of their mechanisms for clock synchronization and sensor-value distribution. These mechanisms need to be "Byzantine fault tolerant" [LSP82], and any faults in the design or implementation of these mechanisms will almost certainly lead to system failure. However, if the difficulties of assuring the underlying mechanisms can be solved, the behavior of synchronized systems is very predictable, with none of the quirks of the asynchronous approach, and with reduced need for costly and fault-prone failure modes and effects analysis (FMEA) [HL91] and fault injections.[20] The balance of advantage seems to favor the synchronized approach, but the delicacy, and criticality, of the implementations of the necessary Byzantine fault-tolerant algorithms is such that this desirable area of the design space may be unsafe to enter without very strong assurance for the correctness of the underlying mechanisms.

Rapid prototypes and even "all-up" system simulations cannot provide the test coverage required to validate complex, parallel, timing- and fault-dependent behavior of these kinds (and, in any case, often distract attention from the real problem into lower-level implementation questions). Formal methods, on the other hand, allow the significant issues to abstracted out so that all behaviors of the chosen design can be considered. Formal methods based on state exploration can often examine all the possible behaviors of simplified instances of these kinds of design problems automatically. The complete coverage of simplified designs that can be achieved in this way is generally more effective at finding faults than partial coverage of the full design using conventional testing or simulation. In other words, formal methods such as state exploration can be an effective (often the most effective) *debugging* technique for these kinds of design problems. Conventional formal verification can then be used to verify the final, debugged design.[21]

Another example where formal methods may expand the design space concerns the selection of static or dynamic (priority-driven) scheduling for real-time systems. Some assert that only static, pre-allocated task schedules can provide the guaranteed satisfaction of hard-real-time constraints required for life-critical applications [XP91]. On the other hand, dynamic schedules degrade more gracefully than static ones under transient overload, and also simplify some software engineering problems [Loc92]. Dynamic scheduling can be seen as a high-risk area in the design space for life-critical systems, with a possibly high-payoff if adequate guarantees could be provided that all hard deadlines will be met.[22] Many papers have analyzed general circumstances under which dynamically scheduled systems will meet their deadlines (e.g., rate-monotonic scheduling can do this if the workload is less than 69% [SR90]), but the characteristics of individual flight-control systems do not always satisfy the necessary conditions (and fault tolerance can complicate

[20] However, additional redundancy is needed to withstand Byzantine faults: at least four channels are required to withstand a single Byzantine fault, and additional message passing is also required.

[21] It will be prudent to consider the higher levels of rigor here: at least one published algorithm, furnished with a Level 1 "proof of correctness," has been found to contain an outright bug [LR93b]. The bug was found and corrected using Level 3 formal verification.

[22] Tomayko [Tom87, page 112] describes a debate over the scheduling philosophy of the flight-control system for the Space Shuttle: "Rockwell ...argued for 2 years about the nature of the operating system, calling for a strict time-sliced system ...IBM, at NASA's urging, countered with a priority-interrupt-driven system similar to the one on Apollo. Rockwell, experienced with time-sliced systems, fought this from 1973 to 1975, convinced it would never work."

matters [HS91]). Individualized mathematical analysis, including formal methods, therefore seem essential if this part of the design space is to be explored safely.

2.2.1.3 Discussion of Formal Methods in the Lifecycle

There are two aspects to using formal methods: the "formal" aspect, and the "method" aspect. The former indicates a commitment to using concepts and techniques derived from mathematical logic, the latter identifies the particular way in which those concepts and techniques are to be used. In the present state of development, many tools and techniques for formal methods are fairly strong on the formal aspect, but weak on the methodological aspect. This means that while a particular specification language or verification system may, in skilled hands, be capable of effective application across a range of methods, less experienced users may be bewildered by the range of possibilities, and disconcerted to discover that finding the best method for a given problem often requires significant intellectual effort and invention. In the absence of methodological guidance, users may drift towards those uses of formal methods that seem most straightforward or best documented, rather than those that might deliver greatest benefit.

By and large, it is the late-lifecycle applications of formal methods that are the easiest to understand and apply: formalized descriptions in the guise of pseudocode, dataflow diagrams, or register-transfer level descriptions are already familiar at this stage, and the transition to fully formal descriptions and analysis is reasonably straightforward. In addition, there is a body of literature to provide encouragement and advice. 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). Consequently, the later stages of the lifecycle generally seem to offer the least return for the investment in formal methods.

The earlier stages of the lifecycle seem to offer rather greater potential benefits for using formal methods, but also pose greater methodological challenges. There are two main applications for formal methods in the early lifecycle: *descriptive* purposes, such as documenting requirements and specifying interfaces, and *analytic* purposes, such as validating major design decisions and crucial algorithms. The first of these can be served by all levels of rigor; the second is best served by Level 3. In all cases, a simple sequential model of computation is seldom appropriate, and it will be necessary either to select a formal method supplied with an appropriate model of distributed or parallel computation (e.g., one based on temporal logic, or on process algebra), or to find some way to represent such a model within a neutral formalism such as higher-order logic.[23] For analytic purposes, skillful use of abstraction will generally be required in order to focus on the substance of the problem without the distraction of extraneous detail. The selection or invention of an appropriate model of computation, its representation in the chosen formal system, and the abstraction of the relevant from the irrelevant, are all very challenging tasks that require talent and training.

Precisely because the early stages of the lifecycle are known to be unstructured and fault-prone, many informal or quasi-formal software engineering methods have been proposed to alleviate these difficulties. The more recent ones generally focus on object-oriented techniques (e.g., OMT [RBP+91]). Many current projects are using or evaluating these new methods; those that are also interested in formal methods face the rather formidable challenge of integrating these two classes of methods, which have developed rather separately. It will probably be some years before the formal element of formal methods is smoothly harnessed to the methodological element of modern software engineering methods.

I offer the tentative conclusion that the benefits of formal methods, given the current state of technology, increase with the intellectual difficulty of their application: routine exercises are likely to provide little benefit over established practice; greater challenges may provide larger returns. This conclusion may appear discouraging, but consider that the tougher intellectual challenges may be much smaller in scale than the routine exercises, thereby greatly improving their benefit to cost ratio. In general, aggressive and skillful use of abstraction and selection are the most effective techniques for reducing the cost and increasing the effectiveness of formal methods.

2.2.2 Formal Methods and System Components

We can expect it to be more difficult and costly to develop and to provide assurance for very critical software than for that which is less critical; and we can also expect a large piece of software to be more of a challenge than a smaller one. Therefore, design techniques that help reduce the criticality level of software components, or that reduce the size of the most critical components, can make important contributions to the development of critical systems. Criticality levels for software are determined by considering the potential severity of the effects of its failure. The FAA ranks

[23] By employing abstraction effectively, important properties of distributed systems can often be analyzed without requiring an explicit model of distributed or parallel computation. For example, Byzantine agreement protocols are conceived as distributed algorithms, but the correctness properties of these protocols can be adequately studied by representing them as mathematical functions [LR93a].

these effects on a five-level scale from "catastrophic" through "hazardous/severe-major," "major" and "minor" to "no effect" [FAA88][24]. DO-178B then 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]. Other standards and guidance for safety-critical systems take a similar approach (e.g., the United Kingdom Defence Standard 00-56, which also factors likelihood into the classification [MOD91b, Section 6]). The software criticality level determines the amount of effort and evidence required to show compliance with certification requirements.

It is sometimes possible to reduce the criticality level of some software components by raising that of others. This can be very worthwhile if the software whose level is lowered is much larger or more complex than that which is raised, or if its criticality is lowered by several levels. DO-178B discusses three such approaches under the heading "system architectural considerations" [RTCA92, Section 2.3]: partitioning, multiple-version dissimilar software, and safety monitoring.

Partitioning refers to techniques that provide fault containment. A software component that performs a function of low intrinsic criticality may be assigned a very high criticality level if it shares resources with (e.g., runs on the same processor as) a truly critical component, since a bug in the first component could interfere with the operation of the second (see [Add91] for a case study). If a really solid fault partitioning mechanism could be provided, so that malfunction of the first component could not affect the second, then the criticality of the first component could revert to its intrinsic level. The most drastic partitioning mechanisms are physical: we simply do not allow critical and noncritical functions to share resources.[25] This approach has its own problems (the need to maintain multiple computer systems and to arrange for their coordination) and becomes less effective as more communication between subsystems is introduced.

An alternative approach is to provide logical partitioning based on the use of memory-management units to enforce isolation of address spaces. An operating system nucleus is then needed to manage these hardware mechanisms. The criticality level of the operating system nucleus that manages the partitioning mechanisms will be at least that of the highest criticality application task that it supports. An operating system nucleus of this kind has much in common with the "security kernel" concept [DoD85] (or, rather, its more primitive foundation, the "separation

[24] Actually, [FAA88, paragraph 6.h] has "catastrophic," "major" (which is subdivided into categories (i) and (ii)), and "minor"; the addition of "no effect" and the naming of the subdivisions of the "major" category are from DO-178B [RTCA92, Subsection 2.2.1].

[25] This is the approach that has been taken, in a very strict form, on NASA interplanetary spacecraft, where functions critical to the survival of the spacecraft are handled by an attitude and articulation control system that is quite separate from the systems that control the scientific experiments. The consequences of the absence of such strict fault containment are well-illustrated by the failures of the Russian Phobos spacecraft [Che89, Coo90].

kernel" [Rus81, Rus84]) used in computer security. Formal specification and verification techniques have been developed [Rus82] and applied [Bev89] for the "secure separation" property, which seems similar to the fault-partitioning property required for safety-critical applications. Even without specialized hardware such as memory management units, it is possible to provide strong assurance of partitioning based on software techniques. Helps [Hel86] describes such techniques applied to an aircraft electronic flight instrumentation system.

When separate functions and computers are integrated into a larger system, it becomes necessary to extend fault partitioning from the processes of a single machine to the distributed ensemble. Combinations of hardware and software can provide the necessary partitioning on and across communication channels (see, for example, the SAFEbus™design for aircraft busses [HDHR91, HD93]).

I do not consider *dissimilarity* (multiple-version dissimilar software) here, since it is orthogonal to formal methods, although I do touch on some of the assurance issues in Section 3.1. The general topic of fault-tolerant software is covered in a separate chapter of the FAA Handbook [FAA-F].

The idea of *safety monitoring* is to backup a primary system with a monitor component that can detect and respond to undesired and potentially hazardous conditions. If its response to a hazard is to shut the primary system down, then monitoring is mainly a protection against unintended function, and its activation results in loss of function. DO-178B [RTCA92, Subsection 2.3.3] suggests that

> "Through the use of monitoring techniques, the software level of the monitored function may be reduced to the level associated with the loss of its related system function."

An early draft of DO-178B explained the rationale in a little more detail:

> "If a specific unintended function of the software can produce a more severe consequence to the aircraft than the loss of intended functions, it may be possible to reduce the software level through the use of monitoring...This can be accomplished by directly monitoring the unintended function that would produce the hazard, and providing an acceptable response that removes or reduces the hazard."

Of course, it is essential that the monitor and its protective mechanisms are independent of the primary system, so that both are not rendered inoperative by the same failure condition. DO-178B [RTCA92, Subsection 2.3.3] states that

> "Safety monitoring software is assigned the software level associated with the most severe failure condition category for the monitored function."

Some safety-critical applications (e.g., nuclear power generation) use monitoring and shutdown as their protection mechanism against catastrophic failures. In these cases the main safety-critical requirement on the primary control system is to reduce demands on the shutdown system to a fairly low level. This allows the acceptable failure rate per demand on the protection system to be quite modest (e.g., around 10^{-4} or 10^{-5}). The United Kingdom Defence Standard 00-56 gives a table [MOD91b, Table 6] that assigns criticality levels to monitoring systems based on the combination of accident severity and probability of failure of the primary component. It also assigns safety integrity levels (similar to DO-178B software levels) to combinations of components of lesser levels [MOD91b, Table 7] and takes care to block recursion (so that a critical function may be implemented by certain combinations that include components of less than maximum integrity level, but those components cannot themselves be implemented by combinations of components having even lower integrity level). It also establishes "claim limits" [MOD91b, Table 8] on the minimum failure rate that can be claimed for given integrity levels.

There are other monitor-like functions that can be performed by software: examples include analogs of the protections provided by hardware interlocks, lockins, and lockouts [Lev86]. Some of these can be enforced fairly generically by a kernel-like operating system nucleus [Rus89]; others will be more application-specific [LSST83].

In addition to the three system architectural considerations discussed in DO-178B (partitioning, dissimilarity, and safety monitoring), a fourth seems worthy of examination: namely, centralized redundancy management. Redundancy management and fault tolerance are among the major challenges in the design of airborne systems. Instead of a single computer executing the software, there will generally be several, which must coordinate and vote (or average) actuator commands and other outputs in order to tolerate hardware faults. Sensors and actuators will usually be replicated also, and the management of all this redundancy and replication adds considerable complexity to both the operating system (generally called an "executive" in control systems) and the application tasks. Complexity is a source of design faults, and there is a distinct possibility that such a large quantity of additional code may lessen, rather than enhance, overall reliability. There is evidence (recounted in Section 3.3) that redundancy management is sufficiently complex and difficult that it can become the *primary* source of *un*reliability in a flight-control system.

There is a degree of design freedom in how much of the redundancy management should be performed by the executive, and how much by the application tasks. The arguments in favor of centralizing as much of the redundancy management as possible (presumably in the executive) are that (1) it can then be given special attention and be done correctly, once and for all, and (2) by relieving applications tasks of this responsibility, those tasks become much more straightforward and therefore easier to get right and to validate. Arguments against centralized redundancy management are that some recovery procedures are application-specific, and that the centralized function is a potential single-point failure. It seems clear that a centralized redundancy management function will likely inherit the criticality level of the most critical task that it supports; centralized redundancy management is unlikely to reduce the criticality level of any application tasks, but may significantly reduce their complexity. Most modern architectures for fault-tolerant systems do provide a number of centralized redundancy management services [DH90, HL91, KWFT88, K$^+$89, WLG$^+$78], and it is recognized that providing assurance for these services is a significant challenge. Discussing a design that subsequently became the primary flight computer (PFC) of the Boeing 777, Dennis and Hills [DH90] noted that 50% of the software was concerned with redundancy management and observed:

> "Since conventional test procedures cannot cover all aspects of the redundancy management design, new validation and verification procedures must be devised to facilitate design proving and hence certification. These are expected to encompass formal mathematical proof of the 'core' redundancy management function..."

In all these "system architectural considerations," we see that by careful design it is often possible to divide a system into software components that have different criticality levels. Ideally, there should be relatively few at the highest levels, and those should be as small and straightforward as feasible. This not only makes it more plausible that those most critical components will perform as required, it also makes for more thorough and credible assurance that they will do so. In particular, it may be appropriate and feasible to apply formal methods of the higher levels to the most critical components. Investment here may have a high payoff. For example, an operating system nucleus providing fault partitioning is a very sophisticated (though not necessarily very large) component, whose failure (either to provide service, or to enforce partitioning) would be catastrophic. It is scarcely credible that adequate assurance for such a nucleus could be achieved without some use of formal methods. Yet the availability of such a nucleus could drastically reduce the criticality levels of other software (by eliminating fault propagation) and could also encourage more modular system design.

2.2.3 Formal Methods and System Properties

Formal methods are often associated with "proofs of correctness": the assumption being that it is the standard, or intended, function of the component concerned that is of most interest. But in safety-critical systems, the standard function of a component may not be the property of critical concern—that is to say, loss of that function may not be an event of high criticality. Rather, it may be certain malfunctions, or possible unintended functions, of the component that have safety-critical implications.

As noted above, architectural mechanisms, such as partitioning and safety monitoring can protect against some malfunctions and unintended functions, but an alternative is to ensure that the component will not exhibit specific malfunctions or unintended functions. Formal methods can play a role here: instead of proving that a component exhibits certain desired properties, we prove instead that it does *not* exhibit certain *undesired* properties.

One example of a technique based on this idea is "Software Fault Tree Analysis" (SFTA). SFTA was introduced by Leveson and others [CLS88, LH83] as an adaptation to software of the traditional fault tree analysis technique developed for electromechanical systems in the 1960's. The goal of SFTA is to show that a specific software design will not produce system safety failures or, failing that, to determine the environmental conditions that could lead it to cause such a failure. The basic procedure is to suppose that the software has caused a condition which hazard analysis has determined to have unacceptable risk, and then to work backward to determine the set of possible causes for the condition to occur, and so on, recursively. Eventually, this will lead either to discovery of circumstances that can cause the hypothesized condition to occur, or to a demonstration that there are no such circumstances.

A fault tree is developed incrementally, starting from a root, which is the undesired condition to be analyzed. Necessary preconditions are described at the next level of the tree with either an AND or an OR relationship. Each subnode is expanded in a similar fashion until all leaves describe events of calculable probability or are incapable of further analysis for some reason. Hardware and software fault trees can be linked together at their interfaces to allow the larger system to be analyzed. This is important since a particular software error may cause a mishap only if there is a simultaneous hardware and/or human failure, or other unexpected stress placed on the software. SFTA can be used at various levels and stages of software development, including the program code level. When working at the code level, the starting place for analysis is the code responsible for output. The analysis then proceeds backward deducing both how the program got to this part of the code and the necessary values of the program variables.

An experimental application of SFTA to the flight and telemetry control system of a spacecraft is described by Leveson and Harvey [LH83]. They report that the analysis of a program consisting of over 1,250 lines of Intel 8080 assembly code took two days and discovered a failure scenario (sensors detecting two sun pulses within 64 ms. of each other) that could have resulted in the destruction of the spacecraft. Conventional testing performed by an independent group had failed to discover this problem. In this example, the fault tree analysis was done by hand in an informal manner. Leveson and her co-workers have also developed fault tree templates for common Ada constructs and some tools to help in their construction and application [CLS88]. They describe a tutorial example in which their approach is used to identify a subtle timing problem in an Ada program for a traffic light controller.

Leveson attributes the success of SFTA in finding errors undiscovered by other techniques to the fact that it forces the analyst to examine the program from a different perspective than that used in development; she likens it to vacuuming a rug in two directions: conventional approaches "brush the pile" in one direction, SFTA in the other, so that between them they do a better job than either used alone.

Another problem domain where the focus has chiefly been on "negative" properties (i.e., on what should *not* happen) is computer security. Here the concern has been to show that mechanisms built into the lowest levels of the operating system guarantee that no communication of information contrary to the security policy is possible, no matter what "untrusted" application programs may do. The techniques employed are chiefly those of "information flow analysis" [DD77b, FLR77] and verification of access control mechanisms [BL76, WKP80].

Great caution must be exercised when formal treatments of negative system properties are combined with those for hierarchical verification. The difficulty is that hierarchical verification typically establishes only that each layer in the hierarchy is sufficient to implement the layer above it; it does not establish necessity (technically, an implication is established, not equivalence). That is to say, hierarchical verification ensures that each layer does *at least* that which is required to support the layer above, but there is nothing to stop it doing *more* than is desired. Since negative properties are largely concerned with what is *not* to be done, conventional hierarchical verification does not guarantee that negative properties proved of a specification will be preserved in its implementation; instead, it seems that these properties must be verified directly at the implementation level. Fortunately, it seems that SFTA, information flow analysis, and similar techniques are sufficiently straightforward that they can be applied effectively at that level.

Even when the critical property of a component is a "positive" one, it need not be equivalent to its full functionality. By focusing on just the critical properties, it may be feasible to apply formal methods of the higher levels at reasonable cost. For example, a component may be designed to provide an optimal solution to some problem, whereas safety is assured by any solution within certain bounds. It may be much simpler to verify that the component produces an adequate solution than that it produces an optimal one. This approach can be combined with certain software fault tolerance techniques to yield formally verified components in which only very simple subcomponents are verified [AW78].

Suppose a component that computes integer square roots is desired. A program based on the Newton-Raphson method may be the most efficient, but may be considered hard to verify, whereas the linear-search method is inefficient, but easy to verify. We can try to get the best of both worlds by constructing a compound unit based on the "recovery block" approach to software fault tolerance [AL90]. The idea behind recovery blocks is that a "primary" and several "alternate" functions are provided, together with an executable "acceptance test" for evaluating their results. The primary is executed first and the acceptance test is applied to its result. If the result passes the test, the block terminates with that result; if not, the system state is rolled back to that prior to execution of the primary, and an alternate is tried. As soon as an alternate yields a satisfactory result, the recovery block terminates with that result; if no satisfactory result is found, an exception is signalled. An experiment by Anderson [ABHM85] demonstrated some effectiveness for recovery blocks (70% of software failures were eliminated, and MTBF was increased by 135%), but found that acceptance tests are hard to write. The way in which formal methods can be factored into this approach is that we prove that the acceptance test is sufficiently strong to ensure satisfaction of the properties of interest, and we prove that one (presumably the simplest) of the alternates will satisfy the acceptance test. For our square root example, we would use Newton-Raphson in the primary, linear search in the alternate, and an acceptance test that simply checks that the square of the answer is close to the number supplied as input. There are obvious drawbacks to this proposal: it requires the state-restoration machinery of backwards error recovery, and it makes prediction of execution time difficult.[26]

Even without the mechanisms of recovery blocks, it may be useful to factor run-time checks into a formal verification: instead of proving that a certain property holds at a certain point, we can simply test it at run time. The argument for formal verification can then be based on the tested properties.[27] The Anna system incorporates ideas similar to these [Luc90]. Of course, it is also necessary to deal appropriately with cases where the run time checks fail. Formal treatments of exception handling focus on these kinds of problems [Cri84, Cri89].

[26] However, some techniques based on recovery blocks are designed to guarantee satisfaction of real-time constraints [CHB79, WHCC80, LC86]. Basically, a task is allowed to run until some fixed time before its deadline; then, if it has not produced a result, it is aborted and control is given to some (presumably) less-desirable task that is guaranteed to produce an acceptable result in the time remaining.

[27] Without the intellectual framework provided by formal verification, run-time checks may be of limited effectiveness. In one study [CLSK87], 24 students were hired to add self-checks to programs containing a total of 60 known faults (there were 8 different programs, each was given to three students). Only 9 out the 24 self-checking programs detected any faults at all; those that did find faults found only 6 of the 60 known faults, but they also discovered 6 previously unknown faults (in programs which had already been subjected to one million test-cases). Sadly, 22 new faults were introduced into the programs in the process of adding the self-checks.

The criterion for applying formal methods to a system or component property need not be criticality alone: the effectiveness of other methods for debugging and assurance should also be considered. For example, a component may be required to perform some critical function, and to be fault tolerant. It may be that traditional methods of assurance are effective for establishing that the function is performed correctly, but that assurance for its fault tolerance is more difficult to provide. In such a case, it might be appropriate to abstract away from the function performed and to apply formal methods only to the property of fault tolerance. This may provide an additional benefit: several components or functions may use similar fault-tolerance mechanisms, so that a single, suitably generic, use of formal methods may increase assurance for all of them.

Aggressive use of abstraction, and focus on particular properties rather than general functionality is a hallmark of formal methods based on state exploration. As I have noted earlier, but explain here in more detail, these precepts can often be pushed very far in the early lifecycle and in cases where the concern is debugging rather than assurance.

The idea is that *complete* analysis of selected properties of a highly simplified version of a design can be more effective at detecting errors than partial coverage of more general properties, or of a more realistic design. 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 many purposes, it may be adequate to examine the protocol with just one or two different data values, and with sequence numbers restricted to 0 and 1. And initially, we might not consider reliable transmission, but only examine whether deadlock is possible. Similarly, cache coherence protocols can be debugged quite effectively by considering just two processors and a single memory location [DDHY92], and the bugs in Byzantine agreement protocols that my colleagues and I have studied can all be discovered in highly simplified versions of the problem (one round, 6 or fewer processors, and four or fewer distinct data values).

The advantage of reducing problems to these very simplified forms is that they may then become amenable to state-exploration methods. These are specialized tools for formal methods that systematically enumerate all the states of a finite-state algorithm and test whether certain predicates hold at those states. Recent techniques allow large numbers (i.e., millions) of states to be handled in an efficient manner.[28] Several systems based on state exploration are available; some of these exploit the close connection between finite state graphs and propositional temporal logic (when they are usually called "model checkers" [CES86]), some provide a higher-level language (e.g., Murϕ [MD93] uses a transition-rule language for concurrent systems that is loosely based on Chandy and Misra's Unity model [CM88]), and

[28] These techniques include hashing [Hol91], and symbolic methods using Binary Decision Diagrams [Bry86, BCM+92].

others are based on ω-automata and language containment [Kur90]. Gupta [Gup92] provides a good survey (though specialized to hardware applications).

Although state-exploration tools can handle large numbers of states, the presence of a single 32-bit register in a design can introduce far more states than it is feasible to enumerate (and a mathematical integer in a more abstract specification will introduce an infinite state space). Hence the need to explicitly simplify or "downscale" many of the problems that are submitted to analysis by state exploration. But note that analysis of the downscaled version will be complete: if there is any circumstance under which the downscaled design fails to satisfy the selected properties, state exploration will detect it, and will often be able to provide diagnostic information that helps identify the source of the fault. Although its applications are somewhat limited in scope and scale, state exploration has the great advantage over formal analysis based on conventional theorem proving that it can be completely automated, and its use requires far less skill and training. The problems for which state-exploration methods are best suited are those involving asynchronous parallel activity, which are also those most difficult to debug using testing. æ

2.3 Validation of Formal Specifications

> "*It is a great advantage for a system of philosophy to be substantially true.*" [George Santayana]

Making a specification formal does not necessarily make it true. In this section I consider ways of gaining confidence that a formal specification says what is intended and what is true. These provide the technical foundation for many of the benefits ascribed to formal methods in the following section, and are also the methods for counteracting most of the fallibilities of formal methods that are identified in the section after that.

From personal experience and observation, it seems that writing formal specifications is at least as difficult as writing good informal specifications, or good programs. The most common failing is suggesting an implementation, rather than specifying simply *what* is required. To a large extent, however, judgments such as these are of only aesthetic or economic significance: they are concerned with how useful a specification may be, not with whether or not it is *right*. This question of whether a specification is right—whether it says what is intended (or what should have been intended!)—is answered through the process of validation. I identify two components in this process: first, seeking assurance that the specification is internally consistent and that it means *something*; second, seeking assurance that it means what is *intended*.

2.3.1 Internal Consistency

A specification implicitly says much more than is explicitly written down; if we believe a specification, we must also believe all the consequences that follow from it by rational deduction. A *formal* specification is a collection of axioms and definitions in some formal system; its consequences are all those formulas that can be derived (i.e., proved) by formal deduction from those axioms and definitions. That set of formulas is the *theory* defined by the specification. If its theory contains some formula and its negation (i.e., both A and $\neg A$ for some A) then a specification is *inconsistent*. By the usual rules of logic, it is easy to see that an inconsistent specification contains not only A and $\neg A$ for *some* formula A, but for *all* formulas.[29] Thus an inconsistent specification does not constrain its theory at all, and therefore fails to say anything useful.

There are two main ways to ensure that a specification is consistent: one is to show that it has a *model*. This notion is explained technically in Appendix Sections A.2, A.3, and A.4, but for the time being we can think of it as meaning that the specification can be implemented. The other way to ensure consistency is by restricting the specification to forms that guarantee what is called *conservative extension*. I will examine these two approaches shortly. First, though, I consider methods for ensuring that a specification is free of various obvious mistakes that may harbor a genuine inconsistency, or may be less damaging, but are in any case best excluded.

2.3.1.1 Strong Typechecking

The basic way to catch mistakes in a specification is through redundancy: if we see a function used with three arguments in one place, but only two in another, then one of them is probably a mistake.[30] If we require the user to declare the function and its number of arguments ahead of its use, then we increase the degree of redundancy available, and hence the likelihood of detecting this kind of mistake. A stronger form of the same idea is to allocate a *type* or *sort* to each entity that distinguishes the kinds of values it can take.[31] Computer scientists are familiar with this idea from programming languages, where variables may be declared as `real`, `integer`, `boolean` and so on. They are also familiar with the idea that

[29] In ordinary logics, the proposition $(A \wedge \neg A) \supset B$ is valid for any A and B. If we have an inconsistency for some A (i.e., both A and $\neg A$ are in the theory), then the rule of inference called *modus ponens* allows us to conclude that B is valid, for any B.

[30] However, some systems admit *polymorphic* functions that can take different numbers and types of arguments.

[31] See Appendix Section A.9 for an introduction to the technical issues.

it is generally not a good idea to multiply a real by a boolean,[32] and that such faults are generally detected at compile time. Specification languages can have much richer type-systems than programming languages (since the types do not have to be represented or implemented directly) and those that have any mechanized support at all are generally provided with a *typechecker*, which is a program, not unlike the front-end of a compiler, which checks that all uses of entities match their declarations, and that entities are combined only in appropriate ways.

The strength (i.e., fault-detecting capability) of typechecking depends partly on the logical foundation that underlies the specification language, and partly on the diligence of the implementors of the typechecker. It is difficult, for example to provide really strict typechecking for languages, such as Z and VDM, that are based on set theory. This is because everything in these systems is ultimately a set, and the type-system has to be grafted onto an essentially untyped foundation. Thus functions *are* sets of pairs in Z, and can take part in set operations such as union, intersection and so on—there is nothing fundamentally contrary to the principles of set theory in taking the union of, say, a function and the set of natural numbers. It is highly unlikely that such a construction would be something the user intended, but only slightly less outré constructions (for example the union of two functions) can be used to rather good effect in skilled hands. Thus, it can be difficult to provide a really thorough typechecker for languages based on set theory without sacrificing some of the flexibility that is considered among the chief advantages of such a foundation.

Type theory, or higher-order logic, which is the chief rival to set theory as a foundation for specification languages, is, as its name suggests, inherently a typed system. Thus, strict typechecking is natural for languages based on this foundation.[33] The price paid is a slight loss of flexibility compared with set theory, but the loss has not been found significant in practice and is more than offset by the benefit of early error detection and the clarity of expression fostered by a typed system.

Given the utility of typechecking in detecting faults and inconsistencies, it is natural that some specification languages should seek to exploit this utility still further by enriching the type-system. The basic idea is to allow much of the specification to be embedded in the types; strict typechecking can then perform a very searching scrutiny of the specification. For example, a very common technique in the language Z is to constrain a function to be injective (one-to-one) or surjective (onto). If a construction is later specified for the values of such a function, it is up to the user to check that the construction satisfies the requirements of injectivity or

---

[32] A further embellishment refines the notion of type with that of *dimension*; thus, real numbers representing time can be distinguished from those representing distance, and operations on inappropriate dimensions (e.g., the attempt to add a number representing distance to one representing time) can be flagged as an error [CG88, Hil88].

[33] In fact, it is *required* in order to avoid antinomies such as Russell's paradox.

surjectivity. But in a specification language with a richer type-system that includes predicate subtypes,[34] the injections and surjections can be specified as subtypes of the functions on the same signature, and typechecking a constructive definition for such a function will *require* a demonstration that it satisfies the predicate that specifies injectivity or surjectivity as appropriate. In the language of PVS [ORS92], for example, the specification fragment

$$\mathit{injection}\colon \mathbf{type} = \{f\colon [t_1 \rightarrow t_2] \mid \forall (i, j\colon t_1)\colon f(i) = f(j) \supset i = j\}$$

specifies *injection* as the subtype of functions from t_1 to t_2 (t_1 and t_2 are type variables) that satisfy the one-to-one (injective) property. If we were later to specify the function *square* as an injection from the integers to the naturals by the declaration

$$\mathit{square}\colon \mathit{injection}[\mathit{int} \rightarrow \mathit{nat}] = \lambda(x\colon \mathit{int})\colon x \times x\colon \mathit{nat}$$ [35]

then the PVS typechecker would require us to show that the body of *square* satisfies the *injection* subtype predicate.[36] That is, it requires the proof obligation $i \times i = j \times j \supset i = j$ to be proved in order to establish that the *square* function is well-typed. Since this theorem is untrue (e.g., $2 \times 2 = (-2) \times (-2)$ but $2 \neq -2$), we are led to discover a fault in this specification. Notice that this approach requires theorem proving during typechecking, and therefore requires a highly integrated system to support it. This degree of sophistication is really limited to the upper reaches of Level 3 formal methods, but serves to illustrate some of the advantages of that level of application.

The pushdown stack encountered earlier provides another example of the power of a rich type-system. Recall that the specification given previously (page 26) made no mention of the empty stack. It is easy to introduce *empty* as a special stack, but then how do we ensure that this value is different from any that can be formed by pushing values onto a stack, and what do we do about *pop*(*empty*)? Using predicate subtypes, we can provide the following signatures for the stack operations (this example is also in the language of PVS).

---

[34] Predicates can be thought of as (and in some systems are) functions that return boolean values; those arguments for which the predicate is true define a *subtype* of the type of its arguments. Another kind of construction found in rich type-systems is the *dependent* type, in which the type of one part of a compound construction depends on the value of another—for example, the range type of the operator $m \operatorname{rem} n$ (the remainder when m is divided by n) is the subtype of the integers in the range $0 \ldots n-1$ (i.e., the type depends on the value of n).

[35] The λ construction is a way of defining functions; think of the variables in parentheses coming after the λ as the formal parameters to the function, and the expression after the colon as its definition.

[36] We would also be required to discharge the (true) proof obligation generated by the subtype predicate for *nat*: $\forall (x\colon \mathit{int})\colon x \times x \geq 0$.

stack: **type**
empty: *stack*
nonempty_stack: **type** = { *s: stack* | $s \neq$ *empty*}

push: [*elem*, *stack* $\rightarrow$ *nonempty_stack*]
pop: [*nonempty_stack* $\rightarrow$ *stack*]
top: [*nonempty_stack* $\rightarrow$ *elem*]

With these signatures, the expression *pop*(*empty*) is rejected during typechecking (because *pop* requires a *nonempty_stack* as its argument), and the theorem

$$push(e, s) \neq empty$$

is an immediate consequence of the type definitions. For the same reason, the construction

$$pop(push(y, s)) = s$$

is immediately seen to be type-correct. And the expression

$$pop(pop(push(x, push(y, s)))) = s, \tag{2.2}$$

which may not appear type-correct if only syntactic reasoning is used (the outermost *pop* requires a *nonempty_stack*, but is given the result of another *pop*—which is only known to be a *stack*), can be shown to be well-typed by proving the theorem

$$pop(push(x, push(y, s))) \neq empty,$$

which follows from the axioms about *pop* and *push* given earlier. Use of theorem proving in typechecking is perfectly reasonable for specification languages (where a powerful theorem prover may be assumed to be available in Level 3 applications) and it allows a very expressive notation to be provided relatively simply. In programming languages, on the other hand, typechecking is usually expected to be fast and deterministic. Unfortunately, many specification languages continue to use programming-language style typechecking and cannot deal with formulas such as (2.2) at all well. Simple typecheckers either regard the formula (2.2) as type-incorrect, or they require the *pop* function to be of type [*stack* $\rightarrow$ *stack*]—in which case, they are forced to regard the expression *pop*(*empty*) as well-typed and lose an opportunity for early detection of faults.

Yet another example of the utility of predicate subtypes arises when modeling a system by means of a state machine. Recall (from Subsection 2.2.1) that the basis of this method is first to identify the components of the system state; an invariant then specifies how the components of the system state are related, and operations are required to preserve this relation. It is, of course, possible to construct a special-purpose tool that will generate the associated proof obligations, but this tool will have only a single function. With predicate subtypes, on the other hand, the generation of the necessary proof obligations comes for free during typechecking: we use the invariant to induce a subtype on the type of states, and specify that each operation returns a value of that subtype.

An alternative to a strong type-system embedded in the specification language and enforced by its mechanized support system is for the user to supply type-predicates within an untyped system. For example, we could define a predicate *is_injection* and supply it whenever we wished to indicate that a function was intended to be one-to-one. For example, we might specify the *is_injection* predicate by

$$is\_injection(f) = \forall i, j\colon f(i) = f(j) \supset i = j$$

and then specify the *square* function axiomatically by

$$is\_injection(square) \wedge square(i) = i \times i. \tag{2.3}$$

The disadvantage of this approach is that it requires the type predicates to be stated repeatedly in the specification, and it requires the user to remember to do so: in effect, it puts the whole burden of typing and typechecking on the user. And it provides no automated help in the discovery of the inconsistency in the axiom (2.3). Some authors prefer this arrangement for its flexibility (e.g., Lamport [Lam91]), but for the most part these objections to strongly typed systems are overcome when typechecking is supported by theorem proving, instead of being the largely syntactic mechanism used for programming languages.

To summarize: strong typechecking is an extremely effective tool for detecting mistakes in specifications. It cannot guarantee to *eliminate* inconsistencies or other kinds of faults, but it is a very cost-effective way to detect a goodly proportion of them. Rich type-systems supported by theorem proving increase the effectiveness of typechecking, and the economy of the specification, by allowing much of the specification to be embedded in the types. In addition to supporting economy of expression and effective fault detection, strong typing also helps in the development of specifications: the types often suggest the kinds of expression that are required (rather like dimensional analysis in physics or valences in chemistry).

2.3.1.2 Definitional Principle

Earlier (in Subsection 2.3.1), I stated that a formal specification is basically a collection of axioms and definitions, but I did not explain the difference between an axiom and a definition. The difference is that whereas an axiom can assert an arbitrary property over arbitrary entities (new and old), a definition is a restricted kind of axiom that defines some new concept in terms of those already known. For example, in the theory of stacks, we have the notion of *popping* a stack, and of *pushing* a value onto a stack; the relationship between these is specified by the axiom

$$pop(push(s, e)) = s$$

which states that if you *push* the value e onto a stack s, and then *pop* the result, you end up with the stack you started with. This specification is an axiom: it is not introducing or defining the concepts *pop* or *push*, it is specifying a certain property of their interaction.

On the other hand, if we say

$$|x| = \textbf{if } x < 0 \textbf{ then } -x \textbf{ else } x \textbf{ endif}$$

and all the concepts on the right-hand side of the = are already known, then we have *defined* the absolute value function, $|x|$. Because they do not assert new properties of existing concepts, but rather define a new concept in terms of those already known, properly formulated definitions cannot introduce inconsistencies. "Properly formulated" means constructed according to a *definitional principle* of the specification language concerned; a definitional principle must ensure that definitions provide only what in logic is called a *conservative extension* to a theory.[37] Many specification languages have an associated definitional principle. Precisely what concepts can be defined in this way depends on the richness of the underlying logic, on the strength of the definitional principle, and on the power of its associated mechanization. For example, in many systems it can be rather difficult or impossible to specify by a definition the function *min* that returns the minimum value of a set. Instead, an axiom of the form

$$min(x) \in x \wedge (\forall y \colon y \in x \supset min(x) \leq y)$$

is usually required. In systems rich enough to provide Hilbert's ε operator [Lei69],[38] however, the function can be specified definitionally by

$$min(x) = \varepsilon z \colon z \in x \wedge (\forall y \colon y \in x \supset z \leq y)$$

The strength of a definitional principle also determines whether recursive definitions can be admitted. The difficulty with recursion is that it may not "terminate" for certain arguments (i.e., the function may be partial). For example,

$$nono(x) = \textbf{if } x = 1 \textbf{ then } 1 \textbf{ else } x + nono(x-1) \textbf{ endif}$$

[37] A theory A is an *extension* of a theory B if its language includes that of B and every theorem of B is also a theorem of A; A is a *conservative* extension of B if, in addition, every theorem of A that is in the language of B is also a theorem of B.

[38] $\varepsilon z \colon P(z)$ means "a z such that P holds for that z." If P is unsatisfiable, then an arbitrary z is chosen. The type of z must be nonempty.

is not well-defined for arguments of zero or less (nor for noninteger arguments). One way to extend a definitional principle to recursive definitions requires that some "measure" function of the arguments should decrease across recursive calls, and that the measure function should be bounded from below.[39] The example above becomes well-defined if, for example, x is required to be a strictly positive integer—in which case the identity function on the strictly positive integers serves as the measure; it decreases on recursive calls to *nono* (i.e., $x - 1 < x$), and cannot decrease without bound because it is specified to be strictly positive.[40]

This definitional principle ensures that recursively defined functions are total (i.e., defined for all values of their arguments). Some authors claim that this is too restrictive and that it is necessary to admit partial functions into the logic underlying a specification language [CJ90]. The standard "challenge" is the function *subp* on the integers defined by

$$subp(i, j) = \textbf{if } i = j \textbf{ then } 0 \textbf{ else } subp(i, j+1) + 1 \textbf{ endif}.$$

This function is undefined if $i < j$ (when $i \geq j$, $subp(i, j) = i - j$) and it is argued that if a specification language is to admit this type of definition, then it must provide a treatment for partial functions. Partial functions can greatly complicate a logic (see Section 2.6) and are inimical to efficient theorem proving. Fortunately, examples such as these do *not* require partial functions: they can be admitted as total functions on a very precisely specified domain. *Dependent types*, in which the *type* of one component of a structure depends on the *value* of another, are the key to this. For example, in the language of PVS, *subp* can be specified as follows.

$$\begin{array}{l} subp((i:\ int),\ (j:\ int \mid i \geq j)):\ \textbf{recursive}\ int = \\ \quad \textbf{if}\ i = j\ \textbf{then}\ 0\ \textbf{else}\ subp(i,\ j+1) + 1\ \textbf{endif} \\ \textbf{measure}\ \lambda\ (i:\ int),\ (j:\ int \mid i \geq j):\ i - j \end{array}$$[41]

[39] Another way to admit recursive definitions is to provide a fixed "template" that such definitions must follow. A meta-proof establishes that all correct instantiations of the template will be well-defined. This approach can be easier to implement than one involving measure functions (it does not require theorem proving during the analysis), but is more restrictive. The definitional mechanisms for abstract datatypes that are described later are essentially a sophisticated template approach for a fairly general class of definitions.

[40] The reader might wonder how the argument would go if the recursive call were $nono(x - 2)$: the same argument seems to work, yet it is clear the definition does not terminate if the outermost argument is an even number. The explanation is that *nono* is specified to take a positive integer as its argument but $x - 2$ does not typecheck as a positive integer (the typecheck proof obligation $x > 0 \wedge x \neq 1 \supset x - 2 > 0$ is false). Hence, the definition will be rejected. This example shows how delicately interdependent some aspects of typechecking can become in languages with rich type systems.

[41] The **measure** clause specifies a function to be used in the termination proof.

Here, the domain of *subp* is the dependent tuple-type

$$[i\colon int, \{j\colon int \mid i \geq j\}]$$

(i.e., ordered pairs of integers in which the first component is greater than or equal to the second) and the function is total on this domain.

Enforcing a definitional principle generally requires mechanical tool support (it can, in theory, be done by hand, but is tedious and error prone). Furthermore, a definitional principle for recursive functions generally requires theorem-proving capability in its enforcement mechanism. But once theorem-proving support is available, it then becomes possible to enrich the definitional principle still further. For example, introducing a constant of a subtype defined by a predicate can introduce an inconsistency if the predicate is unsatisfiable (i.e., if the subtype is empty). Such subtype definitions can be made conservative by a definitional principle that requires their defining predicates to be satisfiable—which in turn requires theorem proving to enforce. It follows that the most extensive definitional principles are associated with specification languages having extensive tool support; conversely, systems that have traditionally lacked tool support (for example Z) generally have no definitional principle associated with them ([Art91] describes an attempt to retrofit a principle of conservative extension to Z using a mechanization based on HOL).

The advantage of a definitional principle is that those concepts introduced by means of definitions cannot introduce inconsistencies.[42] But definitions are not always to be preferred to axiomatic specifications: definitions have a constructive character that can result in "overspecification"—suggesting too much about *how* a concept is to be realized, rather than simply stating *what* is required of it. For example, the axiomatic specification of a pushdown stack given earlier states only the essential properties required of a stack, and we are free to implement it later in any way that satisfies its axioms—with a list, for example, or with an array and a pointer. A definitional specification of a stack, on the other hand, would generally construct it on top of some more primitive concepts, such as an array or list, thereby suggesting an implementation.

By introducing a specialized definitional principle for abstract datatypes, it is sometimes possible to have the best of both worlds and to specify datatypes such as stack definitionally, yet without implementation bias. A datatype definition principle permits a very compact specification of the relations between the "constructors," "accessors," and "recognizers" of an abstract datatype; internally, the compact specification is expanded into a set of axioms whose consistency is assured by a meta-proof on the datatype definition principle. Not all abstract datatypes can

[42] Definitions are also conducive to efficient theorem proving (using a technique known as "rewriting"), and to the development of executable specifications.

be specified in this way, but many can. For example, in PVS, the stack specification given earlier (with the empty stack distinguished) can be specified as follows.

stack [*t*: **type**]: **datatype**
begin
empty: *empty_stack*?
push(*top: t, pop: stack*) : *nonempty_stack*?
end *stack*

Here *empty* and *push* are the constructors, *empty_stack?* and *nonempty_stack?* the recognizers (and the corresponding subtypes) for their respective constructors, and *top* and *pop* are the accessors for nonempty stacks. The whole specification is parameterized by the type t of elements to be pushed on the stack. These few lines of specification expand into a couple or so pages of type definitions, function signatures, the axioms shown previously, and several more axioms that provide an induction scheme and other useful constructs.

A datatype definition principle ensures consistency of an axiomatization by generating a highly stylized set of axioms of a form that has been shown, once and for all, to be consistent. Another approach allows the user to write axioms of a certain restricted form and then checks the axioms for properties that are sufficient to ensure consistency. Equational specifications provide an example of this approach. If all the axioms have the form of equations with the properties of "finite termination" and "confluence" [Hue80], then the axiomatization is consistent. Confluence can be tested using the Knuth-Bendix algorithm [KB70] (though this algorithm may not terminate if the equations are not confluent), and there are a variety of special-case methods for testing the finite termination property. Affirm [Mus80], and more recently Reve [Les86] and RRL [KZ88], support these techniques. An attractive side-effect of this approach is that it can provide a decision procedure for the theory concerned; its disadvantage is that it has rather limited application in practice.

I mentioned above that some specification languages have no definitional principle; conversely, certain systems for specification and verification have only a definitional principle and make no, or reluctant, provision for general axioms. (HOL is like this; its users are reminded [GM93, page 220] of Russell's aphorism likening the advantages of axiomatic postulation over definition to those of theft over honest toil.) In my opinion, neither of these extreme positions is ideal; a specification language should provide rich and powerful definitional principles (including, for example, a datatype definition mechanism), but should not exclude arbitrary axiomatizations. It should, however, provide some assistance in demonstrating the consistency of such axiomatizations, and that is the topic of the next subsection.

2.3.1.3 Exhibition of Models

A set of axioms must be consistent if the axioms are satisfied by some real object. When the real object is a mathematical structure, it is called a *model* for the axioms.[43] One way to show that a specification is consistent, therefore, is to show that its axioms have a model. This can be done fairly straightforwardly in semiformal (i.e., Level 1) applications, but it requires some sophistication to provide mechanical support. The difficulty for mechanization is that we cannot exhibit "real objects," only descriptions (i.e., specifications) of such objects. Rather than relate a specification to a model, we therefore have to relate two specifications. The appropriate way to do this uses the notion of *theory interpretations* [Sho67, Section 4.7]; the basic idea is to establish a translation from the types and constants of the "source" specification (the one to be shown consistent) to those of a "target" specification (that we already believe to be consistent) and to prove that the axioms of the source specification, when translated into the terms of the target specification, become provable theorems of that target specification. If this can be done, then we have demonstrated *relative* consistency: the source specification is consistent if the target specification is. Generally, the target specification is one that is specified definitionally, or one for which we have some other good reason to believe in its consistency.

The machinery of theory interpretations can also be used to demonstrate the correctness of hierarchical developments—that is, to show that a more concrete specification is a satisfactory implementation of a more abstract one. For example, the way we demonstrate that an array and a pointer can be used to implement a stack is by showing that a specification of such an implementation satisfies the abstract stack axioms.[44] The only difference between the use of theory interpretation to demonstrate correctness of an implementation and to demonstrate consistency of a specification is that for the latter, the "implementation" does not have to be useful, or realistic, or efficient; it just has to exist.

When a specification is describing something that is to be implemented, the demonstration of its consistency can be combined with that of the correctness of its implementation. Sometimes, however, a specification is describing assumptions about the world, or specifying an artifact that is to be built by others. In these cases, we will not be developing an implementation, so demonstration of consistency does not come "for free," but requires a deliberate decision to exhibit a model. For example, fault-tolerant clock synchronization is based on some assumptions about the behavior of "good clocks"—that they keep reasonably good time, and that one

[43] These notions are treated in a more technical fashion in Appendix Section A.2.

[44] An alternative way to demonstrate the correctness of an implementation uses the "abstraction" functions introduced by Hoare [Hoa72]; these are called "retrieve" functions in the VDM methodology and are discussed at length by Jones [Jon90].

processor can obtain a reasonably accurate estimate of the reading of the clock of another processor, and so on. These assumptions are captured in a number of axioms and we can demonstrate their consistency by constructing an interpretation in which clocks keep perfect time, and can be read with perfect accuracy. It does not matter that this interpretation is unrealistic: its only purpose is to demonstrate that the axioms are consistent. As I mentioned earlier, Friedrich von Henke and I did not do this for an early version of our verification of the Interactive Convergence Clock Synchronization Algorithm, and Young showed that our axiomatization excluded this interpretation [You92]. Our mistake (using $<$ in one axiom where $\leq$ is preferable) did not render the axiomatization inconsistent (though it could have done, since we had not explicitly checked them), but it excluded an intended model. Another specification that I developed [Rus86] was found to be genuinely inconsistent [KM86]. My experience is surely not unique, and should serve to demonstrate the value and importance of providing models to attest the consistency of axiomatic specifications. Even when demonstration of consistency can be achieved as part of a hierarchical development, it can sometimes be worthwhile to exhibit a very simple model (i.e., a totally unrealistic implementation) in order to gain early assurance of consistency before undertaking the full implementation.

The fact that we may be able to exhibit unrealistic interpretations for the purpose of demonstrating consistency suggests that a specification can have many interpretations (or implementations or models). Generally, this is desirable, for if a specification has only a single implementation, then it is surely a very restrictive specification—really more of an implementation. Since specifications do not, in general, characterize a single, unique implementation, most specification languages are said to use *loose* semantics: that is, any implementation that satisfies the specification is considered acceptable.[45] If an implementation has undesirable properties, then we should have given a more restrictive specification that will rule out implementations with those properties. There are, however, a few areas where restrictions are placed on the models assumed and implementations allowed. For example, in some treatments of specifications based on equational logic (i.e., specifications whose axioms are all equations), *initial* or (more rarely) *final* models are often assumed. And some treatments of computer security require that implementations be "faithful representations" (an undefined term) of their specifications.

Mechanized support for theory interpretations can be quite challenging and is generally found only at the upper reaches of Level 3 applications of formal methods. Among the details that must be dealt with are the need to translate the definitions and theorems of the source theory into the terms of the target theory, as well as the axioms, and the need to ensure that interpretations of equality maintain the

[45] This is consistent with ordinary logic, where a theorem is considered valid if and only if it is true of *all* models that satisfy the axioms.

properties assumed of an equality relation (alternatively, quotient types must be supported).

2.3.1.4 Modules and Parameters

It is generally convenient if specifications can be developed in fairly small, self-contained units which I will call *modules*. Just as with programs, specification modules allow related concepts to be grouped together into a larger conceptual unit and thereby support separation of concerns and reuse. A specification language must provide methods for combining modules in various ways to yield larger specifications. For this to be most effective, modules need to be parameterized, rather like the functions and procedures of a programming language, so that they can specify concepts with a certain genericity. For example, a module that specifies sorting should be parameterized by the type of thing that is to be sorted and by the ordering relation with respect to which the things are to be sorted.

In modern programming languages, it is necessary in a procedure declaration to specify the types of the parameters that are required: we should not be allowed to supply a string of characters to an arctangent procedure. In specification languages, it is necessary to do the same and more: it is often necessary to place constraints on the *properties* of module parameters. In a sorting module, for example, one of the parameters will be the type of thing to be sorted, represented by t, say, and the other will be an ordering relation. But it is not enough, as some specification languages do, to simply state the signature of the required relation (i.e., $t \times t$), it is also necessary to stipulate that it must be a total ordering relation (that is, it must have the properties of trichotomy, reflexivity, transitivity, and antisymmetry). Generally speaking, these properties may be *assumed* inside the module, but must be proved in any instantiation of the module. In the absence of these safeguards, it is possible to combine individually consistent modules into an inconsistent whole. As with some of the techniques of strong typechecking, these safeguards require theorem proving during specification analysis and are therefore often enforced only in Level 3 applications. Even if they cannot be mechanically enforced, it is important that assumptions on module parameters should be specified (and checked by hand) in applications of formal specifications at all levels. Regrettably, some specification languages that stress modularity and parameterization (e.g., Z) have overlooked this important aspect of soundness.

2.3.2 External Fidelity

I have considered methods of examining specifications for internal consistency—these are ways of making sure that a specification says *something* sensible; I now examine ways of scrutinizing specifications to make sure they say what is *intended*.

In the case of specifications of designs to implement other, more abstract specifications, we simply have to show that the design satisfies its superior specification. This is really a problem of verification and, as explained in the previous section, it is a problem that can be formalized and that is amenable to proof. The more challenging problems concern top-level, or *requirements* specifications, and statements of assumptions—for here there are no higher-level specifications against which to conduct a verification. The issue in these cases is truly one of validation. By their very nature, the problems of validating top-level specifications or statements of assumptions do not lend themselves to definitive proof; all that can be done is to examine the specifications from various perspectives, and to probe and test their consequences until we are satisfied that they adequately capture our requirements, or the real-world phenomenon they are intended to describe.

The first level of scrutiny that may be applied to a formal specification is informal human review: those with an understanding of the problem to be specified examine and discuss its formalization. Not all of those with understanding of the problem domain will be skilled in the reading of formal specifications. Consequently, it is usual for a specialist in the formalism concerned to be available to interpret and explain the specification to the domain experts. This, of course, interposes a layer of interpretation between the specification and those best suited to evaluate it, and so an often-expressed desideratum for formal specification languages is that they should be directly comprehensible to nonspecialists, so that this layer of interpretation is not required. For example, diagrammatic or tabular notations are familiar to engineers in many fields, and it is sometimes suggested that formal specifications should be expressed in these forms.

The problem with this approach is that while diagrams and tables may be convenient ways to explain certain types of specification, they do not lend themselves to particularly effective mechanized reasoning: they may lower the barriers to engineering review, but at the price of raising barriers to mechanically-supported analysis. Furthermore, different problem domains lend themselves to different types of diagrams or tables (e.g., compare the state-transition diagram for a controller, the timing diagram for a protocol, and the block diagram for a hardware design), and some do not lend themselves to diagrammatic presentation at all (e.g., an algorithm for clock synchronization). It follows that a formal specification method built around a particular diagrammatic or tabular notation may have rather restricted application, and limited mechanized support for general forms of analysis. On the other hand, it may be very convenient and amenable to effective review in those fields to which its notation is well-matched. A compromise arrangement that seems promising is to develop specifications in languages optimized for expressiveness and the effectiveness of their support for mechanized reasoning, but to provide tools that translate specifications satisfying certain restrictions into a variety of diagrammatic and tabular presentations. Such an arrangement may be compared to one in which differential equations (say) are used to specify mechanical systems, and in which graphs and tables can be computed to assist users' comprehension of the specified system.

Although I recommend that specification languages should primarily be designed for expressiveness and the effectiveness of the mechanical support that can be provided, rather than for the convenience they afford to nonspecialist reviewers, this does not mean that I advocate recondite methodologies or runic notations. For most purposes, specification languages based on classical logics (e.g., higher-order logic or first-order logic with set theory) permit fairly clear and direct expression and, while it takes skill and practice to write good specifications in these (or any) notations, it need not be hard to read and understand them. More ambitious foundations (e.g., constructive type theories) are generally difficult for nonspecialists to comprehend, while foundations based on weaker logics (e.g., unquantified notations such as equational logic, Horn clauses (e.g., Prolog), and the Computational Logic of Boyer and Moore [BM79, BM88]) often necessitate encodings that resemble programs rather than specifications. For example, to state that all components of a certain structure should possess a certain property, a computational logic may require specification of a recursive function that examines each component in turn, whereas a richer logic would allow a simple quantification (i.e., $(\forall x: \ldots)$). Not only is the recursion an indirect encoding of the intended property, it is likely that proofs involving the property will require induction rather than straightforward quantificational reasoning.

If the choice of the underlying logical formalism has a profound influence on the deep structure of specifications, so the notational conventions employed can have an effect on their surface appearance that may be no less profound in determining their readability and acceptability. The use of compact and familiar notations such as infix operators (i.e., $a+b$ rather than `(add a b)`), standard notation for function applications (i.e., $f(x, y)$ rather than `(f x y)`, $f\,x\,y$, or $f(x)(y)$), and computer-science constructions such as **if** ... **then** ... **else** ..., and **case** expressions, can all ease the task of comprehending formal specifications.

It is not necessary that specifications actually be processed by their support system in this form, as long as there are ways to communicate them to human reviewers in civilized notation. There is much to be said for providing specification languages with sophisticated prettyprinters that can typeset specifications into compact notation which matches that employed in the field concerned. PVS, for example, generates output under the control of simple user-written tables so that specification text such as

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

(which arises in analysis of clock synchronization) can be typeset as

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

which is instantly recognizable to those working in this field. Some systems actually manipulate this latter form as the specification text, rather than merely generate it as output. The disadvantages of this approach are that inputting non-ASCII symbols into the system can become complex—involving structure editors or mouse manipulations that rapidly become tedious—and revisions to the notation may require changing every reference to the symbols concerned.

In addition to prettyprinting and typesetting, other simple tools can materially assist in the comprehension and scrutiny of specifications. Cross-reference tables, for example, make it easier to navigate large specifications, and dynamic browsing tools that, for example, permit immediate location of the declaration of an identifier or of all axioms including a certain term, can provide even greater assistance.

Although, as described, much can be done to assist the comprehension of formal specifications, there is a limit to what can be achieved by inspection. More searching scrutiny requires testing or, as I shall say, *challenging* the specification. The idea of a challenge is to pose a question that an adequate specification should be able to answer. 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] report that early 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. Even with an adequate specification, however, we might find that our putative theorem is unprovable; further examination would help us understand why and might lead us 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.

Challenges such as this allow a specification to be explored in an active manner not unlike the testing of a program: in a way we are using theorem proving to "execute" the specification. With certain specification styles this notion can be taken further: the specifications truly can be executed. This is the case with Horn clause (Prolog) specifications, and those written in the form of quantifier-free equations or recursive functions.[46] Other executable specification languages resemble functional programming notations, and are sometimes called "prototyping" rather than specification languages [AJ90].

The problem with executable specifications is that they are often closer to programs than specifications; some of the benefits of formal specification have to be sacrificed in order to achieve executability [HJ89]. For example, directly executable specifications cannot describe the sorting operation simply in terms of its properties (i.e., that the output is an ordered permutation of the input), but must instead specify a particular sorting algorithm. One compromise approach that has some promise allows those parts of a specification that lend themselves to it to be executable directly, while other parts are "animated" by conventional program text supplied by the user or built in to the system [HI88]. Various degrees of rigor are possible here: for example, the animating program text could be proven correct with respect to the specification, or could simply be taken on trust as a "rapid prototype" [47] or, rather than conventional program text it could be an executable specification derived (and perhaps verified) from the original nonexecutable specification.

While execution of test cases can be a useful aid to validation, the more general challenges mentioned earlier may be more revealing and may yield greater insight since, like other formal verification methods, they examine general properties, not merely behavior on particular inputs. However, validation is assisted by examining the specification from as many viewpoints as possible, and both execution and challenges contribute to that end.

Still another viewpoint can be obtained by challenging a specification not with a putative theorem, but with a putative interpretation (which may be an executable implementation in some cases). Although property-oriented specifications are often to be preferred at the requirements level, our intuition may have an algorithmic slant (which is probably why inexperienced writers of formal specification often produce very constructive specifications: rather than simply assert the properties required, they prefer to suggest how to achieve those properties). We can exploit this intuition by checking whether certain implementations satisfy our specification or not. For example, we could reinforce our belief that we have specified sorting correctly by

[46] The execution mechanisms are specialized theorem-proving techniques: SLD resolution in the case of Prolog, and term rewriting in the other cases.

[47] Although a property-oriented specification may not be directly executable, it may yield an executable *test* that can be applied as a run-time check on the correctness of the animating code. For example, a property-oriented specification of sorting is not directly executable, but it could yield an executable test for the property that a sorted sequence is ordered (permutation is more difficult). The use of specifications to provide run-time checks on an implementation is a feature of the Anna system [LvHKBO87, Luc90].

verifying that a simple sorting algorithm is a correct implementation of the specification. For another example, recall that earlier I mentioned how an axiomatization of "good clocks" for a synchronization protocol could be shown consistent by exhibiting an interpretation in which clocks kept perfect time and could be read with zero error. This interpretation not only demonstrates consistency, it is surely an example (albeit extreme) of what good clocks should be, and so it is reassuring that it is indeed a model of the axioms. Sometimes we will want to do a negative test. For example, we might want to check that clocks which have stopped, or that have large read errors do *not* satisfy the axiomatization of "good clocks."

I have considered ways to examine specifications for universal properties (those that should be possessed by all specifications) such as consistency, and ways to gain confidence that a particular specification says what is intended. In between are a number of broad attributes that may be important for some classes of specifications but not others. For example, Jones [Jon90, Chapter 9] describes how to examine "model-oriented" (i.e., constructively defined) specifications for implementation bias (basically, a specification is biased if it distinguishes states that cannot be told apart using the operations provided). Another important notion that has different interpretations for different classes of specifications is that of *completeness*.[48] Informally, a specification can be considered complete if identifies every contingency and defines the behavior required under every possible execution scenario. When considering small components of a system, however, this broad notion of completeness needs to become more specialized.

When specifying an abstract datatype, for example, we may wonder whether the axioms provided for the set of operations are sufficient to adequately specify all behaviors. Guttag and Horning [GH78] define a property called *sufficient completeness* that captures this notion for a certain class of specifications, and they also give some heuristically effective procedures for checking this property. Kapur and Srivas [KS80] consider a slightly different problem: how to tell whether the operations provided for an abstract datatype are sufficiently comprehensive to support all likely uses of that datatype.

Jaffe and Leveson [JL89] (a somewhat different version appears as [JLHM91]) describe a rather different interpretation of completeness that arises in real-time process-control systems. They consider systems whose requirements can be viewed as a set of assertions of the form

$$\mathit{trigger} \Leftrightarrow \mathit{output}$$

where *trigger* denotes the set of conditions under which the output (or set of outputs) should be produced. They stress the importance of the bidirectional implication

[48] In logic, this term has a very precise meaning: a formal system is complete if every true fact is provable (it is *sound* if every provable fact is true). But this is not the sense in which the term is used of specifications.

($\Leftrightarrow$): in safety-critical applications it is not only necessary to be sure that an output is produced when required (i.e., $trigger \Rightarrow output$), but also that it is produced *only* when required (i.e., $trigger \Leftarrow output$). This means that all the conditions that should cause a particular output are disjoined (or'ed together) to create the trigger for that output. Triggers can refer to input data values, to the absence of a data value (within a specified time bound), and to the value of state variables.

The basic notion of completeness of such a specification is that the disjunction of all the triggers should be a tautology (i.e., the triggers must completely cover the space of possible behaviors). For example, if we have the trigger

$$height < 50$$

then we must also have some other trigger containing the expression

$$height \geq 50$$

(or some set of triggers whose disjunction is equivalent to this). More subtly, note that both these triggers imply that a *height* input datum is available; therefore there should also be a trigger that is activated when this input is *not* available.

Generally it is not enough to consider only the presence or absence of input data; most process-control systems are cyclic and it is the presence or absence on each particular iteration, or in some specified time window, that is important. All expressions appearing in triggers should therefore contain time bounds (except those that refer to the start-up event), and the tautology condition applies to these time bounds as well.

Jaffe and Leveson extend these basic completeness criteria with consideration of capacity (and the need to specify what should be done in the case of overload), start-up and shut-down, and responsiveness. The latter is an interesting notion. Certain outputs should lead to noticeable changes in inputs some time later (e.g., a "roll right" output should roll the airplane, and sensor data should shortly confirm that this is happening). Completeness requires that each such output should have two associated triggers: one that recognizes the expected response, and one that recognizes its absence or an unexpected response.

Although most of Jaffe and Leveson's completeness criteria concern triggers, they also propose some for outputs. For example, the output that turns a piece of equipment *on* may normally expect that it was formerly *off*. By including this condition in the trigger for that output, the tautological completeness criterion will force consideration of the case that the device is already *on*. More global criteria can be based on reachability analyses. For example, if all paths from a given state that cause a certain device to be switched *on* have triggers that require the presence of data from a particular source, then loss of that source will prevent the device from being switched on once the given state is encountered. This might be intended (e.g., a fail-safe condition on the arming of a weapon) or it might not (e.g., the inability to use the brakes on landing because a failed load sensor suggests the plane is still airborne) and a warning or override might be appropriate.

Jaffe and Leveson's criteria for completeness are obviously not exhaustive (it seems unlikely that any general criteria could be), but they offer a good starting point that may be modified or extended to suit the circumstances of particular system developments. Equally obviously, the examination of a specification according to their criteria can be performed informally (e.g., Lutz [Lut93b] reports that a checklist derived from Jaffe, Leveson and Melhart's proposals would have identified over 75% of the "safety-critical" faults discovered during integration and system testing of the Voyager and Galileo spacecraft). However, formal specifications allow their criteria to be posed as formal challenges that can be analyzed using formal verification techniques.

2.4 Benefits of Formal Methods

I have indicated several potential benefits of formal methods in earlier sections; here I collect together the various advantages claimed for formal methods and present them in a fairly systematic manner. The presentation divides into three subsections, the first two describing the benefits of formal specifications and formal verifications, respectively, and the third focusing on the potential contributions of formal methods to the assurance of safety-critical systems. In the section after this, I discuss the other side of the coin: the fallibilities of formal methods.

2.4.1 Benefits of Formal Specification

> "*By being educated in a technique, we are also educated to have a way of looking at the matter which is just as firmly rooted as that technique.*" [Wit83, paragraph IV–35]

A major benefit that is claimed for formal methods in general is that the concepts from logic and discrete mathematics that are embodied in formal specification methods provide effective mental tools for organizing our thinking about computer systems and for communicating our thoughts to others. The notions of sets, relations, functions and their various properties and operations, together with the ideas of universal and existential quantification, provide a set of mental building blocks that allow specifications to be built up in a fairly clear and straightforward way, without the looseness and unconstrained freedom of natural language, or the specious precision of most diagrammatic notations, or the algorithmic detail of pseudocode.

The chief technical advantages claimed for formal over informal specifications are precision and lack of ambiguity. Imprecision and ambiguity are two closely related flaws that can afflict specifications; I will say that a specification is *imprecise* if it does not provide enough information to determine what is intended in a particular circumstance, and *ambiguous* if the intentions are open to different interpretations. For example, an imprecise specification might say that the undercarriage should be lowered for landing, but not indicate at what height it should be lowered; whereas an ambiguous specification might say that the control surfaces on each wing should be driven by different channels—does this mean the ailerons on the left should be driven by one channel and those on the right by another, or that the ailerons should be driven by one channel and the spoilers by another, or some yet more complicated arrangement? Imprecision is clearly related to incompleteness, but I will reserve the term *incompleteness* to describe the situation where nothing is said about what is intended in a particular circumstance (so that it is not even clear if there is an intention to specify behavior for this circumstance).

Imprecision and ambiguity in informal specifications can arise because natural language constructions often depend on context,[49] because many words have several different meanings, and because people can be careless or ignorant of grammar. The main way that formal methods help reduce imprecision and ambiguity is by removing the associations carried by natural language: the terms in a formal specification have no meaning other than those explicitly specified. Thus, if we start a specification by stating

$$mode = landing \supset undercarriage\_position = down$$

then the questions naturally arise: what other *mode*s are there, and how and when do we enter and leave *landing* mode? By isolating the term *landing* and placing it in a formal context, we have stripped it of some of its intuitive associations and made it clear that it will be necessary to be explicit about those attributes of "landing" that are germane to the problem being specified.

Thus the simple fact that formal specifications require us to "define our terms" encourages a more systematic, and therefore a generally more complete and precise, enumeration of requirements than informal specifications. Other symbolic specification techniques, such as pseudocode, data dictionaries, and those based on dataflow representations, can have similar benefits, but generally they also impose more concrete and operational detail than may be necessary or desirable in requirements specifications, and they do not lend themselves to formal deduction.

The possibility of formal deduction means that formal specifications can also assist in the development of more precise and complete specifications through the

[49] Compare the injunction "seat belts must be worn" seen at a freeway entrance with "dogs must be carried" seen by an escalator.

process of "challenging" them. When we first challenge a specification with a conjecture such as "does sorting a sorted sequence leave it unchanged?" the answer is often "maybe": we discover that the specification is not sufficiently precise or complete to answer the question. Thus, posing a challenge can prompt us to consider missing or imprecise aspects of the specification (in this case, the topic of stability in sorting), and to decide consciously on the intended treatment—which may be to leave a certain property *deliberately* unspecified at this stage.

Formal specifications can assist in the identification and elimination of ambiguities in much the same way they do for imprecision. Ambiguities can be real or apparent. A real ambiguity arises when two contradictory statements can be deduced from a specification. In this case, the specification must be inconsistent, and techniques for avoiding inconsistent specifications were described in the previous section. Apparent ambiguities arise either when the specification allows two statements that seem in conflict, or when single statements are interpreted in different ways by different people. Assuming a specification is consistent, apparently contradictory statements must indeed be only apparently so. An appropriate response is to challenge the specification further in order to better understand why it entails both statements, or to better understand the reasons why the statements strike us as contradictory. Statements that are interpreted differently by different people can be probed by inviting those involved to sharpen their points of difference. It should be possible to identify a testable distinction: "if this means what I think it does, then this ought to follow, whereas if you're right, that ought to be the case." If one conclusion or the other is a valid consequence of the specification, then the difference of opinion should have been resolved. If both or neither follow from the specification, then we must probe further.

In all these cases, the probing and resolving of imprecision and ambiguity through challenges and deduction are possible precisely because a formal specification is a formal system: the consequences of the specification are just those formulas that can be calculated (i.e., proved) by formal deduction from its axioms and definitions. We can therefore settle with certainty whether or not a given formula is entailed by the specification. Thus, an advantage of formal specifications is that questions about their meaning and the consequences that they entail can be posed and answered with scientific precision. In nonformal developments, such precision is usually possible only much later in the lifecycle, when programming is under way.

And that is another significant advantage of formal over informal methods of specification: they provide specifications that can be rigorously checked, analyzed, and tested in various ways much earlier in the lifecycle than informal methods. This means that more exacting validation is performed earlier in the lifecycle than would otherwise be possible, thereby reducing costs through early detection and correction of faults, and very likely contributing to a better final product.

But although the ability to record and analyze decisions early is one of the main benefits of formal methods so, paradoxically, is the ability to postpone such decisions. As I have mentioned before, other symbolic methods of specification such as pseudocode and dataflow and state-machine representations share some of the advantages of formal specification, but often require premature commitment to design details. Formal methods, especially those in the property-oriented axiomatic style, allow one to specify only what is necessary at a particular level: some properties and behaviors can be left deliberately unconstrained if their elaboration is better postponed to a later stage of development. This "separation of concerns" allows development and analysis to focus on one thing at a time. Furthermore, specification of those properties that are handled at a particular level can be done without necessarily suggesting an implementation or fixing a lot of concrete detail.

2.4.2 Benefits of Formal Verification

> "*Proof, one might say, does not merely show* that *it is like this, but* how *it is like this. It shows* how $13 + 14$ *yield* 27." [Wit83, paragraph III–22]

Formal verification is often equated with "proof of correctness," but this simultaneously over- and underestimates the benefits that formal verification can provide. It overestimates the benefits because "correctness" suggests an absolute guarantee that is impossible for any enterprise that uses modeling to predict the behavior of real-world artifacts. A formal verification provides very strong guarantees on the mutual consistency of the specifications at either end of the chain of verification, but we can have no absolute guarantees that the upper specification captures all the requirements perfectly, nor that the lower specification (which may be a program or gate layout) exactly describes the behavior of the actual system: both of these are issues that require validation, not verification.

The "proof of correctness" conception underestimates the benefits of formal verification by failing to recognize the other products of verification. First, a formal verification does not merely give us strong assurance that certain theorems concerning the mutual consistency of specifications are valid, it enumerates all the assumptions, axioms, and definitions used to establish that fact. It therefore identifies those properties whose satisfaction, or utility, in the physical world must be established by empirical validation. At the top, these will include the requirements specification against which the verification was performed and, at the bottom, the assumptions about the real world (e.g., the behavior of clocks, or the semantics of Ada programs) on which the verification rests.

Second, not only does formal verification *identify* assumptions and requirements precisely, but as I have noted before, the machinery of formal verification can be used in the *validation* of those properties: theorem proving provides a means for testing the specifications of requirements and assumptions by evaluating challenges posed against them.

Third, formal verification is an extremely potent way of detecting design faults. Long before its verification succeeds and a design is pronounced "correct" (subject to the caveats described above), it is likely that several failing attempts at verification will have identified oversights, missing cases, and plain mistakes. Of course, some of these faults might have been caught by informal verification or, much later, by testing, but my experience is that formal verification reveals subtle mistakes whose detection by other means seems uncertain at best. State exploration (recall Section 2.2.3) can be considered as a formal method optimized for such debugging purposes.

Fourth, formal verification allows the consequences of changed assumptions or modified designs to be explored reliably. For example, the journal proof of the Interactive Convergence clock synchronization algorithm assumes that the initial clock corrections $C_p^{(0)}$ are all zero. This turns out to be inconvenient when implementations of the algorithm are considered. With a mechanically-checked formal verification, we can explore the significance of this assumption by simply deleting it from the formal specification and rerunning all the proofs. In this particular case, it turns out that the proofs of a few internal lemmas need to be adjusted, but that the rest of the verification is unaffected. Even if the initial argument for the correctness of this algorithm could be performed reliably by informal means (and we discovered that all but one of the proofs in the journal presentation of this algorithm were flawed [RvH93]), it would be exceedingly difficult to maintain the same standards of accuracy and rigor on subsequent reverifications against slightly altered assumptions. Yet reliable and inexpensive reverification is essential to the iterative processes within any realistic lifecycle model. As a design develops, one discovers simplifications, improvements, and generalizations that should be assisted, not discouraged, by investment in an existing verification. Even if we choose not to improve or change our design voluntarily, a change in external requirements or in the specification of an assumed service will require modification and reverification.

Fifth, formal verification subjects a design to intense intellectual scrutiny. This can be a benefit in its own right, often leading to improved understanding of the problem and to better solutions. For example, we may find after performing a formal verification that we actually used fewer assumptions than were made in the specification. This discovery can allow us to prune the collection of properties that need to be validated experimentally, or to simplify the specification of components whose properties are used in the verification. Alternatively, the improved understanding that comes with a formal verification may allow us to replace some assumptions with others that may be easier to check or to satisfy.

Finally, the understanding that comes from a well-conducted process of formal specification, validation, and verification may give us the confidence to consider new areas in the design space. Equipped only with tools for informal reasoning we may, through caution or ignorance, consider just a small area of that space. We may use certain designs (e.g., asynchronous channels in a flight-control system) because there is a plausibly simple argument for their correctness, and ignore others (e.g., the synchronized approach) because the reasoning that supports them is intricate. Formally supported analysis might reveal that the simplicity of the argument in favor of the preferred design is specious, and might provide the tools for mastering the intricacy of the alternative design.

2.4.3 Assurance Benefits of Formal Methods

Anticipating the case developed in Section 3.1, I assert that assurance for safety-critical computer systems is chiefly derived through scrutiny of the processes employed in its construction: we cannot measure the quality of the product so we look at how carefully it was built. At every stage in its construction, we expect careful analyses and reviews to be conducted to ensure the quality of the development performed in that stage. According to DO-178B [RTCA92, Section 6.3]: "analyses provide repeatable evidence of correctness and reviews provide a qualitative assessment of correctness" (a draft version of DO-178B stated "reviews provide a group consensus"). For assurance, the chief benefit provided by formal methods is that they allow reviews to be replaced or supplemented by analyses. Depending on the level of rigor employed—that is, on the extent to which repeatable, calculational processes are used—formal methods should be able to detect faults *earlier* than otherwise, and with greater *certainty* than otherwise. In certain circumstances, subject to caveats concerning the fidelity of the modeling employed, formal methods can *guarantee* the absence of specified faults. However, not all processes can be reduced to analysis, even with fully formal methods, and reviews will still be required. Formal methods can contribute to high-quality reviews by settling questions concerning syntax, type-correctness, and internal consistency by analytic means, thereby allowing the reviews to focus on matters of greater substance.

Formal methods establish properties of mathematical *models* of the systems considered; many of formal methods' detractors focus, correctly, on the difficulty of establishing the accuracy of the models employed. These issues are considered in the next section.

2.5 Fallibilities of Formal Methods

Formal methods do not guarantee a superior product; as with all tools, formal methods must be used with skill and discrimination if they are to deliver benefit.

In this section, I examine some of the ways in which formal methods may provide less benefit than anticipated and I consider ways to ameliorate these difficulties. In most cases, the fallibilities that I identify also attend informal development methods, but the ameliorations are generally possible only with formal methods (e.g., informal as well as formal specifications can be inconsistent, but only formal specifications admit formal demonstrations of consistency). Also in this section I consider arguments that have been mounted against the use of formal methods in general.

As with my discussion of benefits, I consider the fallibilities of formal specifications and of formal verifications separately.

2.5.1 Fallibilities of Formal Specifications

Fallibilities of formal specifications can include specific deficiencies such as inconsistency or incompleteness that may afflict a particular specification, as well as general vulnerabilities that are sometimes used to call the whole enterprise into question. I begin by considering the more general arguments against formal specification.

The broadest challenge against the formal approach to specification is that mounted by Naur [Nau82]. Naur does not argue against formalism *per se*, but against the rather rigid conceptions of formalization that were being proposed in the 1980s. Although his case is less effective against modern approaches, some of those who oppose formal methods still employ similar arguments.

Naur asserts that late-lifecycle program-level specifications of the kind that were advocated in the 1980s offer little benefit:

> "According to these ideas the programmer must first express the solution in a so-called formal specification language, second express the same solution in a programming language, and third prove that the two solutions are equivalent, in some sense...this approach offers no help to the programmer but only adds to his or her burdens. ...with this approach the programmer has to produce not one but two formal descriptions of the solution, and in addition has to prove that they are equivalent."

This quotation (and the next) reveals one of the major weaknesses of Naur's argument: his focus seems to be "programs" (small entities constructed by a single person) rather than "systems" (large entities developed by teams against multiple and exacting requirements). Formal specifications may contribute little to developments where the program itself can be considered an adequate expression of the solution, but safety-critical systems are seldom of this kind. However, if we interpret Naur's remarks as applying to just the coding stage of the development lifecycle, then his point has some validity: at some point in the development, programming must take over from specification.

Naur notes that in mathematics and engineering, formal and informal modes of reasoning and notation are used in combination, each where it is most effective, and he argues for a similar approach in program specification:

> "Specifications are sometimes a necessary evil, to be used for documentation of such aspects of programs that are not satisfactorily documented by the programs themselves...The ideal specification of an aspect of a program is a description of what that aspect does such that on the one hand it is intuitively obvious to the user that it corresponds to his or her requirements, and on the other hand it is equally obvious to the programmer that it is realized in the program itself. In either case the intuitive understanding may need to be supported by an argument having several steps, a proof..."

Naur observes that the notation and formalism best suited to specifying "aspects of programs" may vary from aspect to aspect and from program to program and he therefore argues against the use of fixed specification languages, instead advocating "formalisms of any suitable form, such as tables for enumerating cases, decision tables, tables or graphs corresponding to finite-state algorithms, program skeletons, formulae of any kind etc."

Naur's position here can be interpreted as a repudiation only of Level 2 and Level 3 formal methods, and as something of an endorsement for Level 1. However, this interpretation must be performed relative to the technology available at the time of his writing, and relative to his conception of the problem addressed. As already noted, it seems clear that Naur's conception of the problem is one of "program development" rather than "systems engineering": that is, he is concerned with the late-lifecycle stages of design. The formal specification notations available in the early 1980s also focused on these stages and for the most part were limited and restricted notations that merited Naur's criticism. The importance of the early stages of the lifecycle has become far more widely recognized in the dozen years since Naur's paper was written, and formal specification notations have become far richer and less like program annotations. While Naur is probably correct to question the value of a formal specification that is little different from the program it is intended to specify, his argument is less effective against formal requirements and early-lifecycle design specifications that are several stages removed from the levels of description where programming begins. Naur's other point, that formal and informal modes of expression should be used together, and that formalisms should be chosen freely, rather than limited to those that can be expressed within a fixed specification language, retains its value today, but needs some qualification. The mixing of formal and informal modes of expression is encouraged by most modern presentations of formal specifications. Rather than a single large block of formal text, modern presentations of formal specification generally intersperse formal text with informal explanations. This style is assisted by the modularization constructs of modern specification languages, which encourage the development of specifications in relatively small units that can be combined to yield larger units.

Naur's recommendation that formalisms should be chosen freely, rather than restricted to those supported by a particular specification language needs to be tempered if the benefits of mechanical analysis and formally checked deduction are desired. Fortunately, the spirit of Naur's recommendation can often be preserved within a single specification notation: modern specification languages based on higher-order logic or set theory are sufficiently expressive that they can encode other formalisms such as finite state machines or temporal logics relatively conveniently. If combined with simple tools for printing such encodings in an attractive form, a single specification language can often accommodate several styles of specification, and furthermore permit their use in combination, yet provide uniform mechanical support.

Several other arguments against formal specifications are similar to Naur's: it can be argued that formal specifications are hard to write and to read, especially for nonspecialists, and that too much of the effort in writing formal specifications is spent wrestling with the formalism and overcoming limitations of notation, rather than focusing on the substance of the problem. Like Naur's criticisms, these have some validity, but are less true of modern formal specification techniques than of those used in the 1980s. Specification languages based on first-order logic with set theory, or on higher-order logic, are generally sufficiently expressive that most concepts can be stated in a fairly direct and unforced way. More limited languages that lack quantification, or that restrict the use of axioms, or the class of expressions that can be used in axioms, do generally require rather convoluted specifications, but may offer compensations such as executability or highly automated theorem proving that can be advantageous in some applications. Of course, one cannot expect even the most perspicuous of formal specifications to be accessible to readers without some training, but the evidence seems to be that those with engineering or scientific backgrounds can be taught to *read* formal specifications in a matter of days—provided they see some tangible benefit (e.g., some form of mechanical analysis) in doing so.

Learning to *write* formal specifications is another matter, but the difficulty is often one of learning to write good, suitably abstract *specifications*, rather than of learning the details of a particular *formal* specification technique. It seems neither necessary nor desirable that "users," "customers," or "engineers" should write formal specifications on their own—any more than they should write their own programs, design their own circuits, or do their own welding. This is not to say that "formal specifier" must be a job title—there is no reason why formal specification should not be an additional skill for someone with other primary responsibilities—but it must be recognized that formal specification no less than, say, technical writing, is a specialized skill that needs talent, training, and experience.

A top-level formal specification of requirements must be developed through a dialog between a specifier and those who understand what is to be specified. Through discussion and questions the specifier elicits and records the salient properties of the thing that is wanted. The acts of recording and of formalizing these properties may suggest questions that need clarification and further discussion. Those who understand what is wanted may either read the evolving formal specification or may ask the specifier to interpret it for them, and in either case can challenge the specification as described in Subsection 2.3.2 to validate it against their informal conception of what is required. Similar dialog between specifiers, designers, and those with an understanding of what is required, must take place at the later stages of the lifecycle, as intermediate specifications are developed on the way to an eventual implementation.

I now turn from general to specific deficiencies that can afflict formal specifications. A formal specification can be faulty in several different ways. It may say too little (thereby failing to characterize the matter of interest), or too much (thereby overly constraining later stages of the development), or it can be wrong. And it can be wrong in at least two major ways: it may not say what we wanted (in the limit, it may be inconsistent and say nothing at all), or it may say what we wanted—but what we wanted was wrong.

Most of these are issues of validation, and were covered in Section 2.3. The primary methods for validating a formal specification are inspection and challenges–that is, posing questions that should be answered by the specification. It cannot be stated too forcefully that validation of formal specifications is critical to their utility. Simply writing specifications in a formal notation does not guarantee their quality.

The fault of overspecification—saying too much and thereby reducing subsequent design freedom—is less amenable to detection by these means than other kinds of fault. Jones [Jon90, Chapter 9] gives some technical criteria for implementation bias in model-oriented specifications, but this is generally a fault that requires judgment and experience to evaluate—and for these reasons is one of the most common faults observed in formal specifications, especially in those written by people whose prior experience is mainly of programming.

2.5.2 Fallibilities of Formal Verifications

The general arguments against formal verification can be summarized as: (1) mathematical modeling cannot account for the behavior of real-world artifacts with perfect accuracy, and therefore the whole endeavor is futile, and (2) the formal conception of proof does not correspond to the evidence that convinces human reviewers. More specific concerns include the possibility that formal proofs, including those checked by machine, may be incorrect. As in the previous section, I begin with the general arguments.

Fetzer [Fet88] observes that the behavior of computer systems in execution depends on that of physical devices (the computer circuitry and, in the case of control systems, sensors and actuators) and logical artifacts (the compiler and operating system) whose actual operation may not match that formalized in the verification. This is, of course, true—as it is true of all applications of mathematical modeling in engineering, whether fluid dynamics, mechanics, or theories of combustion. Fetzer does not mention that similar concerns attend the fidelity of the requirements specification at the other end of the verification.

Whether formally verified or not, a system is a designed artifact, constructed consciously to achieve certain goals subject to certain assumptions and constraints. The system may fail if the assumptions on which it is based, or the requirements it is built to satisfy, prove erroneous. The only difference made by formal, as opposed to informal verification, is that formal verification makes all the assumptions and the requirements explicit, and provides a very strong guarantee that the design satisfies the requirements, subject to the assumptions. It is the job of validation to provide assurance that the requirements and assumptions accord with external reality and expectations. Fetzer seems unaware of the concept of validation and seems to believe that proponents of formal verification assert that it provides unequivocal "proof of correctness" and can replace testing. He bolsters his case with selected quotations (mostly rather old) from Hoare, Dijkstra, and others that appear to lend credence to this claim.

As I hope the earlier sections of this report have made clear, modern conceptions of formal methods do not claim unequivocal "correctness" (see, for example [Coh89b]) and, in fact, are deeply concerned with balanced, whole-lifecycle approaches to assurance. Far from ignoring assumptions about the behavior of the physical world, formal methods help identify these assumptions more clearly, so that they may be examined and validated—empirically, if necessary. A concept associated with formal methods for critical systems is *design for validation* [JB92], which holds that systems should be designed so that it is feasible to validate all assumptions, and to measure all parameters, empirically. The argument for this approach focuses on the transition probabilities of Markov reliability models, but it is applicable to all assumptions that underlie the design of critical systems. If an assumption cannot be validated, or a parameter measured, in feasible time on test, then the system should be redesigned so that it does not depend on those uncertain properties. Often, this may entail use of more sophisticated reasoning. If, for example, a fault-tolerant system is designed to tolerate specific failure modes, then we should require quantified or analytic evidence that no other modes of failure can occur. If it is infeasible to provide such evidence, then perhaps the system should be redesigned so that it does not depend on those specific assumptions. Providing and demonstrating fault tolerance in the absence of assumptions about faults is difficult (this is what Byzantine fault tolerance is all about), and it may require formal methods to achieve an adequate level of assurance that this has been achieved.

An older argument against formal verification is due to De Millo, Lipton, and Perlis [DLP79]. Their case is that a proof is truly an argument that convinces other people, not the formalist conception of a sequence of inferences in some formal system: they claim that a proof can become accepted as such only through a "social process" of human review. There is something to this point of view (which is primarily an argument against automated Level 3 applications of formal methods): at the highest level the arguments that sustain a verification should be subjected to responsible human review, not delegated to an automatic theorem prover. Few proponents of formal methods would disagree with this, but De Millo, Lipton, and Perlis invent a parody of their position and excoriate them for it:

> "The scenario envisaged by the proponents of verification goes something like this: the programmer inserts his 300-line input/output package into the verifier. Several hours later, he returns. There is his 20,000-line verification and the message 'VERIFIED'."

In fact, a formal verification system or, more particularly, its theorem prover does not act as an oracle that certifies programs for inscrutable reasons, but as an implacable skeptic that insists on its human user providing justification for every significant step of the argument. After persuading a formal verification system to accept a verification, at least one human—the user of the verification system—will have achieved great insight into the arguments that sustain the proof. In my experience, this insight is often deeper than that obtained through ordinary informal proofs, but can be distilled into an informal proof that can be communicated to others. Usually, this informal distillation of a formal proof is superior to an informal proof constructed normally in its attention to details and boundary cases, and yet is sometimes simpler, too.

So a formal verification need not usurp the social process but can enhance it, by providing a better proof for consideration. This is important because, while the social process may be valuable in evaluating the broad structure of an argument, it does not seem particularly effective at scrutinizing intricate details. It is important to recognize that the arguments involved in verification are rather different from those of conventional mathematical theorems, which De Millo, Lipton, and Perlis take as their model. First, the arguments in verification are seldom intrinsically interesting, nor are the theorems proved of general interest—facts that limit widespread participation in a social process. Second, the arguments tend to be full of intricate detail, boundary conditions, and case analysis—the types of reasoning where human faculties seem most fallible, and mechanical assistance most effective.

I can cite my own experience in support of these observations. The journal proof [LMS85] that the Interactive Convergence Clock Synchronization Algorithm maintains synchronization despite the occurrence of Byzantine faults comprises five lemmas and a main theorem. Formal analysis, conducted by Friedrich von Henke and myself using EHDM, demonstrated that four of the five lemmas, and the main theorem, were false as stated [RvH93]. As far as I know, these flaws had not previously been detected by the "social process" of refereeing prior to publication and peer scrutiny afterwards, despite this being a frequently cited paper. Some of the faults in the proofs are painfully obvious once they have been spotted: for example, the problem in the main theorem is that it seeks to establish a strict inequality by an inductive proof, but the inductive step yields only an "approximate" inequality. Another example includes a modified algorithm for interactive consistency (the problem of distributing consistent sensor samples in the presence of faults) [TP88] that contains an outright bug, despite a refereed and published informal proof. If published proofs of important algorithms are inadequately checked by the social process, how are the larger and less interesting arguments for particular systems to be adequately examined? Barwise [Bar89b] provides a thoughtful discussion of these and related issues raised by Fetzer and by De Millo, Lipton, and Perlis.

Even those who are undisturbed by the general arguments against formal verification may be troubled by use of mechanical theorem provers: "what if the prover is wrong?" is a frequently asked question. Although this concern cannot be dismissed, I do not consider it a significant threat to the overall utility of formally checked verification—just as concern for possible numerical inaccuracies in programs for finite-element analysis does not prevent engineers building better and safer structures with the aid of such programs than without.

All who build and use automated theorem provers must take care to ensure the soundness of the deductions performed, but concern should not become paranoia. Although theorem provers are complex programs, much of their complexity is in the search procedures that look for an appropriate deduction to apply; once such a deduction is found, the code that applies it can be relatively straightforward. A bug in the search procedure will affect the prover's ability to find proofs, but not the soundness of the proofs produced. Soundness depends on the relatively small part of the system that applies the rules of inference. This code can reflect quite directly the corresponding rules of inference in the logic and is exercised with great frequency: often there are no more than a few dozen such rules, and they are called tens or hundreds of thousands of times in even quite modest-sized proofs. Thus, bugs in well-designed and well-tested theorem provers tend to be found in their search procedures, not their basic mechanisms, and consequently their failure mode is an inability to prove true theorems, not the erroneous "proof" of false ones.

There is some design freedom in choosing the number and complexity of the basic deductions performed by a theorem prover. Some provers are deliberately built on very few rules of deduction, with very direct implementations, in the belief that this is the best way to ensure their soundness:[50] HOL [GM93], for example, has only seven primitive inferences. Unfortunately, this choice eliminates decision procedures for arithmetic and other basic theories and renders HOL and similar systems vastly less productive for proof development than more automated systems; to my observation, it is economically infeasible, other than in research environments, to use systems with a very austere foundation to prove theorems of any scale or complexity.

Those who can be satisfied only with a proof that is checked by a system such as HOL with a very direct implementation of the rules of inference for its underlying logic, could consider using a more automated system for a similar logic to actually develop the proof and switch to the more "secure" system only for its final "certification." The UK Interim Defence Standard 00-55 [MOD91a, paragraph 32.2.3] goes further and requires that mechanically-checked proofs should be duplicated using a "diverse tool." In my opinion, the resources expended on such "second opinions" would probably be better expended on independent scrutiny of the assumptions and modeling employed, which are rather more likely to be faulty than mechanically-checked proofs.

The soundness of the analysis performed with the aid of a mechanized tool depends on more than the basic deductions performed by the theorem prover: the specification text must be parsed correctly, definitions must be well-formed, the consistency of axioms must be demonstrated, and dependence on unproved lemmas must be identified. For these reasons, de Bruijn, in his excellent paper [dB91], suggests that the term "justification system" should be preferred to "proof checking system" since "they handle much more than proofs...some of these systems can check

[50] Soundness depends on more than just the number and simplicity of the basic deductions; the representation used for formulas of the language is also very important. In particular, nested quantifier and other variable-binding operators (such as λ-expressions) must be handled with great care in order to avoid "variable capture." Experts generally favor the "de Bruijn representation" [dB72] for this purpose, and those concerned about the soundness of the tools that they use should inquire about the treatment of these representational issues.

complete theories including whether definitions and axioms are well-formed...and they can check relations between different theories."

Of course, bugs in a theorem prover or justification system must be detected by a user noticing something wrong. Some may be skeptical of a user's ability to notice faulty reasoning in a theorem prover, and will complain that I am guilty of a double standard: I advocate the use of mechanically-checked proofs because human reasoning can be unreliable, and I minimize the seriousness of bugs in the theorem prover because the human user will catch them. This is not quite so implausible as it may seem: if the individual inference steps are well-matched to those employed by humans (i.e., little more detailed than the steps of a journal proof), then the user and the theorem prover are engaged in a kind of "social process," each checking the other's work.

Finally, it should be remembered that the reason for using mechanized theorem proving is not to relieve human users of responsibility, but to augment their ability to conduct arguments of great length and in extreme detail. This recognition should also guard against extravagant interpretations of terms like "formally proved." As noted in the introduction and explained in more technical detail in the earlier sections of this chapter, "proof" is a technical term for a certain kind of "logical calculation," and does not, on its own, connote complete certainty nor fitness for purpose. An interesting lesson in this regard can be drawn from circumstances attending the British VIPER microprocessor. The following description is summarized from MacKenzie [Mac91].

VIPER was a microprocessor designed by an agency of the British Ministry of Defence for safety-critical applications [Cul88]. The specification and verification of this device used a variety of Level 1, 2, and 3 formal methods [CP85, Pyg88]. The portions that were subjected to Level 3 analysis employed HOL [Coh88, Coh89a]. A commercial company licensed some aspects of VIPER and promoted it vigorously using claims such as "the first commercially available microprocessor with a formal specification and a proof that the chip conforms to it."

When a NASA-commissioned review of the VIPER concluded that the VIPER had been "extensively simulated and informally checked...but not formally verified" [BH90, BH91] (the reviewers reserved the term "formally verified" for top-to-bottom Level 3 applications and seemed to miss some of the pragmatic utility of the compromises made in the VIPER development), the company that had licensed VIPER technology took legal action in the British High Court, alleging that the claim that VIPER had been "proved" was negligent misrepresentation.[51] Shortly

[51] The scientific papers describing verification of VIPER are scrupulously careful to describe the limitations of what had been accomplished [Coh88, Coh89a, Coh89b]. It seems that the plaintiffs were unaware of these papers.

thereafter, the company went into liquidation and dropped the case, so the court did not have to rule on what constitutes mathematical proof.

The lesson here is that the dispute did not center on whether there was adequate assurance that the VIPER was fit for its intended (safety critical) purpose, but on whether the particular combination of formal methods that had been applied to it constituted "proof." Repetition of this foolishness should be avoided. Formal methods can provide important evidence for consideration in certification, but they can no more "prove" that an artifact of significant logical complexity is fit for its purpose than a finite-element calculation can "prove" that a wing spar will do its job. Certification must consider multiple sources of evidence, and ultimately rests on informed engineering judgment and experience.

2.6 Mechanized Support for Formal Methods

In this section I discuss the main themes and directions in the development of support systems and tools for formal methods; I focus primarily on those intended for Level 3 applications. This is not a survey of particular systems, and I mention by name only a few that are representative of the main directions that I identify. My intent is to provide information that will enable users and certifiers to calibrate claims about the effectiveness of various approaches and to alert them to potential drawbacks as well as advantages. The reader should beware that this section reflects personal opinions (acquired over more than a decade spent building and using tools for formal methods), and does not represent a consensus view.[52]

There can be value in applying the techniques of formal methods using just pencil and paper (i.e., Level 1). However, larger specifications may benefit from mechanical assistance. Parsing and typechecking of specifications (i.e., Level 2) generally detects numerous small mistakes, and attempting to prove conjectured properties of a specification usually detects many more, and deeper, errors. Proofs of properties of specifications or implementations often involve very lengthy, detailed, and error-prone chains of argument. Use of automated theorem provers or proof-checkers (i.e., Level 3) can eliminate these errors, and allows a truly exacting degree of scrutiny that is infeasible by other means. It is through use of mechanized tools that formal methods contribute most obviously to assurance for safety-critical systems—by replacing intuition and group consensus with repeatable, checkable, analysis.

A formal specification language and its Level 3 support tools need to be grounded on formal logic, but simply mechanizing the standard notations of logic and the techniques of proof theory would provide utterly inadequate support for practical

[52]The reader should also beware that one system which receives favorable mention here (PVS) is a product of the laboratory to which I belong, and therefore conforms to my own prejudices.

formal methods. The formal systems of logic were developed for study, not for use: they were designed to answer questions about the foundations of mathematics and about their own consistency and completeness. Logicians are mostly satisfied to show that it is possible *in principle* to formalize parts of mathematics in some formal system or other, they are generally not interested in actually doing so. In formal methods, we want to formalize concepts and undertake formal proofs *in practice*—a quite different goal from that of logicians. The languages and techniques that serve logicians are no more appropriate for formal methods in computer science than the 5-tuples of a Turing machine are suitable as a practical programming language.

Hence, the challenge in developing languages and support tools for formal methods is to create systems that draw on the soundness and other properties and techniques of logic, while recasting them in a manner that allows them to be used productively in a practical setting. In the case of language design, this means a search for conveniently expressive notations that nonetheless have a simple foundation (inexpressive notations lead to the "Turing Tarpit" where everything is possible, but nothing is easy); and in the case of theorem proving, it means a search for a combination of relatively few, relatively powerful basic inference mechanisms, that can be combined to prove useful results in a manner that is both efficient and enlightening. Basing a theorem prover on some standard method from logic (for example, natural deduction) is to completely miss the point: we are not interested in mechanizing what logicians do—their systems were built for their purposes, ours need to be built for ours. Those who believe that only by mechanizing very elementary deductive systems can we be assured of soundness, need to face the consequence of that choice: namely, that it will be practically or economically infeasible to mechanically verify results of any significant interest (another variation on the Turing Tarpit: everything is possible in principle, but almost nothing is in practice).[53] The more productive approach is to establish the soundness of the implementations of techniques that are effective in practice.

Current verification systems represent three separate lines of development. One, which I will call the *integrated* line of development, consciously attempts to integrate a specification language and a mechanized theorem prover or proof checker into a single coherent system. The integrated verification systems were the first to be developed. Early systems of this type generally made many compromises in order to accommodate the technological limitations of their time: for example, the specification language was generally a fairly simple extension to first-order predicate calculus (sometimes closely linked to a programming language of the Pascal family), and the proof checker was generally fairly primitive. Early examples of integrated verification systems include Affirm [GMT+80, Mus80], FDM [LSSE80, SS84],

[53] "Just as in algebraic calculation and in almost all forms of notation commonly used by mathematicians, a workable instrument is preferable to one which is theoretically more perfect but in practice far more cumbersome." [Bou68, page 10]

Gypsy [GAS89, SGD88], Iota [NY82, YN85], HDM [RLS79, RL76, SLR78], and the Stanford Pascal Verifier [ILL75, LGvH+79]. Most of these systems embodied a particular approach to formal specification and analysis (e.g., state machines in the case of HDM and FDM, or equational specifications for abstract datatypes in the case of Affirm), and their versatility was correspondingly limited.

The second line of development grew from theorem-proving research; the availability of effective theorem provers encouraged people to write specifications directly in the logics supported by those provers. In most cases, the logic lacked the conveniences of a specification language (e.g., conventional syntax, computer-science data structures, support for modular specifications), but users were willing to trade those linguistic advantages for effective theorem-proving support. Examples of theorem provers that have been used for verification include the Boyer-Moore prover [BM79, BM88], Isabelle [Pau88], Otter [McC90] and RRL [KZ88].

The third line of development derives from efforts to provide truly expressive and attractive specification languages, initially without concern for their mechanization. Examples include VDM [Jon90] and Z [Spi93]. Users found these methods and notations helpful [Hay93, HK91] and later started developing mechanized support for them. Initially, the support was limited to typesetting, then parsing and typechecking were provided, and more recently support for theorem proving has been added [BR91, Nic89]. However, the capabilities of these tools, particularly the theorem provers, are currently very limited, scarcely equaling those of the integrated verification systems developed more than a decade earlier (although the languages are much richer).

None of the lines of development described above has yet provided a totally satisfactory system for formal specification and verification: the theorem-proving line offers inadequate support for specification, the specification line offers inadequate support for verification, and most examples in the line of integrated verification systems have been comprehensively inadequate. When and where, then, can we expect truly effective verification systems to emerge? I roughly characterize such a system as one that

- Supports a specification language as attractive and rich as those that have found favor in unmechanized applications,
- Provides support tools to assist in the validation and documentation of specifications, including parsing and typechecking of sufficient strictness that most simple faults are quickly flushed out,
- Is supplied with a theorem prover or proof checker that allows conjectures of all kinds (large, small, easy, difficult, valid and invalid) to be proved or found in error with an efficiency at least equal to that of the best stand-alone theorem provers.

One school believes that bringing modern theorem-proving techniques to bear on the specification-language line of development will produce the desired result. As noted, these attempts have not been successful so far, and I do not think it likely that they will be. Rich specification languages such as Z and VDM are not particularly well-suited to mechanical analysis. Their logical foundation is axiomatic set theory, which is an essentially untyped system. Thus, strict typechecking, which is desirable in a system subject to mechanical analysis, has to be grafted on in an ad-hoc and necessarily partial way [DHB91] (also recall Subsection 2.3.1.1). In addition, there are loose ends in the semantics of these notations, which may be of little moment when they are used quasiformally by people of good will, but which cannot be tolerated in fully mechanized systems that relieve the user of some of the responsibility for ensuring soundness. In Z, for example, the exact interpretation of partial functions, and of some aspects of schemas, still seem open to debate. Resolution of these loose ends in ways that are amenable to effective mechanical analysis and verification is proving difficult.

Faced with the challenge of mechanizing very rich logics, those developing theorem provers for the specification language line of verification systems have chosen to provide relatively little in-built automation and instead provide an interface that allows the user to select and apply basic rules of inference and to compose these into larger units (the theorem provers of Raise [RAISE92], the Balzac/Zola support system for Z [Har91], and *mural* [JJLM91] for VDM are like this). Attractive interfaces with multiple windows and menus have been provided and the systems are referred to as "proof assistants," the implication being that they actively help a human user develop and check a proof. Some systems of this kind, for example *mural*, aim to be generic with respect to the formal system they support, so that the specific rules for reasoning about VDM (its main application) are not built in to *mural*, but are loaded as a library (although the result "is not a fully-fledged support environment for VDM" [BR91, page 387]).

The motivation behind these approaches is reasonable, but the efficiency and power of the theorem-proving support that is achieved leaves much to be desired. Of one small example using *mural*, the authors state

> "the task of verifying and validating an example like the one presented here may take an experienced user two weeks. This is not a problem specific to *mural*, but is more to do with the level of detail at which one is forced to work when doing 'fully formal' development" [FEG92].

I can see nothing in this example that should detain an experienced user, armed with effective tools, for more than a few hours. A step-by-step diary of the development of a rather more challenging—though still small—example (the correctness of a compiler for additions and assignments) using the Boyer-Moore prover, indicates that it was completed in about five hours [WW93]. This seems a more reasonable expenditure of effort, and is consistent with my own experience: theorem provers with built-in mechanisms for automation are at least an order of magnitude more productive than systems that lack such mechanisms.

The woeful inefficiency of the "proof assistant" type of theorem prover does not rule out more effective automation for the specification language line of verification systems. However, theorem proving for the rich formal systems underlying the attractive specification languages is a challenging problem—it is no accident that the most powerful theorem provers tend to be associated with the most restricted logics. For example, functions are inherently partial in set theory (where they are defined as sets of pairs), and this is inimical to efficient theorem proving. VDM, for example, uses a nonstandard three-valued logic to accommodate partial functions [BCJ84] and incurs the necessity to constantly discharge definedness obligations at proof time.[54] Set operations also lend themselves less well to mechanical deduction than predicates (the comparable notion in systems built on higher-order logic).[55] All of these problems can doubtless be overcome, but there is little indication that current work in the specification language line of development is headed in directions that will do so.[56]

If the specification language line of development is not yielding truly productive verification systems, what of the theorem-proving line? Could we not take an effective theorem prover and use its logic as a foundation for a specification language? An obvious difficulty is that, as noted above, the most effective theorem provers are associated with the most restricted logics (e.g., the logic of the very powerful Boyer-Moore prover is an unquantified, highly constructive first-order logic of recursive equations) and therefore provide the least comfortable foundation for an attractive specification language. Conversely, the proof checkers associated with more attractive logics, such as the higher-order logic of HOL [GM93] and TPS [AINP88] are

[54] Cheng and Jones [CJ90] consider some of the choices among multiple-valued logics. Other treatments of partial functions retain the standard two-valued logic, but change the interpretation of the quantifier rules. These are described in Appendix Section A.11.2.

[55] There are effective theorem provers for set theory, such as Eves [CKM$^+$91] and Ontic [McA89] (the latter is unconventional and very interesting), but these are outside the VDM and Z tradition.

[56] Building a theorem prover, let alone a complete verification system, is a demanding undertaking that poses many challenges. I know of no group that got it right on their first attempt, so a good question to ask the purveyors of any tool is: "how many previous tools of this kind have you built, and what was wrong with them?" At the end of a very good tutorial paper on theorem prover design [Pau92], Paulson states: "My final advice is this. Don't write a theorem prover. Try to use someone else's." For the same reason, I encourage those who would build special-purpose tools for their own applications to think again: before long you will find yourselves involved in language design and implementation, and in theorem proving techniques, and you will be distracted ever further from the problems you set out to solve. An approach that is more likely to be productive is to collaborate with the developers of an existing tool in order to augment it to provide the specialized analyses desired.

not particularly powerful. Intermediate points include provers for pure first-order logic such as Otter [McC90], and first-order equations such as RRL [KZ88].

But the main reason why pure theorem provers offer an imperfect foundation for verification systems is that few of them have the attributes required for really effective theorem proving *in support of verification*. Formal verification is a specialized application of mechanized deduction and imposes distinctive requirements on the theorem-proving tools that support it. It is not enough simply to provide a "powerful" theorem prover; the prover must be tailored to the requirements of verification. My colleagues and I, and others who have undertaken substantial verifications [SGGH91], have found not only that verification imposes specialized requirements on a theorem prover, but that different theorem-proving requirements emerge at different stages of a verification and that truly productive theorem proving must support the requirements of all stages.

My experience has been that each formal verification evolves through a succession of phases, not unlike the lifecycle in software development. I have found it useful to identify four phases in the "verification lifecycle" as follows.

Exploration: In the early stages of developing a formal specification and verification, we are chiefly concerned with exploring the best way to approach the chosen problem. Many of the approaches will be flawed, and thus many of the theorems that we attempt to prove will be false. It is precisely in the discovery and isolation of mistakes that formal verification can be of most value. Indeed, the philosopher Lakatos argues similarly for the role of proof in mathematics [Lak76] (see the quotation at the head of Section 3.6). According to this view, successful completion is among the least interesting and useful outcomes of a proof attempt at this stage; the real benefit comes from failed proof attempts, since these challenge us to revise our hypotheses, sharpen our statements, and achieve a deeper understanding of our problem: proofs are "less instruments of justification than tools of discovery" [Kle91].

The fact that many putative theorems will be false imposes a novel requirement on theorem proving in support of verification: it is at least as important for the theorem prover to provide assistance in the discovery of error, as that it should be able to prove true theorems with aplomb. Most research on automatic theorem proving has concentrated on proving true theorems; accordingly, few heavily automated provers terminate quickly on false theorems, nor do they return useful information from failed proof attempts. By the same token, powerful heuristic techniques are of questionable value in this phase, since they require the user to figure out whether a failed proof attempt is due to an inadequate heuristic, or a false theorem.

Development: Following the exploration phase, we expect to have a specification that is mostly correct and a body of theorems that are mostly true. Although debugging will still be important, the emphasis in the development phase will be on *efficient* construction of the overall verification. Here we can expect to be dealing with a very large body of theorems spanning a wide range of difficulty. Accordingly, efficient proof construction will require a wide range of capabilities. We would like small or simple theorems to be dealt with automatically. Large and complex theorems will require human control of the proof process, and we would like this control to be as straightforward and direct as possible.

Experience shows that formal verification of even a moderately sized example can generate many hundreds of theorems, many of which involve arithmetic. Effective automation of arithmetic, that is the ability to instantly discharge formulas such as

$$x \leq y \wedge x \leq 1 - y \wedge 2 \times x \geq 1 \supset F(2 \times x) = F(1)$$

(where x and y are rational numbers), is therefore essential to productive theorem proving in this context.[57] Decision procedures are known for certain other common theories, but to be useful these need to work with each other and with those for arithmetic in order to decide the *combination* of their theories. Practical methods for doing this are known [NO79, Sho84] and are used in some systems.

Other common operations in proofs arising from formal verification are to expand the definition of a function and to replace one side of an equation by the corresponding instance of the other. Both of these can be automated by a technique called *rewriting*. But it is not enough for a prover to have arithmetic and rewriting capabilities that are individually powerful: these two capabilities need to be tightly integrated. For example, the arithmetic procedures must be capable of invoking rewriting for simplification—and the rewriter should employ the arithmetic procedures in discharging the conditions of a conditional equation, or in simplifying expanded definitions by eliminating irrelevant cases. Theorem provers that are productive in verification systems derive much of their effectiveness from tight integration of powerful primitives

[57] The formal system known as Presburger Arithmetic, that is arithmetic with constants and variables, addition and subtraction, but with multiplication restricted to the linear case (i.e., multiplication by literal constants only), together with the relations $<$, $>$, $\leq$, $\geq$, $=$, and $\neq$, is decidable—this means there is an algorithm that can tell whether any such expression is valid or not. Implementations for verification systems generally need to consider extensions (e.g., the presence of uninterpreted function symbols) that are undecidable in the general (quantified) case. Restriction to the unquantified (also called "ground") case is often feasible, and practical decision procedures have been developed for this case [Sho77, Sho78, Sho79] and are highly effective in practice.

such as rewriting and arithmetic decision procedures. And the real skill in developing such provers is in constructing these integrations [BM86]. More visibly impressive capabilities such as automatic induction heuristics are useful, but of much less importance than competence in combining powerful basic inference steps including arithmetic and rewriting.

An integrated collection of highly effective primitive inference steps is one requirement for productive theorem proving during the proof development phase; another is an effective way for the user to control and guide the prover through larger steps. Even "automatic" theorem provers need some human guidance or control in the construction and checking of proofs. Some perform deduction steps on direct instructions from the user (PVS [ORS92] is like this); others receive guidance indirectly, either through the order and selection of results they are invited to consider (the Boyer-Moore prover [BM88] is like this), or in the form of a program that specifies the proof strategy to be used (the "tactics" of LCF-style provers [GMW79] such as HOL [GM93] are like this), or through the settings of various "knobs and dials" that control search parameters (Otter [McC90] is like this). In skilled hands, any of these forms of control can be effective, although direct instruction is the easiest to understand (provided the repertoire of basic steps is not too large).

A large verification often decomposes into smaller parts that are very similar to each other. For example, the top-level verification of a microprocessor design generally divides into a case analysis that separately considers each instruction in the order code, and many of these instructions will have much in common with each other. Similarly, a large part of the verification of the security or partitioning provided by an operating system reduces to showing that each operation preserves certain properties. In these cases it is useful if the user can develop a customized proof control "strategy" that can automate the repetitive elements of the proof. Methods inspired by LCF-style tactics can do this very effectively.

Presentation: Formal verification may be undertaken for a variety of purposes; the "presentation" phase is the one in which the chosen purpose is satisfied. For example, one important purpose is to provide evidence to be considered in certifying that a system is fit for its intended application. I do not believe the mere fact that certain properties have been formally verified should constitute grounds for certification; the *content* of the verification should be examined, and human judgment brought to bear. This means that one product of verification must be a genuine proof—that is a chain of argument that will convince a human reviewer. It is this proof that distills the insight into why a certain design does its job, and it is this proof that we will need to examine if we subsequently wish to change the design or its requirements. Many powerful theorem-proving techniques (for example, resolution) work in ways that do not lend themselves to the extraction of a readable proof, and are unattractive on this count.

Generalization and Maintenance: Designs are seldom static; user requirements may change with time, as may the interfaces and services provided by other components of the overall system. A verification may therefore need to be revisited periodically and adjusted in the light of changes, or explored in order to predict the consequences of proposed changes. Thus, in addition to the human-readable proof, a second product of formal verification should be a description that guides the theorem prover to repeat the verification without human guidance. This proof description should be robust—describing a strategy rather than a line-by-line argument—so that small changes to the specification of lemmas will not derail it. The easy rerunning of proofs is essential in order that incorporation of improvements and adaptation to changed requirements or environment may be assisted, not discouraged, by investment in an existing verification.

In addition to the modifications and adjustments that may be made to accommodate changes in the original application, another class of modifications—generalizations—may be made in order to support reuse in future applications, or to distill general principles. For example, we may extract and generalize some part of the specification as a reusable and verified component to be stored in a library.

The conclusion I draw from this analysis is that although there is much that must be learned from the specification language and the theorem-prover lines of development, truly effective verification systems require careful compromise between language and theorem-proving conveniences and a tight integration between language and prover. This is the path of the integrated verification systems—a line of development whose early examples were comprehensively inadequate. Later examples such as Eves [CKM+91], HOL [GM93], Imps [FGT93], Larch [GH93b], Nuprl [CAB+86], PVS [ORS92], and Veritas [HDL89] have achieved much more sophisticated integration of capabilities and one or two of these systems have been proved effective on large or difficult problems. I believe that the next generation of integrated verification systems will provide truly productive environments for Level 3 formal methods.

This assessment of the classes of verification systems and their development is not intended to discourage potential users of this technology, nor to suggest that only certain approaches and systems are worthwhile; rather, it is intended to encourage a realistic appraisal of the likely capabilities, strengths, and weaknesses of the various approaches. Almost all tools for formal methods can be and have been used to undertake worthwhile activities. But successful application of any tool requires an appreciation of its capabilities. Using one of the tools for a rich specification language to support a Level 2 application could be very successful; using it for heavy theorem-proving exercise will lead to disappointment. Conversely, a theorem prover for a raw logic might dispatch a verification very efficiently, but it might not be easy for others to understand what it is that has been verified.

Unfortunately, there are no agreed benchmarks for evaluating verification systems and their theorem provers, nor are there many cases of the same exercises being undertaken in more than one system, so potential users must evaluate different systems very much on their own. The evaluation should focus on the needs of the type and the scale of application concerned: small demonstration examples and tutorial exercises can give very misleading impressions of the capabilities required to undertake larger examples.

Finally, note that not all applications of formal methods require a general-purpose support tool; for some purposes, a state-exploration system, or an executable specification language, or an environment similar to a CASE tool with searching and cross-reference facilities may be all that is required.

2.7 Industrial Applications of Formal Methods

Formal methods can contribute to the development of systems in two ways (and ideally both together):

1. They can improve the development process, leading to a better product, and/or reduced time and cost, and

2. They can contribute to the assurance and certification of a system.

Most uses of formal methods that claim success in delivering the first of these benefits have used primarily Level 1 or 2 formal methods in the early lifecycle; those that have aimed at the second have been primarily late-lifecycle applications of Level 3 formal methods. In my opinion, these characteristics reflect accidents of history, rather than guides to future action. In this section I briefly outline a few of the documented applications and point the reader toward the articles that describe them in detail.

A good introduction to Level 1 and 2 formal methods in industry is the paper by Hall [Hal90], which is one of the very best nontechnical papers available on formal methods. Hall cites a number of successful applications of the Z notation. One of these, which appears in the same issue of the journal as Hall's paper, describes use of formal methods to model oscilloscopes at Tektronix [DG90]. Unlike most uses of formal methods, this exercise was not concerned with a specific design, but with gaining insight into system architectures for electronic instruments. Informally checked proofs were used to probe and challenge the formal models that were developed. Although it is not described in the paper, it seems that the real benefit obtained from this exercise was the capture and explicit documentation of much scattered experience and wisdom concerning system design for oscilloscopes (a process called "asset capture"). Tektronix apparently regards the product of this exercise as having outstanding proprietary value.

Perhaps the most widely known successful commercial application of formal methods is IBM's use of Z in the development of a major new release of its transaction processing system CICS [HK91]. The motivations for the use of formal methods were to improve the quality of the product and to reduce development costs, primarily by reducing the overall number of faults and by detecting the remaining faults as early as possible in the development cycle. The new release contained 500,000 lines of unmodified code, and 268,000 lines of modified code, of which 37,000 lines were produced from Z specifications and designs, and a further 11,000 were partially specified in Z. About 2,000 pages of formal specifications were produced. The formally specified portion of the system had fewer faults at all development stages except the very first, and a higher proportion of its faults were detected in the earlier stages, compared with the portion of the system that had not been formally specified. The numbers of problems per line of code reported by customers following release were in the approximate ratio 2: 5 for the formally specified and nonformally specified portions of the system. It was estimated that use of formal methods reduced costs for that portion by 9%. The formal specifications were subjected to formal inspections. A tool for parsing, typechecking, and cross-referencing Z specifications became available halfway through the project and markedly increased the productivity of the specifiers and inspectors.

Other similar examples include the use of formal methods by the semiconductor manufacturer Inmos in the development of a floating point unit for the T800 Transputer [Bar89a]. It was estimated that use of formal methods enabled the chip to be developed in less than half the time that would have been required for traditional methods. Furthermore, faults were found in the IEEE floating point standard and in other implementations used for back-to-back testing. Industrial applications of the Cleanroom methodology are described by Dyer [Dye92], while industrial use of VDM is considered by Plat [Pla93].

Fenton [Fen93] excoriates the claims of improved quality and reduced cost made in many of the references cited above, and also the attribution of these to formal methods. He argues that the data collected are so poorly controlled that they provide anecdotal evidence, at best, for the efficacy of the methods concerned. For example, skeptics might wonder whether the formal inspection process (or the fact that the developers were working on a second version of the system) might have contributed as much to the success of the CICS project as the use of formal methods.

A rather different criticism of claims made on behalf of formal methods concerns the British VIPER microprocessor. The controversy surrounding this was described in Section 2.5.2, but the salient point is that although the person who performed the HOL verification was scrupulously careful to explain what had been accomplished [Coh89b], some of the marketing material for VIPER made rather extravagant claims, suggesting that the device had a complete formal verification from the specification of its external behavior down to its gate-level design. This was rightly condemned in a report on the verification of VIPER commissioned by NASA [BH90, BH91].

The VIPER story repeats, to an uncanny degree, some of the failings of the SIFT project of a decade earlier. SIFT (Software Implemented Fault Tolerance) was originally a systems design and implementation project whose goal was to develop a fault-tolerant computer of sufficient reliability for fly-by-wire passenger aircraft [WLG+78, Gol87]. Prior to SIFT, the units of redundancy in fault-tolerant systems were generally components or functional blocks, not full computer channels (the IBM Launch Vehicle Digital Computer for the Saturn V rocket [Tom87, page 212] is a classic example). SIFT took a radically different approach: running identical software in separate, synchronized channels, and subjecting the results to majority voting in order to mask faulty channels. While attempting to develop the justification for the SIFT approach, Pease, Shostak, and Lamport (exploring intuitions described by NASA personnel) realized that there were fault modes in the distribution of single source data that could not be overcome by simple majority voting. Thus was the whole field of Byzantine agreement born [PSL80, LSP82].

When SIFT was well underway, a new project was started to apply formal verification to some aspects of the SIFT architecture. After several years, a formal specification and verification of some aspects of the fault-tolerance mechanisms of an abstraction of the SIFT architecture were developed. Another piece of work verified some pieces of code that resembled (small) parts of the SIFT operating system. However, the design and construction of the SIFT system, and of its code, were conducted entirely separately from its verification. What ran was not what was verified. It is not clear what relationship, if any, existed between the actual SIFT system and the abstractions that were subjected to verification. The papers that described the SIFT verification did not make this clear: they gave the impression that what had been verified was an actual flight-control system: "the formal proof, that a SIFT system in a 'safe' state operates correctly despite the presence of arbitrary faults, has been completed all the way from the most abstract specification to the Pascal program" [MSS82]. A peer review convened by NASA to examine the substantiation for these claims found them unwarranted: "Many publications and conference presentations concerning SIFT appear to have misrepresented the accomplishments of the project. In many cases, the misrepresentation stems from omission of facts rather than from inclusion of falsehood... Several panel members expressed serious concerns regarding both the possible effects of such misunderstandings on those seeking to apply the work and the reactions that might follow the discovery that the work had been overstated." [NAS83, page 24].

> "*Those who cannot remember the past are condemned to repeat it*"
> [George Santayana]

One would hope that the lessons of SIFT and VIPER would remove any temptation to overstate the accomplishments of formal methods. Yet even in 1992 one could find a paper bearing the title "Formal Verification of a Fault Tolerant Computer" [BJ92] that describes work which, whatever its merits, amounts to rather less than is claimed by its title. In my opinion, the enthusiasm of the formal methods community for its own accomplishments should be tempered by a rather greater respect for the scale of the challenges faced by those who develop and certify real safety-critical systems.

Formal methods in support of certification have been mainly undertaken in the United States and in order to qualify for the government's highest (A1) rating for secure computer systems [DoD85]. The criteria against which these evaluations are performed were drawn up in the early 1980s and based on concepts for secure systems that were a good deal older than that [BL76]. The criteria require Level 3 verification using either Gypsy [GAS89] or FDM [LSSE80, SS84] (HDM [Fei80, RLS79] and its variants [HOPW87] were also used in the early days)—all tools with their roots in the 1970s. The tools, evaluation criteria, and contractual mechanisms that were imposed sometimes had the effect of divorcing the formal specification and verification activities from the main development efforts. Nonetheless, several substantial projects were completed, and some significant flaws were apparently detected and corrected by these means [Sil83, Lin88, Wei88]. Some of these verifications were quite large: the proof logs for 461 lemmas undertaken for one project totalled nearly 18 million characters and would have occupied 7,600 pages if printed out [DGK$^+$90]. The verification of a secure system called "Multinet Gateway" is one of the case studies described in a recent survey of industrial applications of formal methods [CGR93b] (summaries appear in [CGR93a, GCR93]). In the following paragraphs I will mention three other cases covered in that study; readers should consult the study report for more details.

Darlington is a nuclear power station under development on a site east of Toronto. Each reactor has two independent shutdown systems, which are among the first such systems to employ software. When the plant was almost ready to come on line, the regulatory body (Atomic Energy Control Board of Canada) became very concerned about assurance for the shutdown software. Eventually, this led to retroactive application of formal methods in order to document and analyze the shutdown code [AHW+90]. The methods used were derived from the A7 methodology [H+78], and were performed almost entirely by hand. (A spreadsheet was used to record the "function tables.") Parnas [Par93] exhibits some simple theorems that are "more difficult than the majority of the theorems that arose in the documentation and inspection of the Darlington Nuclear Plant Shutdown Systems." (See [RS93] for automated proofs of these theorems.) He also exhibits the function tables that gave rise to those example theorems and states that the Darlington exercise "resulted in about 40 kg. of such trivial tables." The overall cost of the Darlington shutdown computer systems (including hardware) was about $16 million (Canadian). Of that total, 25% was spent on the verification effort. The code concerned was 1,362 lines of Fortran and 1,185 lines of assembler. Being required to delay operation of the plant and to undertake a formal methods effort unexpectedly at a time when interest charges were costing the project $20 million a month was "upsetting to the nuclear industry" and "resulted in considerable unfavorable publicity to the project as well as enormous additional cost" [Cra92].

A rather happier "reverse-engineering" of formal specifications occurred on the SACEM project for one line of the commuter train network in Paris [GH90]. Initially, pre- and post- conditions and loop invariants were derived for the program code from its informal design documentation, and Hoare-style code verification was performed by hand. Later, an after-the-fact formal specification (it is called a "formal re-expression") was constructed in order to validate the theorems that had been proved. Review of the formal specification "was not particularly easy because some of the best signaling experts were not able to read mathematical material...for this reason the re-expression team rewrites separately and directly from the formal specification a complete natural language specification. It is this last specification which has been checked by [signaling] experts...doing formal specifications forced the specification teams to clarify everything and to go deeper into the real nature of the problem to be solved" [GH90, page 191]. The total code was 21,000 lines of Modula-2, of which 63% were considered safety-critical and subjected to formal specification and verification. The V&V activity for SACEM required 315,000 hours of effort (i.e., rather more than 150 man-years, which was about 1.5 times the effort to develop the system), of which formal proof consumed 32.4%, module testing 20.1%, functional testing 25.9%, and formal re-expression 21.6%. For the on-board component of the system, 111 of 180 procedures were formally proved, while for the trackside component 21 of 167 procedures were formally proved. The developers of SACEM have subsequently refined their formal development methodology (it is derived from the "B Method" of Jean-Raymond Abrial [ALN+91]) and have partial automated some of the proof-generation and checking [Cha92]. The method is being applied to a project for the Calcutta subway and is being proposed for a new driverless train for the Paris Metro.

Another project where formal specification has apparently led to improved understanding concerns the Traffic Alert and Collision Avoidance System (TCAS II), which is required to be installed on all airline aircraft with more than 30 seats by 31 December 1993. The purpose of the TCAS family of instruments (of which TCAS II is a member) is to reduce the risk of midair collisions between aircraft. As a result of the discovery of flaws in the "Minimum Operational Performance Standards" [RTCA89] which is essentially a pseudo-code description of TCAS II requirements, the RTCA formed a special committee (147A) to develop a new specification and, ultimately, commissioned Nancy Leveson (then at the University of California at Irvine, now at the University of Washington) to develop a formal specification. Leveson's specification is based loosely on statecharts [Har86]; it represents decision logic in a tabular manner and is carefully typeset and presented. In my terminology, it represents a sophisticated Level 1 formal method. Reviewers, who include engineers and pilots, consider the formal notation better than the pseudocode and certainly better than the English language specification that was developed in parallel. No tools were used in the development of the formal specification. Even without using analysis tools, there appears to have been a general increase in confidence and understanding of the TCAS requirements.

At the other extreme from TCAS are projects involving state-exploration methods—where powerful tools are of the essence. Protocol verification has been a fruitful target for these techniques; errors have been discovered in the very important and subtle cache-coherence protocols used in shared-store multiprocessors, and some have been formally verified in this way [DDHY92, CGH$^+$92, MS91]. AT&T's COSPAN system [HK90] has been used in the design of several significant VLSI devices, with major reductions in the effort required (one project that was estimated to require 30 man-years using conventional methods was completed in 6 using COSPAN). In addition, faults have been detected in certain protocols and devices that had been subjected to very thorough conventional analysis (one had been simulated for 13 months with no errors detected in the last 4; COSPAN found a "showstopper" fault almost immediately). Intel apparently uses a system similar to COSPAN to probe some of the trickier design problems in its advanced microprocessors. It is interesting to note that whereas conventional formal methods are often applied to simple problems such as sequential code and the datapaths of VLSI devices, state-exploration is often applied to the very hardest problems: protocols and other distributed algorithms, and to control-dominated devices.

Although hardly an industrial activity, the products of a research program sponsored by NASA Langley Research Center are worth mentioning here since they are among the few examples where formal methods have been applied to problems taken directly from avionics applications. The program has undertaken mechanically-checked formal verification of the basic algorithmic and architectural aspects of synchronous fault-tolerant systems [BCD91], including their global fault-masking and transient-recovery properties [DB92, Rus93], and the crucial clock-synchronization [RvH93, Sha92] and sensor-distribution [BY92, LR93a] algorithms and their implementations [SB91, SB92]. In addition, some Ada library routines have been subjected to mechanically-checked program verification [EH93]. Current work involves collaborative projects between organizations with specialist skill in formal methods (Computational Logic Incorporated, ORA, and SRI International) and several aerospace companies (Allied Signal, Boeing, Honeywell, IBM, and Rockwell-Collins).

Another documented example of formal methods applied to an avionics problem involved use of the LOTOS languages and tools to specify and model software for the Flight Warning Computer (FWC) of the Airbus 330 and 340 aircraft [GH93a]. The goal of the project, which was undertaken by Aerospatiale, INRIA, CEAT (the French certification authority), and British Aerospace, was to see whether formal processes could significantly reduce development costs for avionics software. The planned development process was to write the detailed design in an executable specification language, assess its correctness by formal verification, and then generate executable code directly from the formal design. Assuming this last step were performed by a verified compiler, unit test could be eliminated, thereby achieving large cost savings. The specification language chosen for the exercise was LOTOS, which is an ISO standard notation (based on process algebra with an algebraic datatype facility) intended for specification and analysis of telecommunications protocols [ISO88]. The FWC was chosen as a test case since it is representative of the type and complexity of many other airborne systems, and because it had already been implemented—thereby providing a base for comparison (and a debugged set of requirements).

The FWC is a distributed system composed of three Intel 80386 CPUs, with software programmed in Ada, PL/M, and assembler. Its purpose is to monitor inputs from several analog, discrete, and digital input sources, check them for validity (these are the functions of CPU 1), identify alarm conditions and post these to the Display Management Computer (these are the functions of CPU 2; essentially it simulates a large combinational Boolean circuit, reducing 11,000 possible signals to 2,000 possible alarms, and then executes “complex and imperative rules” to select and organize presentation of the alarms). CPU 3 is used to generate audible alarms and synthesized speech.

It was found that 95% of the design specification for the FWC comprised descriptions of data structures and sequential algorithms. The data structures concerned were those of a typical programming language (enumerations, records, arrays) plus "memories," which model the I/O and communications resources of the FWC. The datatype facility of LOTOS was inconvenient for these purposes, so a preprocessor was written to "compile" these structures into the recursive equational form required by LOTOS. (At least two other such preprocessors were later discovered to have been developed elsewhere in Europe). Similarly, the constructs required for the sequential algorithms executed by the CPUs were those of a typical programming language (`if`, `case`, `while`, `for` etc.) and, again, the facilities of LOTOS did not provide particularly convenient expression for these concepts (they had to be encoded in terms of guarded commands, recursion, and sequential composition).

The remaining 5% of the design specification concerned real-time aspects, and these could not be expressed in LOTOS since it lacks the concept of an *urgent action* (the nearest equivalent is nondeterministic, which is unacceptable in airborne applications). Consequently, the real-time aspects of the design were specified as timing diagrams (and would have been translated into Ada by hand if the experiment had been completed).

The design description for the FWC comprised over 27,500 lines of LOTOS (a total of 43,021 lines were written when the preliminary design is included). It was observed that the LOTOS specification for CPU 2 was approximately the same size and level of abstraction as the existing Ada code. However, about 30% greater effort was required to develop the LOTOS specifications than the Ada code (some of this was expended on tool development), and it took longer to debug the LOTOS specifications than the Ada code (since the LOTOS diagnostics were insufficiently explicit).

The LOTOS design specifications of the FWC were too large to be processed by the available formal verification tools. Consequently, the specification was validated by simulation and testing. A special-purpose program was written to check for nondeterminism (this seems to have been a simulator, driven by specific test data, rather than a model checker to detect all nondeterminism). C code was compiled from the LOTOS design specification and subjected to a number of tests; the validity of the output behavior was determined visually by an Aerospatiale engineer. Several faults were detected in the LOTOS specifications by this means. The C code generated from the specifications for CPUs 1 and 2 compiled into an object program of 2.5 Mbytes that was able to process a set of input data "in some minutes" on a SPARC 10. (The size and performance of the existing Ada code is not documented, but the frame rate of the FWC is stated as 480 msec.) Due to lack of time, CPU 3 was not analyzed.

It is argued that because of the particularly simple, deterministic, character of the computations performed by CPU 1 and CPU 2 (basically, straight-line computations), formal verification, simulation, and testing all coincide. Thus, "it was not necessary to use sophisticated verification tools ...which are intended to [sic] the verification of complex systems, where concurrency and nondeterminism play an important role."

The lessons of the industrial applications of formal methods described in this section are mixed. The applications where formal proofs were performed on detailed designs or code (Darlington and SACEM) were from industries with little prior experience of safety-critical software, and which lacked the established requirements and practices for stringent software quality assurance found in the aerospace industry.[58] It is debatable whether the formal methods concerned would add much to the assurance currently achieved for flight-critical software. The relevance of the relatively simple, small, sequential (and largely loop-free) systems of Darlington and SACEM to the far greater size and complexity of flight-control systems (which are generally measured in hundreds of thousands of lines of code and are replete with the challenges of fault tolerance and of distributed, concurrent processing) is also questionable.[59]

For similar reasons, the cost and quality improvements found in applications such as CICS are of limited relevance to airborne software: formal methods may improve the development process for commercial data processing, but evidence that would translate to increased assurance for safety-critical functions is slight.

The TCAS experiment seems more encouraging: well-chosen and well-presented formalized notations can apparently improve the review and inspection process (SACEM has similar elements). Experience with COSPAN and similar systems is also promising, suggesting that state-exploration methods can find bugs in complex systems that have gone undetected by other methods (similar experience is also reported by the NASA program using conventional theorem proving).

Finally, the experiment involving the Airbus 330/340 flight warning computer serves as a salutary lesson in the misapplication of formal methods: choosing an inappropriate goal, selecting unsuitable formal languages and techniques, targeting a problem too large for the available tools, or too simple for them to be of much value, are all good ways to vitiate any potential benefits of formal methods.

We should bear these lessons in mind when we consider the role of formal methods in validation of airborne systems.

[58] I understand that formal methods were required for Darlington only after the developers had been unwilling or unable to perform more traditional processes for software assurance to the satisfaction of the regulatory authorities.

[59] With few exceptions [AN89, CV91], digital railroad switching and train control does not use the redundant channels of flight-control systems (perhaps because there is a safe state: stop the train). Instead they use single processors with extensive self-checks, redundant computations, and error correcting codes (the terminology is of "vital" systems and "coded processors") that are intended to make for fail-safe operation [Mon92, Mar92, Pat92, Hol86, HB86, Rut90].

2.8 Further Reading

In this section I mention some books that may be useful for those who wish to learn more about formal methods and related topics.

Logic and discrete mathematics provide the basis for formal methods and I recommend the text by Gries and Schneider [GS93] as the best introduction to these topics for those who wish to use them as tools in computer science. Those who seek an introduction or refresher on the construction of traditional (i.e., not specifically formal) mathematical proofs are recommended to consult Solow [Sol90]. Lakatos [Lak76] provides a thoughtful examination of the role of proof in mathematics, much of which also applies to formal methods in computer science.

The standard notation and constructions of logic and discrete mathematics, and the traditional mathematical conception of proof, are sufficient to undertake formal methods with the rigor that I call "Level 1." Moving beyond this, to the second level of rigor, we find self-contained specification languages, and more structured treatments of proof obligations, and (sometimes) of proofs themselves. Unfortunately, I have found no books that can be recommended unreservedly as good general introductions to this level of formal methods. Many of the current texts on formal methods are from the United Kingdom and generally introduce formal methods in the style of Z or VDM with no discussion of alternatives or drawbacks. They also tend to stress data structures and applications that resemble data processing; they are weak on the kinds of problems that give most concern in safety-critical systems, fail to address adequately the need to validate complex specifications, and make little reference to automation and mechanized proof checking. With those caveats, the texts by Ince [Inc88] and by Woodcock and Loomes [WL89] can be recommended as introductions to the use of mathematical concepts in specification, in the style of VDM and Z, respectively, while Cohen, Harwood, and Jackson give a brief and fairly general overview of formal specification techniques [CHJ86].

A book explicitly intended as an introduction to Z is that by Wordsworth [Wor92], while Spivey's texts are the standard descriptions of Z [Spi93, Spi88]. A collection of case studies undertaken in Z has been published by Hayes [Hay93] (a new edition is due shortly), and a more eclectic collection (using other notations such as LOTOS) by Scharbach [Sch89]. VDM is introduced by Jones [Jon86, Jon90], and the somewhat related RAISE languages and tools by George and colleagues [RAISE92].

Formal methods supported by automated tools, and especially those that provide proof checking or theorem proving for the third level of rigor, are generally described in books that are specific to individual systems. The HOL system, which is a theorem proving environment for higher-order logic, is described by Gordon and Melham [GM93]. LCF (the progenitor of HOL) is described by Paulson [Pau87], while the *mural* proof assistant (whose main application is VDM) is described in [JJLM91].

Returning to the United States, the ANNA annotation language for Ada is described in [Luc90], the Larch languages and tools in [GH93b], and the Cleanroom methodology in [Dye92]. The Nuprl system is described in [CAB+86], the Boyer-Moore prover in [BM79, BM88], and ONTIC in [McA89].

A modern introduction to the basics of theorem proving is given by Paulson [Pau92]; an older but still useful account is given by Bundy [Bun83]. Connections between specification and prototyping are explored by Hekmatpour and Ince [HI88] (using VDM), by Alexander and Jones [AJ90], and by Henderson [Hen93].

Many of the books cited above describe formal methods and tools that could be considered (by their authors at least) to be "ready for use." Those who wish to explore more widely should examine some of the classic texts, such as those of Dijkstra [Dij76] (the debate [Dij89] is particularly accessible) and Hoare [Hoa85, Hoa89] on mathematical approaches to computer programming, and should also seek out current research articles on the formal modeling of concurrency, real time, and fault tolerance.

Safety critical systems, with passing mention of formal methods, are considered by Pyle [Pyl91], and jet engine control, also with some mention of formal methods, by Thompson [Tho92]. A collection edited by Sennett [Sen89] provides more extended treatment of formal methods in the context of safety-critical systems. The compendium edited by McDermid [McD93] contains good short survey treatments of a number of relevant topics, including formal methods, background mathematics, safety, and management. All of these books are from the UK and reflect that perspective.

Those seeking a technical introduction to logic should examine Barwise [Bar78], Hodges [Hod83], or Ryan and Sadler [RS92]. Manna and Waldinger provide an introduction to logic that is explicitly intended for computer scientists [MW93]. Many other books on logic are mentioned in the Appendix.

Those considering the issues of transferring formal methods from research to practice may wish to consult Austin and Parkin [AP93], who analyze 126 questionnaires returned by organizations, mostly in the UK, on industrial use of formal methods. Good accounts of the general problems of technology transfer are given by Davis [Dav92] and by Potts [Pot93]. Systems engineering for computer-based systems is the focus of a recent study group report [WAH+93].

Chapter 3

Formal Methods and Digital Systems Validation

"*Are we just going to happily scale up from system sizes in which you can contemplate the whole system in front of you? A system that is small enough that you can contemplate it in your head and have some idea of what is going on in it. What happens as we go to systems that are beyond the easy scope of an individual imagination, beyond the easy scope of a single programmer running one single set of programs that that programmer can examine? What happens when it gets bigger than that?... We abstract certain things. We aggregate parts of it and say: 'Now we have a model.' When we analyze the model, we will in some sense understand and have analyzed the thing itself... What is the nature of verifiability when you say 'I have a model and it describes what the object will be like'—especially if it is a very complicated object so that testability is not a very easy thing to do? ... When we build things, how are we sure that the model we have abstracted from the reality... is a good enough model to enable us to find all those peculiarities? How are we going to do enough testing so we know that some of those peculiarities (that we think are well outside the operating range) don't turn out to be inside the operating range for some strange operating configuration somebody is bound to try when you have* 10^{11} *tries a year, and which you can't find when the test program has* 10^3 *or* 10^4 *tries per year... We can, I'm sure, succeed in finding statistical methods to assure us that on the whole, everything is all right. The bad cases are well out at the edge, and they are improbable; while in the region of the mean, or average behavior, everything is fine. I must remind you, however, that disaster does not occur in the mean. Disaster always occurs in the variance. That's where the problem is: where something unanticipated and unanticipatable happens.*" From a speech by former NASA Administrator Robert Frosch [Fro85].

The more severe classes of aircraft failure conditions must be extremely rare. In particular, system failures that could lead to a catastrophic failure condition must be "extremely improbable," which means that they must be "so unlikely that they are not anticipated to occur during the entire operational life of all airplanes of one type" [FAA88, paragraph 9.e(3)]. "When using quantitative analyses... numerical probabilities... on the order of 10^{-9} per flight-hour[1] may be used... as aids to engineering judgment... to... help determine compliance ... with the requirement for extremely improbable failure conditions" [FAA88, paragraph 10.b].[2]

Clearly, software components of systems that have the potential for catastrophic failure conditions must have low probabilities of anomalous behaviors, but this does not mean that the software need have a reliability of $1 - 10^{-9}$: it may have much lower reliability provided its failures do not contribute to serious failure conditions. However, it does mean that software failures or anomalies that can lead to serious failure conditions must be extremely rare, and that strong assurances must be provided that they are so. In this chapter, I consider how such assurances can be provided for software (and custom digital devices such as ASICs, which have similar characteristics to software), and examine the potential contributions of formal methods.

3.1 Background to Assurance

Requirements and considerations for certification of software in airborne systems are described in FAA Advisory Circular 25.1309-1A [FAA88] and in DO-178B [RTCA92], which is incorporated by reference into the former document (see [FAA93], part of which is reproduced in Section 3.2). However, in order to best understand the potential contributions of formal methods, it is first necessary to examine the intellectual basis for assurance of safety-critical software.

The general approach to development and certification of safety-critical systems is grounded in hazard analysis. Potential hazards (conditions that can lead to an accident) are identified and analyzed for risk (the combination of severity and likelihood). Unacceptable risks are eliminated or reduced by respecification of requirements, redesign, incorporation of safety features, or (as a last resort) incorporation of warning devices.

The criticality of a particular component or system is a measure of the severity of the possible effects that could follow from failure of that component or system.

[1] Justification for this number was given in the Note on Terminology on page 4.

[2] In Europe, certification is performed by the Joint Airworthiness Authority (JAA) which was set up in 1988 by the individual authorities of France, Germany, The Netherlands, and the United Kingdom. The requirements and documents of the JAA parallel those of the FAA. In particular, the material just quoted from FAA Advisory Circular 25.1309A [FAA88] is also present in the JAA "Advisory Material–Joint" document AMJ 25.1309.

Failure includes the possibility of performing functions incorrectly, or performing unintended functions, as well as the loss of intended functions. Design alternatives are explored in order to reduce the number of critical components and systems, and their degree of criticality. The degree of criticality associated with a particular component or system determines the degree of assurance that should be provided for it: a system whose failure could have grave consequences will be considered highly critical and will require very strong assurances that its failure will be extremely improbable.

Failures can be *random* or *systematic*; the former are due to latent manufacturing defects, wear-out and other effects of aging, environmental stress (e.g., single-event upsets caused by cosmic rays), and other degradation mechanisms that afflict hardware components, while the latter (which are sometimes called *generic* faults) are due to faults in the specification, design, or construction of the system. The probability of random failure in a system can be measured by sufficiently extensive and realistic testing, or (for suitably simple systems) it can be calculated from historical reliability data for its component devices and other known factors, such as environmental conditions. Classical fault-tolerance mechanisms (e.g., n-modular redundancy, stand-by spares, back-up systems) can be used to reduce the probability of system failure due to random component failures to acceptable levels, though at some cost in system complexity—which is itself a potential source of (systematic) design faults.

Systematic failures are not random: faults in specification, design, or construction will cause the system to fail under specific combinations of system state and input values, and the failure is *certain* whenever those combinations arise. But although systematic failures occur in specific circumstances, occurrences of those circumstances are associated with a random process, namely, the sequence over time of inputs to the system.[3] Thus, the manifestations of systematic failures 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 it to exhibit a systematic failure is less than 10^{-9} per hour. Note that this probabilistic measure applies whether we are talking about system reliability or system safety; what changes is the definition of failure. For reliability, a failure is a departure from required or expected behavior, whereas for safety, failure is any behavior that constitutes a hazard to the continued safe operation of the airplane. This, apparently small, difference between the notions of reliability and safety nonetheless has a profound impact on techniques for achieving and assuring those properties.

First, although there may be many behaviors that constitute failure from the reliability point of view, there may be relatively few that constitute safety failures—

[3] Its *environment*—the states of the other systems with which it interacts—is considered among the inputs to a system.

especially of the higher severity classes. Thus, whereas reliability engineering seeks to improve the quality of the system in general, safety engineering may prefer to concentrate on the few specific failures that constitute major hazards; at the risk of reducing these complex issues almost to caricature, we could say that reliability tries to maximize the extent to which the system works well, while safety engineering tries to minimize the extent to which it can fail badly.

Second, techniques for improving reliability naturally deal first with the major sources of unreliability: that is, the most frequently encountered bugs get fixed first. There is a huge variation in the rate at which different faults lead to failure, and also in the severity of their consequences. Currit, Dyer, and Mills [CDM86] report data from major IBM systems showing that one third of the faults identified had a mean time to failure (MTTF) of over 5,000 years (and thus have an insignificant effect on overall MTTF), and a mere 2% of the faults accounted for 1,000 times more failures than the 60% of faults encountered least often. Reliability-based approaches would concentrate on detecting and removing the faults that contribute most to unreliability (indeed, the cited data are used by Currit, Dyer, and Mills to demonstrate that random testing would be 30 times more effective than structural testing in improving the reliability of the systems concerned). The most rarely encountered faults can therefore hide for a long while under a testing and repair regime aimed at improving reliability—but if just one or two rare faults could lead to catastrophic failure conditions, we could have a reliable but unsafe system.[4] Data cited by Hecht [Hec93] indicates that such rare faults may be the dominant cause of safety- and mission-critical failures.

Third, the reliability-engineering approach can lead to concentration on the reliability of individual components and functions, whereas some of the most serious safety failures have been traced to poorly understood top-level requirements and to unanticipated subsystem interactions, often in the presence of multiple failures (Leveson [Lev86] quotes some examples and, although it does not concern software, Perrow's classic study [Per84] is still worth examination).

Elements of both the reliability and safety engineering approaches are likely to be needed in most airborne systems: although a reliable system can be unsafe, an unreliable system is unlikely to be safe in these applications. (Primarily because there are few safe failure modes in flight-control applications. This can be contrasted with nuclear power generation, where a protection system that shuts the reactor down unnecessarily may be unreliable, but perfectly safe.)

Just as the subtly different characteristics of reliability and safety lead to differences in methods used to achieve those properties, so they also lead to differences in their methods of assurance. Both reliability and safety are measured in probabilistic

[4]With 500 deployed systems, a single serious fault with a MTTF of 5,000 years could provoke several catastrophic failure conditions over the lifetime of the fleet.

terms and can, in principle, be assessed by similar means. However, the numerical requirements for safety in airborne systems are orders of magnitude more stringent than those normally encountered for reliability. Systems designated "highly reliable" may be required to achieve failure rates in the range 10^{-3} to 10^{-6} per hour, whereas requirements for safety often stipulate failure rates in the range 10^{-7} to 10^{-12} per hour.[5] I will speak of required failure rates of 10^{-7} to 10^{-12} per hour as the "ultra-dependable" range, and will talk of such systems as "ultra-dependable." Bear in mind that these probabilities generally refer only to the incidence of safety-critical failures, and not to the general reliability of the systems concerned, and are assessed on complete systems—not just the software they contain.

The change in acceptable failure rates between highly-reliable and safety-critical systems has such a profound impact that it goes beyond a difference of degree and becomes a difference in kind: the reason being that it is generally impossible to experimentally validate failure rates as low as those stipulated for safety.

There are two ways to estimate the failure rate of a system: the experimental approach seeks to measure it directly in a test environment; the other approach tries to calculate it from the known or measured failure rates of its components, plus knowledge of its design or structure (Markov models are often used for this purpose).

The experimental approach faces two difficulties: first is the question of how accurately the test environment reproduces the circumstances that will be encountered in operation; second is the large number of tests required. If we are looking for very rare failures, it will be necessary to subject the system to "all up" tests in a highly realistic test environment—installed in the real airplane, or very close facsimile (e.g., an "iron bird"), with the same sensors and actuators as will be used in flight. Furthermore, it will clearly be necessary to subject the system to very large numbers of tests (just how large a number is a topic I will come to shortly)—and if we are dealing with a control system, then a test input is not a single event, but a whole trajectory of inputs that drives the system through many states.[6] And since we are dealing with a component of a larger system, it will be necessary to conduct tests under conditions of single and multiple failures of components that

---

[5]Nuclear protection systems require a probability of failure on demand of less than 10^{-4} [LS93]; failures that could contribute to a major failure condition in an aircraft require a failure rate less than 10^{-5} per hour [FAA88, paragraph 10.b(2)]; the Advanced Automation System for Air Traffic Control has a requirement for less than 3 seconds unavailability per year (about 10^{-7}) [CDD90]; failures that could contribute to a catastrophic failure condition in an aircraft require a failure rate less than 10^{-9} per hour [FAA88, paragraph 10.b(3)]; controllers for urban trains must have failure rates lower than 10^{-12} [LS93].

[6]The key issue here is the extent to which the system accumulates state; systems that reinitialize themselves periodically can be tested using shorter trajectories than those that must run for long periods. For example, the clock-drift error that led to failure of Patriot missiles [GAO92b] required many hours of continuous operation to manifest itself in a way that was externally detectable.

interact with the system under test. Obviously, it will be very expensive to set up and run such a test environment, and very time-consuming to generate the large and complex sets of test inputs and fault injections required.

So how many tests will be required? Using both classical and Bayesian probabilistic approaches, it can be shown that if we want a median time to failure of n hours, then we need to see approximately n hours of failure-free operation under test [LS93].[7] So if we are concerned with catastrophic failure conditions, we will need to see 10^9 failure-free hours of operation under test. And 10^9 hours is a little over 114,000 years! [8]

To reduce the time under test, we could run several systems in parallel, and we could try "accelerated testing," in which the inputs to the system are fed in faster than real time and, if necessary, the system is run on faster hardware than that which will be used in actual operation. (This naturally raises questions on the realism of the test environment—particularly when one considers the delicacy of timing issues in control systems.[9]) But at best these will reduce the time required by only one or two orders of magnitude, and even the most wildly optimistic assumptions cannot bring the time needed on test within the realm of feasibility. Similarly, quibbles concerning the probability models used to derive the numbers cannot eliminate the gap of several orders of magnitude between the amount of testing required to determine ultra-dependable failure rates experimentally, and that which is feasible.

The analyses I have considered so far assume that no failures are encountered during validation tests; any failures will set back the validation process and lower our estimate of the failure rate achieved. "Reliability growth models" are statistical models that avoid the need to restart the reliability estimation process each time an error is detected and repaired; they allow operational reliability to be predicted from observations during system test, as bugs are being detected and repaired [MIO87]. But although they are effective in commercial software development, where only modest levels of reliability are required, reliability growth models are quite impractical for requirements in the ultra-dependable region. Apart from concerns about

[7] The Bayesian analysis shows that if we bring no prior belief to the problem, then following n hours of failure-free operation, there is a 50:50 chance that a further n hours will elapse before the first failure.

[8] Butler and Finelli [BF93] present a similar analysis and conclusion (see also Hamlet [Ham92]). Parnas, van Schouwen, and Kwan [PvSK90] use a slightly different model. They are concerned with estimating *trustworthiness*—the probability that software contains no potentially catastrophic flaws—but again the broad conclusion is the same.

[9] Its feasibility is also questionable given the fault injections that are needed to test the fault-tolerance mechanisms of safety-critical systems: experiments must allow a reasonable time to elapse after each injected fault to see if it leads to failure, and this limits the amount of speed-up that is possible.

the accuracy of the model employed,[10] a law of diminishing returns greatly lessens the benefit of reliability growth modeling when very high levels of reliability are required [LS93].

Since empirical quantification of software failure rates is infeasible in the ultra-dependable region, we might consider the alternative approach of calculating the overall failure rate from those of smaller components. To be feasible, this approach must require relatively modest reliabilities of the components (otherwise we cannot measure them); the components must fail independently, or very nearly so (otherwise we do not achieve the multiplicative effect required to deliver ultra-dependability from components of lesser dependability); and the interrelationships among the components must be simple (otherwise we cannot use reliability of the components to calculate that of the whole). Ordinary software structures do not have this last property: the components communicate freely and share state, so one failure can corrupt the operation of other components [PvSK90]. However, specialized fault-tolerant system structures have been proposed that seek to avoid these difficulties.

One such approach is "multiple-version dissimilar software" [RTCA92, Subsection 2.3.2] generally organized in the form of N-Version software [AL86, Avi85] or as "Recovery Blocks" [Ran75a]. The idea here is to use two or more independently developed software versions in conjunction with comparison or voting to avoid system failures due to systematic failures in individual software versions. For the N-Version technique to be effective, failures of the separate software versions must be almost independent of each other.[11] The difficulty is that since independence cannot be assumed (experiments indicate that coincident failures of different versions are not negligible [ECK$^+$91, KL86], and theoretical studies suggest that independent faults can produce correlated failures [EL85]—though the correlation can be negative [LM89]), the probability of coincident failures must be measured. But for this design approach to be effective, the incidence of coincident failures must be in the ultra-dependable region—and we are again faced with the infeasibility of experimental quantification of extremely rare events [BF93]. For these reasons, 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]. A further limitation on the utility of N-version

[10] Different reliability growth models often make very different predictions, and no single model is uniformly superior to the others; however, it is possible to determine which models are effective in a particular case, but only at modest reliability levels [BL92].

[11] For the Recovery Block technique to be effective, failure of the "Acceptance Test" must be almost independent of failures of the implementations comprising the body of the recovery block. The test and the body are intrinsically "more dissimilar" than N-Version components, which must all accomplish the same goal, but it is difficult to develop acceptance tests of the stringency required.

software is that, anticipating data presented in the next section, the most serious faults are generally observed in complex functions such as redundancy management and distributed coordination. These employ fault-tolerant algorithms that work under specific fault-hypotheses. For example, fault-tolerant sensor-distribution algorithms are based on very carefully chosen voting techniques, and plausible alternatives can fail [LR93b]. Supplying N implementations and additional voting does not necessarily make these functions more robust, but it certainly changes them and may violate the constraints and fault-hypotheses under which they work correctly. The daunting truth is that some of the core algorithms and architectural mechanisms in fault-tolerant systems are single points of failure: they just *have* to work correctly.

Another design technique suggested by the desire to achieve ultra-dependability through a combination of less-dependable components is use of unsynchronized channels with independent input sampling. Redundant computer channels (typically triplex or quadruplex) are required for fault tolerance with respect to random hardware failures in digital flight-control systems. The channels can operate either asynchronously, or synchronously. One advantage claimed for the asynchronous approach is that the separate channels will sample sensors at slightly different times and thereby obtain slightly different values [McG90]. Thus, even if one channel suffers a systematic failure, the others, operating on slightly different input values, may not. Like design diversity, effectiveness of this "data diversity" depends on failures exhibiting truly random behavior: in this case the requirement is that activations of faults should not cluster together in the input space. As with the independence assumption in design diversity, experimental evidence suggests that this property cannot simply be assumed (there is some evidence for "error crystals" [DF90]) and it must therefore be measured. And as before, experimental determination of this property is infeasible at the low fault densities required.[12]

If we cannot validate ultra-dependable software by experimental quantification of its failure rate, and we cannot make substantiated predictions about N-version or other combinations of less-dependable software components, there seems no alternative but to base certification at least partly on other factors, such as analysis of the design and construction of the software, examination of the life-cycle processes used in its development, operational experience gained with similar systems, and perhaps the qualifications of its developers.

[12]Like N-version software, asynchronous operation could also be proposed as a way to provide additional protection beyond that required and achieved by an individual software channel. This proposal overlooks the possibility that an asynchronous approach will complicate the overall design, having ramifications throughout the system, from fault detection, through reconfiguration, to the control laws. As the AFTI-F16 flight tests revealed (these are described below), this additional, unmastered complexity has become the primary source of failure in at least one system.

We might hope that if these "subjective" factors gave us a reasonable prior expectation of high dependability, then a comparatively modest run of failure-free tests would be sufficient to confirm ultra-dependability. Unfortunately, a Bayesian analysis shows that feasible time on test cannot confirm ultra-dependability, unless our prior belief is already one of ultra-dependability [LS93] (see also [MMN+92] for a detailed analysis of the probability of failure when testing reveals no failures). In other words, the requirement of ultra-dependability is so many orders of magnitude removed from the failure rates that can be experimentally determined in feasible time on test, that essentially *all* our assurance of ultra-dependability has to come from "subjective" factors such as examination of the lifecycle processes of its development, and review and analysis of the software itself.

This is a rather chastening conclusion: assurance of ultra-dependability has to come from scrutiny of the software and scrupulous attention to the processes of its creation; since we cannot measure "how well we've done" we instead look at "how hard we tried." This, in essence, is the burden of DO-178B (and most other guidelines and standards for safety-critical software, e.g., [IEC86, MOD91a]). Of course, extensive testing is still required, but it is perhaps best seen as serving to validate the assumptions that underpin the software design, and to corroborate the broad argument for its correctness, rather than as a validation of reliability claims. Indeed, most standards for safety-critical software state explicitly that probabilities are not assigned or assessed for software that is certified by examination of its development processes:

> "...it is not feasible to assess the number or kinds of software errors, if any, that may remain after the completion of system design, development, and test" [FAA88, paragraph 7.i].
>
> "Development of software to a software level does not imply the assignment of a failure rate for that software. Thus, software levels or software reliability rates based on software levels cannot be used by the system safety assessment process as can hardware failure rates" [RTCA92, Subsection 2.2.3].

(See also [MOD91b, paragraph 6.6 and Annex F].)

The infeasibility of experimental quantification of ultra-dependable software not only impacts the process of validating software, it also places constraints on the design of redundancy management mechanisms for tolerating hardware failures. Although the assumption of independent failures cannot be assumed for different software versions, it *is* a reasonable assumption for properly configured redundant hardware channels. Overall reliability of such a redundant system then depends on the failure rates of its components, and on properties of the architecture and implementation of the fault-tolerance mechanisms that tie it together (in particular, the coverage of its fault-detection mechanisms). The overall system reliability can be calculated using reliability models whose structure and transition probabilities are determined by hardware component reliabilities and by properties of the fault-tolerance mechanisms. These transition probabilities must either be calculated in some justifiable manner, or they must be measured: if they cannot be calculated or measured in feasible time on test, the architecture cannot be validated and its design must be revised.

This analysis motivates the methodology for fault-tolerant architectures known as "design for validation" [JB92], which is based on the following principles.

1. The system must be designed so that a complete and accurate reliability model can be constructed. All parameters of the model that cannot be deduced analytically must be measurable in feasible time under test.

2. The reliability model does not include transitions representing design faults; analytical arguments must be presented to show that design faults will not cause system failure.

3. Design tradeoffs are made in favor of designs that minimize the number of parameters that must be measured, and that simplify the analytic arguments.

Johnson and Butler [JB92] present a couple of representative examples to show how these principles might apply in practice. In one case, a dual-channel system is shown to be able to meet its mission reliability requirements provided the fault-tolerance software (which has to correctly recognize and shut down a faulty channel) has coverage greater than 0.9995.[13] Further analysis reveals that 20,000 tests will be required to validate the fault-tolerance software to the necessary level of statistical accuracy. If fault injections can be performed at the rate of one per minute, then 333 hours (a little over 14 days) of continuous time on test will be required. This is feasible, and so this attribute of the proposed architecture can indeed be validated.

The second example concerns a two-fault-tolerant 5-plex. In this case, it is shown that the coverage of the fault-tolerance mechanisms must exceed 0.9999982, and that over a million fault-injections will be required to validate satisfaction of this requirement. This amount of testing is probably infeasible (it is equivalent to 1.9 years on continuous test at one fault-injection per minute). We therefore have little alternative but to abandon this architecture in favor of one whose critical design parameters can be measured in feasible time.[14]

[13] What I am concerned with here is not design faults in the software, but the fact that it is (provably) not possible to always correctly identify the faulty channel in a pair. The effectiveness (or coverage) of a particular detection and diagnosis scheme generally has to be evaluated empirically.

[14] It might seem that we could enumerate all hardware failure modes and prove that the fault-tolerance mechanisms correctly counter each one. The difficulty is providing evidence (to the

To summarize the discussion so far: all software failures are of the systematic variety—there is nothing to go wrong but the processes of specification, design, and construction. Nonetheless, software failure can be treated as a random process and can be quantified probabilistically. However, validation of achieved failure rates by experimental quantification is infeasible in the ultra-dependable region. (This also places constraints on the design of fault-tolerant architectures, since system reliability models require accurately measured or calculated probabilities of coverage for the redundancy-management and fault-tolerance software.) The realization that experimental validation is infeasible for software in the ultra-dependable region means that its validation must derive chiefly from analysis of the software and from control and evaluation of its development processes. Thus, the goals of the very disciplined lifecycle processes required by almost all standards and guidelines for safety-critical software are to minimize the opportunities for introduction of faults into a design, and to maximize the likelihood and timeliness of detection and removal of the faults that do creep in. The means for achieving these goals are structured development methods, extensive documentation tracing all requirements and design decisions, and careful reviews, analyses, and tests. The more critical a piece of software, the more stringent will be the application of these means of control and assurance. In the next section, I briefly describe the particular form given to these general considerations in the RTCA Document DO-178B and also reproduce the sections of that document that address formal methods directly.

3.2 RTCA Document DO-178B

The RTCA ("Requirements and Technical Concepts for Aviation, Inc.") document known as DO-178B [RTCA92] (a revision to the earlier DO-178A) provides industry-accepted guidelines for meeting certification requirements for software used in airborne systems and equipment, and is incorporated by reference into European[15] and United States regulatory and advisory documents:

> "RTCA Document RTCA/DO-178B was developed to establish software considerations for developers, installers, and users when the aircraft equipment design is implemented using microcomputer techniques. It

required level of statistical significance) that all failure modes have been accounted for (the issue is called "assumption coverage" [Pow92]). There is an alternative, however: it is possible to prove that certain architectures can mask failures (up to a specified number), *no matter what the mode of failure* [HL91, KWFT88, K$^+$89, WLG$^+$78]. These architectures are based on Byzantine-resilient algorithms [LSP82, PSL80, Sch90], which are proved correct without making any assumptions about the behavior of failed components.

[15] DO-178B is known as EUROCAE ED-12B in Europe.

> is expected that current and future avionics designs will make extensive use of this technology. The RTCA document outlines verification, validation, documentation, and software configuration management and quality assurance disciplines to be used in microcomputer systems.
>
> "An applicant for a TSO (Technical Standard Order), TC (Type Certification), or STC (Supplemental Type Certification) for any electronic equipment or system employing digital computer technology may use the considerations outlined in RTCA document RTCA/DO-178B as a means, but not the only means, to secure FAA approval of the digital computer software. The FAA may publish advisory circulars for specific FAR's (Federal Aviation Regulations) outlining the relationship between the criticality of the software-based systems and the appropriate 'software level' as defined in RTCA/DO-178B. Those may differ from and will take precedence over the application of RTCA/DO-178B." [FAA93]

DO-178B is quite long (over 90 pages when the appendices and index are included) and densely packed with information. Here, I simply give an overview and explain how formal methods are admitted into its guidelines. Where I quote material from DO-178B, I present it in italics.

As noted in the FAA Advisory Circular quoted above, 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. This process is outlined in Sections 9 and 10 of DO-178B. I interleave paragraphs from these two sections since it seems to make for easier comprehension.

10.1. Certification Basis: *The certification authority establishes the certification basis for the aircraft or engine in consultation with the applicant. The certification basis defines the particular regulations together with any special conditions which may supplement the published regulations.*

9.1. Means of Compliance and Planning: *The applicant proposes a means of compliance that defines how the development of the airborne system or equipment will satisfy the certification basis. The Plan for Software Aspects of Certification defines the software aspects of the airborne system or equipment within the context of the proposed means of compliance. This plan also states the software level(s) as determined by the system safety assessment process. The applicant should*

...

c. *Obtain agreement with the certification authority on the Plan for Software Aspects of Certification.*

10.2. Software Aspects of Certification: *The certification authority assesses the Plan for Software Aspects of Certification for completeness and consistency with the means of compliance that was agreed upon to satisfy the certification basis. The certification authority satisfies itself that the software level(s) proposed by the applicant is consistent with the outputs of the system safety assessment process and other system life cycle data. The certification authority informs the applicant of any issues with the proposed software plans that need to be satisfied prior to certification authority agreement.*

9.2. Compliance Substantiation: *The applicant provides evidence that the software life cycle processes satisfy the software plans. Certification authority reviews may take place at the applicant's facilities or the applicant suppliers' facilities. This may involve discussion with the applicant or its suppliers.*[16] *The applicant arranges these reviews of the activities of the software life cycle processes and makes software life cycle data available as needed. The applicant should*

...

b. *Submit the Software Accomplishment Summary...to the certification authority.*

...

10.3. Compliance Determination: *Prior to certification, the certification authority determines that the aircraft or engine (including the software aspects of its systems or equipment) complies with the certification basis. For the software, this is accomplished by reviewing the Software Accomplishment Summary and evidence of compliance. The certification authority uses the Software Accomplishment Summary as an overview for the software aspects of certification.*

Section 10 of DO-178B also notes that

[16]The Federal Aviation Act authorizes the FAA to delegate certification activities to designated, FAA-approved employees of the manufacturer. These Designated Engineering Representatives (DERs) act as surrogates of the FAA in analyzing, testing, and examining aircraft designs and systems. FAA staff are responsible for overseeing DERs' activities and making the final determination as to whether a design meets FAA's safety requirements. Between 90 and 95% of all activities are currently delegated to Boeing and McDonnell Douglas DERs; delegation to DERs can be even greater in some software systems—for example, approval of the entire 747-400 flight management system was delegated to Boeing DERs because FAA staff "were not sufficiently familiar with the system to provide meaningful inputs to the testing requirements or to verify compliance with regulatory standards" [GAO93, pp. 17, 19, 20, 27]. For aircraft imported into the United States, the FAA relies on foreign authorities to conduct many of the necessary certification activities, but is responsible for certifying that the aircraft meet its requirements.

> "*the certification authority considers the software as part of the airborne system or equipment installed on the aircraft or engine; that is, the certification authority does not approve the software as a unique, stand-alone product.*"

Thus, the software must be considered in its relationship to the total system of which it forms a part. As the design for the overall system develops, it generates requirements for software components; and as the design of the software develops, so it feeds constraints and requirements back into the system development process. Among the important requirements and constraints that pass back and forth in this way are those for system safety. The relationship between system and software safety is discussed in Section 2 of DO-178B.

> "*The failure condition category of a system is established by considering the severity of failure conditions on the aircraft and its occupants*" [RTCA92, Section 2.2].

Failure condition categories are a five-point scale from "catastrophic" through "hazardous/severe-major," "major," and "minor" to "no effect."

> "*An error in software may cause a fault that contributes to a failure condition. Thus, the level of software integrity necessary for safe operation is related to the system failure conditions*" [RTCA92, Section 2.2].

Software levels are identified as Level A through E based on the severity of their potential failure conditions (i.e., Level A is software whose malfunction could contribute to a catastrophic failure condition). The software level determines the amount of effort and evidence required to show compliance with certification requirements [RTCA92, Subsection 2.2.2]; it does not imply the assignment of a reliability figure or failure rate to the software [RTCA92, Subsection 2.2.3]. The variations of processes and products by software level are tabulated in Annex A to DO-178B. These variations include the extent to which configuration management controls are imposed (lifecycle data in *Control Category 1* (CC1) are subject to full configuration management, those in *Control Category 2* (CC2) to a subset of the configuration management guidelines; the allocation of lifecycle data to CC1 or CC2 varies with software level), and the extent to which *independence* (the involvement of people other than the originators) is required during evaluations.

Section 3 of DO-178B discusses the processes of the software lifecycle. These processes are divided into three major categories: "planning," "development," and "integral" processes. The "planning" processes tie the development and integral processes together. The development processes include software requirements analysis, design, coding, and integration. The integral processes support the development processes by ensuring the correctness and quality of all processes and the delivered software. These processes comprise "software verification," "software configuration management," "software quality assurance," and "certification liaison." DO-178B discusses the planning process in its Section 4, and the development processes in Section 5. The integral processes are discussed in Sections 6 through 9, which cover verification, configuration management, software quality assurance, and certification liaison, respectively. Section 10 of DO-178B discusses airborne system certification (most of this section and a large part of Section 9 is reproduced above), and Section 11 describes software lifecycle data, which are produced to plan, direct, explain, define, record, and provide evidence of activities throughout the software lifecycle.

Section 6 of DO-178B makes a distinction between reviews and analyses that is pertinent when considering formal methods.

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.

...

6.2. Software Verification Process Activities: ...

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...*

...

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.* (An earlier draft 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.*") *A review may consist of an inspection of an output of a process guided by a checklist or similar aid. An analysis may examine in detail the functionality, performance, traceability and safety implications of a software component, and its relationship to other components within the airborne system or equipment.* (An earlier draft of DO-178B added "*An analysis may include the use of a formal proof of correctness*").

Section 12 of DO-178B discusses a variety of additional considerations. In particular, its Section 12.3 introduces "alternative methods," among which are included formal methods.

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 in* [sic] *this document.*

Alternative methods may be used to support one another. For example, formal methods may assist tool qualification, or a qualified tool may assist the use of formal methods.

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 life cycle 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.*

  (2) *The impact of the proposed method on the software life cycle data.*

  (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.

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.

Formal methods may be applied to software development processes with consideration of these factors:

Levels of the design refinement: *The use of formal methods begins by specifying software high-level requirements in a formal specification language and verifying by formal proofs that they satisfy system requirements, especially constraints on acceptable operation. The next lower level of requirements are then shown to satisfy the high-level requirements. Performing this process down to the Source Code provides evidence that the software satisfies system requirements. Application of formal methods can start and stop with any consecutive levels of the design refinement, providing evidence that those levels of requirements are specified correctly.*

Coverage of software requirements and software architecture: *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.*

These criteria can be used to determine the set of requirements at the level of the design refinement to which formal methods are applied.

Degree of rigor: *Formal methods include these increasingly rigorous levels:*

- *formal specification with no proofs.*
- *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.*

The three areas identified in the quotation above for possible tradeoffs in the application of formal methods (levels of the design hierarchy, coverage of software requirements and architectures, and degree of rigor) correspond to those discussed in Sections 2.1 and 2.2 of this Report. The degrees of rigor identified in DO-178B for the application of formal methods do not quite correspond to my Levels 1, 2, and 3: DO-178B does not consider methods corresponding to my Level 1 (general use of concepts and notation from discrete mathematics), and subdivides my Level 2 into its degrees of rigor 1 and 2, depending on whether manual proofs are performed; we agree on level 3. The reason I do not subdivide my Level 2 according to whether proofs are performed is that I attach little credibility (or utility) to specifications whose consequences have not been explored by proof. For safety-critical applications, I believe that my interpretation of Level 1 rigor (semi-formal mathematical notation and proofs) is preferable to DO-178B's (formal specifications and no proofs).

An earlier section of DO-178B considers the qualification of any tools used during system development—which could include those to support formal methods. As defined by DO-178B, such tools fall in the category of "Software Verification Tools" and the qualification criteria are primarily that the tools must be shown to perform as specified in normal operational use.

12.2. Tool Qualification: *Qualification of a tool is needed when processes of this document are eliminated, reduced, or automated by the use of a software tool without its output being verified... The use of software tools to automate activities of the software life cycle processes can help satisfy safety objectives insofar as they can enforce conformance with software development standards and use automatic checks.*

> *The objective of the tool qualification process is to ensure that the tool provides confidence at least equivalent to that of the process(es) eliminated, reduced, or automated.*
>
> ...
>
> *Software tools can be classified as one of two types:*
>
> ...
>
> **Software verification tools:** *Tools that cannot introduce errors, but may fail to detect them. For example, a static analyzer, that automates a*

> *software verification process activity should be qualified if the function that it performs is not verified by another activity. Type checkers, analysis tools and test tools are other examples.*

...

A tool may be qualified only for use on a specific system where the intention to use the tool is stated in the Plan for Software Aspects of Certification.

12.2.2. Qualification Criteria for Software Verification Tools *The qualification criteria for software verification tools should be achieved by demonstration that the tool complies with its Tool Operational Requirements under normal operational conditions.*

12.2.3.2. Tool Operational Requirements *Tool Operational Requirements describe the tool's operational functionality. This data should include:*

a. *A description of the tool's functions and technical features.*

b. *User information, such as installation guides and user manuals.*

c. *A description of the tool's operational environment.*

This excerpt concludes my overview of DO-178B. Before considering how formal methods might contribute to the objectives of that document, I next present a brief survey of experimental data on the efficacy of various software engineering and assurance methods, and of the historical data on faults and failures discovered in safety-critical systems. By knowing what seems to work, and what goes wrong, we may be able to decide how formal methods can best supplement current assurance methods.

3.3 Experimental and Historical Data on Assurance, Faults and Failures

A rather startling fact is that very little documented evidence attests to the efficacy of the various methods for software quality control and assurance when applied to safety-critical software.

There are, however, studies that indicate significant reductions in "defects per KSLOC" (i.e., programming faults per thousand lines of source code), compared with industry averages, when certain software engineering methods or techniques are employed. For example, Dyer describes a study performed by the Software Engineering Laboratory (SEL) of the Goddard Space Flight Center, in which a component of the ground support software for a satellite was developed using the

Cleanroom methodology. The component comprised 31,000 lines of Fortran and testing revealed a defect rate of 3.3 faults per KSLOC, compared with an SEL average of 6 faults per KSLOC [Dye92, page 34]. But although they seem relevant to our concerns, numbers such as these need to be interpreted with care, and on two counts: control and relevance.

Before we attribute reductions in defect-rate to specific processes, we need to be sure that all other factors are adequately controlled. Although researchers such as Basili and his colleagues [BW84, BSH86] have described sound methodological approaches to the gathering of data on software engineering processes, it is difficult and expensive to perform the necessary experiments, particularly on other than "toy" examples. Consequently, although Dyer's data may have been well-controlled, many of the other "successes" reported for various software engineering practices, including formal methods, must be considered to provide anecdotal rather than objective evidence for the efficacy of the practices concerned. For example, Vessey and Weber examined the empirical evidence in support of structured programming and found the results to be problematical and "a manifestation of poor theory, poor hypotheses, and poor methodology" [VW84, page 398]. Fenton [Fen93] is similarly critical of data cited in support of formal methods (including Cleanroom)—recall Section 2.7.

Leaving aside the question of experimental controls, we must also question the relevance to safety-critical systems of much of the data cited in support of formal methods. For example, the defect rate of 3.3 faults per KSLOC cited above is nowhere near that required for safety-critical applications. And even if there were objective evidence that certain practices reduce the density of defects in delivered software to extremely low levels, this would not necessarily guarantee fewer safety-critical failures in operation—for the relationship between density of faults and rate of serious failure is not straightforward (see, e.g., [Bis93]). It could be that certain practices are good at reducing the total number of faults, but are not specially effective on those that can lead to safety-critical failures (recall the earlier discussion of the differences between reliability and safety).[17] Thus, although there could be

[17] Recent work by Voas and others is pertinent here. Voas [Voa92] defines the *testability* of a program P as the probability that if P could fail, P will fail under test (given a particular testing strategy). Assessment of testability depends on a model of the relationship between faults and failures. Voas proposes a simple model (with admitted defects): each location in a program is considered to be the potential location of a fault. A possible fault can result in failure if and only if all the following conditions apply.

Execution: An input results in execution reaching the location of the fault.

Infection: The data state that results from execution of the fault is in error.

Propagation: The error leads to failure.

A program with low testability "hides its faults" in the sense that they are unlikely to come to light under test; this can be due to low probability of execution, infection, or propagation. Voas evidence that various methods are effective in quality *control* (i.e., in preventing, detecting, and eliminating faults during the development process), there seems little objective evidence to correlate these successes with any quantifiable level of quality *assurance*, especially for failure densities at the safety-critical level.

Since controlled experimental evidence is lacking that various software engineering techniques confer ultra-dependable levels of assurance, perhaps we should turn to the operational record. It can be argued that some deployed systems have demonstrated the desired levels of ultra-dependability in actual operation; for example, some Full-Authority Digital Engine Controls (FADECs) may be approaching 10^9 hours of operation—with no catastrophic failures.[18] Surely, following similar development practices should enable subsequent systems to achieve similar safety records. Laprie [Lap92] argues this point of view, observing that we are not certifying isolated products, but representatives of a continuing process with a measurable track record. There is much sense in this argument, but it is not totally convincing.

First, what does it mean to follow similar development practices? How do we know that certain unobserved and uncontrolled variables were not responsible for the quality achieved on those previous projects? The first project might have been particularly successful because it was the *first*, and attracted especially talented and motivated people.[19] We might retain or even improve the objective practices, yet miss the unknown *something* that separates success in the ultra-dependable region (10^{-9}, and the expectation of no catastrophic failures) from failure (10^{-8}, and the expectation of 5 catastrophic failures in the lifetime of the fleet). The point here is not to cast doubt on the seriousness and effectiveness of past and present practices, but simply to make it clear that we do not *know* in any quantifiable sense what the contribution of any particular method or practice is to the achievement, or to the assurance of achievement, of ultra-dependability.

proposes ways to estimate each of these factors separately. Hamlet and Voas [HV93] discuss ways to "amplify" reliability estimates derived from testing (i.e., ways ensure that the true reliability will be greater than that estimated), by taking testability into account. They also give plausible accounts why systematic test strategies (e.g., structural unit testing) may be superior to random testing in the ultra-dependable region, and caution that instead of amplifying reliability, formal methods may only reduce testability (i.e., rather than reducing the number of faults, formal methods may replace faults that are easily found in testing by those that are hard to find). Based on this analysis, they recommend two improvements that could be made to formal development methods (specifically, Cleanroom): behavior on erroneous and out-of-range inputs should be analyzed, and structural partition tests should be performed. In my opinion, this work is worth careful study.

[18] Digital autopilots have also been in use for many years. Up to the present, no airplane crash has been attributed to a software problem. However, "service experience showing that the failure mode has not yet occurred may be extensive, but it can never be enough" [FAA88, paragraph 7.g].

[19] Petroski's book subtitled "the role of failure in successful design" [Pet85] is interesting in this regard. It indicates that seldom does the first of a new kind of bridge structure collapse: disastrous bridge failures are usually the result of later developments that, often in ignorance of the rationale for some of the design decisions taken in the original, proceed to shave margins, or stretch capability, or make apparently small variations in the design without comprehending their consequences.

Second, the generation of airborne software that is approaching operational demonstration of ultra-dependability was developed a dozen or more years ago, and is vastly simpler in design and functionality than more recent systems. For example, current goals for primary flight computers (PFCs) include the ability to continue normal operation despite failed components (until regular scheduling brings the airplane to a maintenance base). One architecture developed to this requirement uses asynchronously operating triple-redundant lanes (each employing different hardware and programming languages[20]) within each of three synchronous channels [DH90]. Such elaborate architectures bear little resemblance to a dual-redundant FADEC. Elsewhere, there seems interest in shaving redundancy from quad to triple [SM92]. Previously isolated functions are being integrated,[21] and the sheer quantity of onboard software is much greater than previously (e.g., 20 Mbytes in the case of the Airbus A340 [GH93a]). All of these developments raise the scale of the challenge, and diminish the relevance of earlier projects.

Given the absence of experimental evidence on the efficacy of formal methods in safety-critical systems, let us turn instead to the fault and failure data for such systems and see whether it offers any guidance on the best uses, and likely benefits, of formal methods in quality control and assurance for safety-critical systems.

Considering quality control first of all, we can perhaps best seek guidance by examining documented critical failures or critical faults found late in the assurance process, and asking whether formal methods, or any other techniques, might have been effective in eliminating the introduction, or at least reliably detecting the presence, of the faults concerned. As usual, paucity of documented evidence hampers this exercise.[22] There are extensive data on failures in information-processing sys-

[20] This architecture has evolved into that used for the PFCs of the Boeing 777. The goal of programming the software independently and in a different programming language for each of the three lanes has been abandoned because it led to too many nuisance disagreements. Different processors and compilers continue to be used for each lane, but now only a single source program is used.

[21] For example, the MD-11 Automatic Flight System (AFS) provides "in addition to autoland and windshear functions,...Longitudinal Stability Augmentation Systems (LSAS) and roll control wheel steering when autopilot is disengaged; speed envelope protection via autothrottle or LSAS; yaw damper and turn coordination; elevator load feel and flap limiting control; attitude alert; stall warning with stick shaker and autoslat extend; automatic ground spoilers; wheel spin-up and horizontal stabilizer in motion detection; automatic throttle and engine trim via the FADEC; takeoff, cruise, and Cat II approach autopilot and flight director; in flight maintenance monitoring and ground maintenance functions interfacing with the central fault display system; annunciation, warning, and alert interfaces with the electronic instrumentation system" [DG93] (the perplexing punctuation is in the original).

[22] It is not only that the evidence may be considered proprietary and is therefore unpublished, it is not clear that some of the most vital data are even collected. For example, it seems that voting disagreement between redundant channels is not recorded in flight. Mandatory collection and publication of such data would be of considerable scientific value.

tems, but these have little similarity to critical airborne systems, and few studies identify the procedural or intellectual source of the faults that led to failure.

Fortunately, one paper does provide some relevant and interesting data: Lutz [Lut93a] reports on 387 software faults detected during integration and system testing of the Voyager and Galileo spacecraft.[23] 197 of the faults were characterized as having potentially significant or catastrophic effects (with respect to the spacecraft's missions). Lutz calls these 197 faults "safety-critical." Although spacecraft have obvious differences to airplanes, the embedded, real-time control software examined by Lutz may be expected to have many similarities with airborne systems, and the development practices at the Jet Propulsion Laboratory may also be considered representative of the aerospace industry (though bear in mind that Galileo was built a decade ago, and Voyager a decade before that).

Lutz classified the faults using a scheme introduced by Nakajo and Kume [NK91] that identifies the "human flaws" (root causes) as well as the "process flaws" responsible for the fault. Only 3 of the safety-critical faults found were programming mistakes, and very few problems attributable to faults in programming have occurred during flight. It seems that these faults are eliminated very effectively by the development processes employed. The remaining faults were divided approximately 3:1 overall between "function faults" (faults within a single software module) and interface faults (interactions with other modules or system components).

Two thirds of functional faults were attributed to flawed requirements; the remaining one third were due to incorrect implementation of requirements. Omissions constituted the majority of flawed requirements, the rest were due to imprecise, unsystematic, or wrong specifications of requirements. Incorrect implementation of requirements (i.e., faulty design or algorithms) tended to involve inherent technical complexity, rather than a failure to follow the letter of the requirements.

Half of all interface faults concerned misunderstood hardware interfaces. As is to be expected, the large majority of interface faults (93% on Voyager and 78% on Galileo) were attributed to poor communication between (as opposed to within) teams, primarily between software developers and system engineers. Particularly troublesome were undocumented or anomalous features of interfaces and of hardware devices that led software developers to make incorrect assumptions about their behavior.

Lutz' own recommendations for improving the development and assurance processes include: an early focus on internal interfaces, particular those between soft-

[23] Another interesting data point is provided by the Space Shuttle on-board flight-control software. This comprises about 500,000 lines of code, and has a good record: typically at most one or two defects are identified in each mission, and some have achieved their "zero-defect" goal. (Six faults that could have been life-threatening if activated are know to have flown, all of them prior to the Challenger disaster.) In contrast to this good record at the code level, there are over 400 "user notes" that document various kinds of anomalies in the requirements for the software.

ware and hardware; early identification of hazards; use of formal specification techniques; promotion of communication between teams, especially as requirements evolve; and use of defensive design techniques.

What specific contributions could formal methods make? It seems clear that developing and communicating requirements and assumptions (particularly concerning hardware behavior and interfaces) is the major problem. Use of a pseudocode based on a modern programming language such as Ada would allow the syntactic attributes of interfaces to be specified, but it appears that the real problems are associated with *semantic* attributes of the interfaces, such as initial states, timing and sequencing constraints, sampling frequencies, fault modes, and so on. Formal methods seem well suited to the task of documenting and modeling such properties of hardware interfaces.

Regarding missed requirements: rapid prototyping, animation, and other experimental approaches are advocated as ways to elicit complete and accurate requirements specifications. But while these methods seem appropriate for software with significant human interactions, they seem less so for embedded control systems. The proposals of Jaffe, Leveson, and Melhart [JL89, JLHM91] (recall Subsection 2.3.2) seem more apposite to this case, and require at least a semi-formal method of requirements specification.[24] Formal specification of requirements could also support formal challenges as a means of validation (similar to the "scenarios" generally employed today), and might allow more requirements to be stated in terms of constraints or invariants—which may be more robust and complete than imperative statements of required behavior.

Lutz' data identifies weaknesses in software quality *control*, but what does it suggest about *assurance*? One obvious, but important point is that if our primary assurance of ultra-dependability is to come from control and scrutiny of the software development process, then every fault that is discovered at a later stage of the lifecycle than that in which it was introduced casts doubt on the adequacy of all the intervening quality control processes. The greater the number of such faults that are discovered, and the greater the number of lifecycle stages that separate the comission and detection of each fault, the more seriously should the adequacy of the development and control processes be questioned.

The faults described by Lutz were detected during system and integration test; thus, though they were detected late, they were caught during what may be regarded as part of the normal software development process. I now turn to software failures in critical airborne systems detected during flight test: serious failures detected

[24] In recent work, Lutz [Lut93b] reports that a checklist derived from Jaffe, Leveson and Melhart's proposals would have identified 149 (i.e., over 75%) of the "safety-critical" faults examined in her study.

this late focus attention on the credibility of the processes employed for software assurance.

The flight tests of the experimental Advanced Fighter Technology Integration (AFTI) F16 were conducted by NASA, and are unusually well-documented [IRM84, Mac88]. The AFTI-F16 had a triple-redundant digital flight-control system (DFCS), with an analog backup. The DFCS had different control modes optimized for air-to-air combat and air-to-ground attack. The Stores Management System (SMS) was responsible for signaling requests for mode change to the DFCS. On flight test 15, an unknown failure in the SMS caused it to request mode changes 50 times a second. The DFCS could not keep up, but responded at a rate of 5 mode changes per second. The pilot reported that the aircraft felt like it was in turbulence; subsequent analysis showed that if the aircraft had been maneuvering at the time, the DFCS would have failed.

The DFCS of the AFTI-F16 employed an "asynchronous" design. In such designs, the redundant channels run fairly independently of each other: each computer samples sensors independently, evaluates the control laws independently, and sends its actuator commands to an averaging or selection component that drives the actuator concerned. Because the unsynchronized individual computers may sample sensors at slightly different times, they can obtain readings that differ quite appreciably from one another. The gain in the control laws can amplify these input differences to provide even larger differences in the results submitted to the output selection algorithm. During ground qualification of the AFTI-F16, it was found that these differences sometimes resulted in a channel being declared failed when no real failure had occurred [Mac84, p. 478]. Accordingly, a rather wide spread of values must be accepted by the threshold algorithms that determine whether sensor inputs and actuator outputs are to be considered "good." For example, the output thresholds of the AFTI-F16 were set at 15% plus the rate of change of the variable concerned; in addition, the gains in the control laws were reduced. This increases the latency for detection of faulty sensors and channels, and also allows a failing sensor to drag the value of any averaging functions quite a long way before it is excluded by the input selection threshold; at that point, the average will change with a thump that could have adverse effects on the handling of the aircraft [Mac88, Figure 20].

The danger of wide sensor selection thresholds is illustrated by a problem discovered in the X29A. This aircraft has three sources of air data: a nose probe and two side probes. The selection algorithm used the data from the nose probe, provided it was within some threshold of the data from both side probes. The threshold was large to accommodate position errors in certain flight modes. It was discovered in simulation that if the nose probe failed to zero at low speed, it would still be within the threshold of correct readings, causing the aircraft to become unstable and "depart." Although this fault was found in simulation, 162 flights had been at risk before it was detected [MA89a].

An even more serious shortcoming of asynchronous systems arises when the control laws contain decision points. Here, sensor noise and sampling skew may cause independent channels to take different paths at the decision points and to produce widely divergent outputs. This occurred on Flight 44 of the AFTI-F16 flight tests [Mac88, p. 44]. Each channel declared the others failed; the analog back-up was not selected because the simultaneous failure of two channels had not been anticipated and the aircraft was flown home on a single digital channel. Notice that all protective redundancy had been lost, and the aircraft was flown home in a mode for which it had not been designed—yet no hardware failure had occurred.

Another illustration is provided by a 3-second "departure" on Flight 36 of the AFTI-F16 flight tests, during which sideslip exceeded 20°, normal acceleration exceeded first −4g, then +7g, angle of attack went to −10°, then +20°, the aircraft rolled 360°, the vertical tail exceeded design load, all control surfaces were operating at rate limits, and failure indications were received from the hydraulics and canard actuators. The problem was traced to a fault in the control laws, but subsequent analysis showed that the side air-data probe was blanked by the canard at the high angle of attack and sideslip achieved during the excursion; the wide input threshold passed the incorrect value through, and different channels took different paths through the control laws. Analysis showed this would have caused complete failure of the DFCS and reversion to analog backup for several areas of the flight envelope [Mac88, pp. 41–42].

Several other difficulties and failure indications on the AFTI-F16 were traced to the same source: asynchronous operation allowing different channels to take different paths at certain selection points. The repair was to introduce voting at some of these "software switches."[25] In one particular case, repeated channel failure indications in flight were traced to a roll-axis software switch. It was decided to vote the switch (which, of course, required *ad hoc* synchronization) and extensive simulation and testing were performed on the changes necessary to achieve this. On the next flight, the problem was there still. Analysis showed that although the switch value was voted, it was the unvoted value that was used [Mac88, p. 38].[26]

The AFTI-F16 flight tests revealed numerous other problems of a similar nature. Summarizing, Mackall, the engineer who conducted the flight-test program, writes [Mac88, pp. 40–41]:

[25] The problems of channels diverging at decision points, and also the thumps caused as channels and sensors are excluded and later readmitted by averaging and selection algorithms, are sometimes minimized by modifying the control laws to ramp in and out more smoothly in these cases. However, modifying control laws can bring other problems in its train and raises further validation issues.

[26] This bug is an illuminating example. At first, it looks like programming slip—the sort of late-lifecycle fault that was earlier claimed to be very reliably eliminated by conventional V&V. Further thought, however, shows that it is really a manifestation of a serious design oversight in the early lifecycle (the requirement to synchronize channels at decision points in the control laws) that has been kludged late in lifecycle.

> "The criticality and number of anomalies discovered in flight and ground tests owing to design oversights are more significant than those anomalies caused by actual hardware failures or software errors.
>
> "...qualification of such a complex system as this, to some given level of reliability, is difficult ...[because] the number of test conditions becomes so large that conventional testing methods would require a decade for completion. The fault-tolerant design can also affect overall system reliability by being made too complex and by adding characteristics which are random in nature, creating an untestable design.
>
> "As the operational requirements of avionics systems increase, complexity increases...If the complexity is required, a method to make system designs more understandable, more visible, is needed.
>
> "...The asynchronous design of the [AFTI-F16] DFCS introduced a random, unpredictable characteristic into the system. The system became untestable in that testing for each of the possible time relationships between the computers was impossible. This random time relationship was a major contributor to the flight test anomalies. Adversely affecting testability and having only postulated benefits, asynchronous operation of the DFCS demonstrated the need to avoid random, unpredictable, and uncompensated design characteristics."

Clearly, much of Mackall's criticism is directed at the consequences of the asynchronous design of the AFTI-F16 DFCS. Beyond that, however, I think the really crucial point is that captured in the phrase "random, unpredictable characteristics." Surely, a system worthy of certification in the ultra-dependable region should have the opposite properties—should, in fact, be *predictable*: that is, it should be possible to achieve a comprehensive understanding of all its possible behaviors. What other basis for an "engineering judgment" that a system is fit for its purpose can there be, but a complete understanding of how the thing works and behaves? Furthermore, for the purpose of certification, that understanding must be communicated to others—if you understand why a thing works as it should, you can write it down, and others can see if they agree with you. Of course, writing down how something as complicated as how a fault-tolerant flight-control system works is a formidable task—and one that will only be feasible if the system is constructed on rational principles, with aggressive use of abstraction, layering, information-hiding, and any other technique that can advance the intellectual manageability of the task. This calls strongly for an architecture that promotes separation of concerns (whose lack seems to be the main weakness of asynchronous designs), and for a method of description that exposes the rationale for design decisions and that allows, in principle, the behavior of the system to be calculated (i.e., predicted or, in the limit, proved). It is, in my view, in satisfying this need for design descriptions which, in principle at least, would allow properties of the designs to be proved, that formal methods can make their strongest contribution to quality assurance for ultra-dependable systems: they address (as nothing else does) Mackall's plea for "a method to make system designs more understandable, more visible."

The AFTI-F16 flight tests are unusually well documented; I know of no other flight-control system for which comparable data are publicly available. However, press accounts and occasional technical articles reinforce the AFTI-F16 data by suggesting that timing, redundancy management, and coordination of replicated computing channels are tricky problems that are routinely debugged during flight test.

- During flight tests of the HiMAT remotely piloted vehicle, an anomaly occurred that resulted in the aircraft landing with its landing skids retracted. "The anomaly was caused by a timing change made in the ground-based system and the onboard software for uplinking the [landing] gear deployment command. This coupled with the on-board failure of one uplink receiver to cause the anomaly. The timing change was thoroughly tested with the on-board flight software for unfailed conditions. However, the flight software operated differently when an uplink failure was present" [MA89a, page 112].

- A significant software fault was discovered in flight testing the YC-14. "The error, which caused mistracking of the control-law computation in the three channels, was the result of incorrect use of cross-channel data for one parameter. Each synchro output was multiplied in software by a factor equal to the ratio of the nominal reference voltage to the actual reference voltage. Both the synchro outputs and the reference voltages were transmitted between channels, and the three inputs would be compensated in each channel prior to signal selection. However, because of an error in timing, each channel was using the current correction factor for its own sensor, whereas the correction factors for the other two sensors were from the previous frame. Thus, each channel performed signal selections on a different set of values, resulting in different selected input data for the three channels. Although the discrepancies were small, the effect of threshold detectors and integrators led to large mistracking between channels during flight. In the laboratory, the variations in the simulated synchro reference voltages were sufficiently small that this error would not be detected unless a bit-by-bit comparison between channels had been made" [MG78].

- In the 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]. It seems that either a potentially dangerous condition (i.e., divide-by-zero) had been present but undetected in the previous application, or it was known (and known not to be dangerous in that application) but undocumented. In either case, it seems to indicate inadequate assurance. This example also points to one of the perils of reuse: just because a component worked in a previous application, you cannot assume it will work in a new one unless *all* the relevant characteristics and assumptions are known and taken into account.

- The C17 has a quad-redundant digital flight-control system [KSQ92]. During the initial flight test of the C17 "On three occasions, warning/caution lights in the cockpit annunciated that flight-control computer (FCC) 'dropoffs' had occurred. ...FCC 3 dropped offline twice, and both FCC 3 and FCC 4 dropped off at the same time once" [Sco92]. For an account of software engineering and management on the C17, see [GAO92a].

One of the purposes of flight test is to uncover problems, and so the discovery of those just described can be considered a vindication of the value of flight test. Some might even consider these problems merely a matter of tuning, and regard their identification and repair during flight test as the proper course.[27] Others might argue the opposite point of view: flight test is for evaluating and tuning handling and controls, and the discovery of basic software problems indicates that the traditional methods of assurance are seriously deficient. Whatever view is taken of the seriousness of these problems, the salient fact seems to be that software problems discovered in flight test often concern redundancy management, coordination, and timing.

Overall, present development and assurance practices for aircraft seem to have worked so far: to my knowledge, there is no case of a software fault leading to a serious failure condition in a military or commercial airplane. There are several cases where a computerized system has contributed to an accident, but the responsible faults seem to have been in systems engineering and human-computer interfacing, rather than in software development.

For example, according to Frankfurter Allgemeine, Wednesday 10th November:

> "The computer-controlled Airbus A320 landing system was clearly one of the causes for the Warsaw aircraft accident on the 14th September,

[27] For example: "the FMS of the A320 'was still revealing software bugs until mid-January,' according to Gérard Guyot (Airbus test and development director). There was no particular type of bug in any particular function, he says. 'We just had a lot of flying to do in order to check it all out. Then suddenly it was working,' he says with a grin" [Lea88].

> concluded Lufthansa, which has instructed its pilots to employ a different landing procedure in certain [weather] conditions, said the airline in Frankfurt on Tuesday.
>
> "The Luftfahrtbundesamt said that the thrust reversers and spoilers of the Lufthansa Airbus were first activated after 9 seconds delay, 'because the logic of the airplane [design] doesn't allow otherwise.' ...The logic of the Airbus landing system requires that both main gear must be on the ground before reverse thrust and brakes are activated."

This and other press accounts indicate the following scenario. A shift in the wind caused the airplane to land fast and with one wing high. The pilots tried to apply the brakes but their input was overridden by the control logic of the software concerned. The airplane did not settle onto its second main landing gear (thereby allowing the brakes to be activated) for nine seconds, at which point it was still traveling at 154 knots—20 knots above normal landing speed—with only 1000 meters of runway left. The airplane ran off the end of the runway, collided with an earth embankment, and caught fire. Two people died. The A320 control logic requires the airplane to be settled on both main landing gear before the brakes, ground spoilers, and thrust reversers can be activated. Other airplanes use similar logic, but allow the ground spoilers to be deployed with only one main landing gear down, thereby dumping lift and hastening the settling of the other main landing gear.

Another example concerns the crash of an SAS MD-81 shortly after takeoff from Stockholm in December 1991 [Lea93].

> "An automatic engine-control function in the McDonnell Douglas MD-81, of which the operating airline was unaware, was a major factor in the Scandinavian Airlines System (SAS) accident near Stockholm, Sweden, in December 1991, says the official report into the accident.
>
> "SAS confirms the report's implication that, had the engine-control function not been incorporated in the aircraft's control software, or had the pilots known about its existence, the aircraft might have been recovered safely to Stockholm Arlanda Airport. In fact, it crash-landed in a field, with the right engine failed and the left stalled. Although the fuselage broke into three sections, no-one was killed.
>
> "The report confirms the well-established main cause of the accident as the failure of the airline to detect clear ice on the wings. The ice broke free from the wings just after takeoff and entered the engines '...causing damage to the engine fan stages, which led to engine surges which destroyed the engines,' says the report.
>
> "The connection with the engine-control system can be described by a simplified sequence of events:

- the ice entered the engines and damaged the fans;
- the right engine surged;
- the pilots reacted basically correctly to the surges by retarding the right throttle;
- the automatic thrust restoration (ATR) system caused both throttles to advance without the pilots noticing, making the surging worse in the right engine and starting surging in the left engine;
- the surges '...destroyed the engines,' says the report.

...

"The ATR is described by the accident report as a digital flight-guidance system function, conceived as a sensitive system for guarding against the accentuated dangers of single-engine failure just after takeoff at airports with strict noise-abatement procedures where the engines are throttled back...If a 7% difference in power is detected, the ATR automatically opens the 'good' engine to go-around power.

"McDonnell Douglas had removed the external controls for selecting or disarming the ATR from the SAS aircraft. The only way to disconnect the ATR was to disconnect the entire autothrottle system."

These accidents certainly seem to indicate flaws in systems engineering, in the requirements specified for functions assigned to software, and in the (non)provision of facilities for intervention by the pilots—but they do not indicate flaws in the processes of software development and assurance, nor failures in software itself. If we want to look at software failures in actual operation, it seems we have to look to fields other than aviation. There are, of course, numerous examples of egregious software failures (see [Neu92] for a partial list, and Wiener [Wie93] for an entertaining and generally balanced discussion of the dangers of excessive dependence on software), but few of these have arisen in fields that practice quality control and assurance to the levels used in commercial aviation.[28] Some examples from telecommunications, space, and missiles seem pertinent, however.

- The nationwide saturation of AT&T switching systems on 15 January, 1990 was due to a timing problem in a fault recovery mechanism [Tra90]; "ironically, this was a condition that spread throughout the network because of our

[28]The most horrifying consequences of software faults known to me are those of the Therac 25 medical electron accelerator, which led to massive overdoses of radiation and the subsequent deaths of six patients. However, as the account by Leveson and Turner [LT93] makes clear, no explicit software quality control and assurance program was employed in the development of that machine, and software engineering practices were rudimentary at best.

redundancy," said the AT&T Chairman, Robert Allen [MB90]. The problem affected all 114 of AT&T's 4ESS switching systems, and blocked about half of the long-distance, international, SDN (software-defined network), and toll-free 800 calls dialed on AT&T's public-switched network for a period of 9 hours.

- The first attempt to launch the Space Shuttle (STS-1) failed because the fifth (backup) on-board computer could not be synchronized with the main quad: "there was a very small, very improbable, very intricate, and very old *mistake* in the initialization logic of the primary avionics software system" [Gar81].

- Voyager spacecraft suffered 42 SEUs in the intense radiation surrounding Jupiter [Wil90]. The clocks lost synchronization and skewed 8 seconds, causing some scientific data to be lost. Clock synchronization was reprogrammed for Voyager 2's encounter with Neptune [Ano89].

- The Magellan spacecraft broke Earth lock and lost communications several times in August 1990 (soon after entering Venus orbit). It took over six months to identify the source of the problem, which was a timing error in the flight software.

- "On February 25, 1991, a Patriot missile defense system operating at Dharan, Saudi Arabia, during Operation Desert Storm failed to track and intercept an incoming Scud. This Scud subsequently hit an Army barracks, killing 28 Americans." The fault was in the way clock ticks were accumulated and converted to time. The conversion "results in a loss of precision causing a less accurate time calculation. The effect of this inaccuracy on the range gate's [i.e., target tracking] calculation is directly proportional to the target's velocity and the length of time the system has been running" [GAO92b].

As before, a common feature of these examples is that the faults arose in the context of redundancy management and timing-dependent coordination. Hecht [Hec93] offers some suggestions why this may be so: using data from a variety of sources, including the final testing of Release 8B of the Space Shuttle Avionics Software (this was the first release following the loss of Challenger), Hecht shows that safety- and mission-critical failures are associated disproportionately often with the occurrence of "rare events," and that the simultaneous (and unanticipated) arrival of two or more rare events seems the most common cause of severe failure. Rare events tend to be component failures and exceptions of various kinds, and so it is the redundancy management code that is stressed by these very unusual combinations of events.

The lessons of the data cited in this section seem to be that there is little hard experimental evidence that formal methods improve the quality of safety-critical software (though there is anecdotal evidence that they may do so), and that the most egregious and elusive faults in safety-critical systems are found in redundancy management, and in other elements of the software that coordinate distributed, concurrent, timing-dependent activities.

It seems to me that these two observations are not unrelated. Perhaps because it is feared that their techniques and tools will be overstretched by larger challenges, formal methods have usually been applied to relatively routine design problems, where traditional methods are already adequate. But the areas that seem to need most help are the hardest, most critical, and most fault-prone aspects of design—so if formal methods are to make a real contribution to quality *control* in safety-critical systems, this is where they should be put to work.

In addition, the quality *assurance* aspects of formal methods have not been fully exploited. The problem with current methods of assurance is that they merely reduce, and cannot eliminate, doubt. This is because assurance derives chiefly from *reviews*, which are a consensus process: they increase our confidence that certain faults are not present, but do not provide demonstrable evidence of that fact. Using formal methods, we can replace or supplement reviews with *analyses* that do provide repeatable evidence that certain kinds of faults are not present (at the stages of development considered). There are powerful caveats on such assertions, however, mainly concerning the fidelity of the mathematical modeling employed—so that formal methods cannot eliminate doubt any more than can traditional processes. But although they cannot *eliminate* doubt, formal methods can *circumscribe* it: with their aid, we can become fully confident in the correctness of a critical algorithm, say, and our residual doubts then focus on the fidelity of the assumptions employed, the interpretation of the property that has been verified, and the correctness of the implementation of the algorithm. Review processes can then be focussed on these residual doubts.

In the next section I expand on these points and consider use of formal methods in the context of DO-178B. In the section after that I discuss choice of the level of rigor and the selection of tools.

3.4 Formal Methods in Support of Certification

There seem to be two reasons why an applicant might consider using formal methods in support of certification: (1) to achieve the same level of quality control and assurance as by other means, but to derive some other benefit such as reduced cost; or (2) to provide a greater level of quality control and assurance than can be achieved by other means. I will consider these cases separately.

3.4.1 Use of Formal Methods for Reasons other than Improved Quality Control and Assurance

In this subsection, I assume that a developer has decided that formal methods offer some significant benefits, but that there is no intent to claim increased quality control or assurance through their use. Presumably, the main motivation will be to save time or money, and formal methods will be used either as a debugging technique that can quickly eliminate faulty requirements, specifications, designs, and implementations, or as a methodological aid that can systematize and render more reliable the process of development. A rather more aggressive use of formal methods to reduce development costs would be to substitute formal techniques for certain review and test processes. It is up to the developer to decide whether formal methods really can help achieve these improved efficiencies; here I consider only the impact that formal methods might have on the process of certification. I consider three increasingly ambitious applications of formal methods:

- To supplement traditional processes and documentation,
- To substitute formal specifications for some traditional documentation,
- To substitute formal proofs for some traditional reviews and analyses.

Formal methods as supplement to the traditional processes and documentation

In the most cautious form of this scenario, the standard development processes would continue to be performed (though more efficiently, since it is to be hoped that formal methods will ensure that less time and effort will be wasted on flawed constructions), and the standard lifecycle data would be produced. This use of formal methods is a strictly "internal" process, in that no external documentation of their use is submitted in support of certification. It seems to me that this mode of employing formal methods presents no challenges to certification; however, it seems a useful precursor to more aggressive uses of formal methods, since it is one way to gather "*evidence of the use of the method*" as required by Section 12.3 of DO-178B.

Formal specifications in place of some traditional documentation

A somewhat more aggressive use of formal methods would substitute formal specifications for some of the requirements and design documents required by DO-178B, but would retain traditional methods of review and analysis. This approach will require attention to the software requirements and software design standards required by Sections 11.6 and 11.7 of DO-178B. For example, standards are required for "*notations to be used to express requirements, such as data flow diagrams and formal specification languages*"[29] [RTCA92, Section 11.6.b].

It will also require attention to the "*guidance for using an alternative method*" documented in Section 12.3 of DO-178B (this was reproduced in Section 3.2 above). The most demanding of the guidance items requires a "*rationale for use of the alternative method which shows that the system safety objectives are satisfied*" [RTCA92, Section 12.3.b(3)].

I think what is required here is evidence that (1) the formal specifications contain at least the same information as the informal ones they are to replace, and (2) they support review, analysis, and other lifecycle processes as least as well as the informal specifications. Perhaps the most challenging of these will be to provide evidence that formal specifications can be reviewed as effectively as traditional specifications using natural language.

One approach, which was used in the SACEM example described in Section 2.7, finesses this problem by deriving a natural language description from the formal text and using the natural language version in the review process. This is really a retreat to the less aggressive use of formal methods that was discussed earlier, so let us now consider the case where the formal specification itself is used in reviews and other processes.

At bottom, the issue comes down to the extent to which those who write and review formal specifications can demonstrate mastery of the formal method or notation concerned. When the formalism is a relatively simple tabular representation of state machine transitions, as with Leveson's TCAS specification or Parnas' function tables for the Darlington shutdown system (both described in Section 2.7), then it seems plausible that all who need to do so can achieve adequate mastery of the notation (indeed, these notations were developed for just that purpose). Furthermore, it is an essential element of Parnas' method that certain systematic analyses are undertaken to check well-formedness of the tables (e.g., the conditions heading the columns of a table must be shown to exhaust all possibilities and to be pairwise disjoint).

Use of more elaborate formal specification languages for limited purposes, such as describing data structures (VDM is often used like this), should also present few challenges to those with programming experience. The difficult choices begin with

[29] The standards required are those for the project concerned: large aerospace projects establish very detailed standards for every facet of their operation. These project standards incorporate relevant regulatory and professional standards and guidelines, company policy, and a host of detailed prescriptions (extending, for example, to the naming of identifiers in programs). The requirement on standards for formal specification languages does not imply that any language used must have been standardized by some national or international body, but that there must be some agreed reference for the language, and guidelines for its use.

larger scale applications of Level 2 formal methods 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 the equivalent of "programming faults" that render them inconsistent or incorrect.

Those who learn a Level 2 formal specification language from textbooks or training courses, but who do not perform numerous proofs (preferably with automated checking), are in a very similar position to those who would learn a programming language by the same means and without the opportunity to execute the programs that they write—in fact, worse, since experience with other programming languages is likely to help them learn a new one, whereas many of those learning a 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 may be no easier than programming. In my experience, everyone has to learn this for themselves: only the personal shock of discovering egregious errors in our own specifications teaches us the requisite humility.

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. The 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 (e.g., 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.[30]

- Formal challenges (i.e., posing and proving putative theorems that should be entailed by the specification).

I have been surprised, and not a little shocked, to discover in some projects a reluctance to undertake some of these analyses (except animation). In one project, the authors of a specification gave up attempting to typecheck it once the typechecker revealed errors, yet they expected reviewers to study the specification. In another, the proof obligations generated by typechecking were not discharged by the author, but were examined during reviews; I consider it pointless to expend the intellectual resources available during a review on matters that can be decided (more reliably) by analytic processes.

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. If the means of analysis includes automated tools, then it will also be necessary to attend to the requirements of Section 12.2 (Tool Qualification) of DO178B (relevant excerpts were quoted earlier in Section 3.2).

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, we can ensure that the formal specifications submitted for review are free of gross defects and the reviews can focus on deeper issues. The question then 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 to 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 fully comprehend a formal specification may be enhanced if the suggestions of Parnas

[30] Duke [Duk89] documents a methodology used at NASA Dryden that apparently involves development of a working software prototype, extraction of requirements from that prototype, development of fully engineered software to those requirements, and iteration until both software versions and the requirements agree. Duke notes that this approach seems best suited to the development of experimental systems.

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).

My experience has been that engineers have little difficulty in learning to read formal specifications and are quite willing to do so once they see a payoff. What constitutes an adequate payoff seems to be evidence that you can *do* something with formal specifications—and one of the things that can be done is to eliminate most of the trivial errors that waste a good deal of time in reviews of informal specifications.

Another payoff is that mechanical analysis may discharge some of the requirements for certification. 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 non-conflict can be established mechanically for formal specifications by strong typechecking and related analyses.

Yet another payoff of formal specifications, but one that requires very skilled specifiers to achieve, is the ability to be precise without being excessively detailed. Using natural language and informal methods of expression, the desire to be precise often drives requirements authors into excessive levels of detail and to a notation close to that of a pseudocode—but then the requirements become descriptions of an implementation, and the rationale and intent behind them is lost.[31] This is a serious flaw, since requirements validation is primarily an examination of rationale and intent, and cannot be performed adequately if these are not recorded. Some current trends are exacerbating this problem. For example, requirements are often developed with the aid of simulators. In these circumstances, the system engineers sometimes cut and paste sections of their simulation program into the software requirements document: in effect, the simulation code becomes the requirements specification. Requirements validation then necessitates "reverse engineering" from the simulator code to the goals and constraints (i.e., to the "real" requirements)

[31] For example, Tomayko [Tom87, page 111] reports that C-Level requirements specifications for the Space Shuttle flight-control system "include descriptions of flight events for each major portion of the software, a structure chart of tasks to be done by the software during that major segment, a functional data flowchart, and, for each module, its name, calculations, and operations to be performed, and input and output lists of parameters, the latter already named and accompanied by a short definition, source, precision, and what units each are in. This is why one NASA manager said that 'you can't see the forest for the trees' in Level C, oriented as it is to the production of individual modules."

that it is asserted to satisfy. A requirements analyst on one project observed that the whole process then operates backwards: "the systems engineers write code, and the software developers have to do systems engineering."[32]

Formal methods could help reduce some of these problems: they offer the possibility of documenting requirements precisely, yet without implementation detail, and could also contribute to the documentation of rationale and intent (e.g., by allowing intended consequences to be stated as theorems, and invariants and constraints to be recorded as axioms or assumptions).

It requires considerable skill in formal methods to develop formal specifications of this kind—and it also requires deep understanding of the problem domain and of the lifecycle processes being followed. The last of these is sometimes overlooked when formal methods experts and domain specialists are left to develop the specification on their own: it is not enough for the specification be elegant, or even correct—it has to be appropriate for the part it is to play in the development process. This means it must be presented in a form that is suitable for review in the current stage of the lifecycle and that can be used as an input to the next. The requirements for these processes can range from formatting rules (e.g., lines must be numbered to facilitate inspections), through identifier-naming conventions, to onerous documentation standards. It seems that the diverse skills required for the development of effective formal specifications are best brought together through the collaboration of several individuals, each skilled in particular facets of the overall problem.

We have just considered formal specifications as a way to supplement or replace some of the *Software Requirements Data* described in Section 11.9 of DO-178B. Another opportunity is to apply formal methods to the *Design Description* required by DO-178B Section 11.10.

---

[32]Another disadvantage of these excessively detailed forms of requirements specification is that they close off implementation options. For example, one change to be introduced in the OI-24 version of the Space Shuttle on-board software is intended to ensure that reaction control jets are selected for firing in a way that minimizes the angle between the acceleration that will be produced by firing the selected jets and the direction desired (previously the aim had been to maximize the scalar product between the acceleration produced by firing the selected jets and the direction desired; the new scheme is intended to save propellant). The requirement is specified in pseudocode (probably derived from a simulator) that has triply nested loops. An implementation of the requirement as specified consumes 67 msec. of a frame in which only 40 msec. is available. Because the requirement specification mandates a triply nested loop, rather than specify the properties desired of the output of the computation, the implementors have little room for maneuver in reducing execution time. It is possible that a costly revision to the requirements will be needed, or that this attempt to reduce propellant usage will be abandoned.

In another example (found while performing formal verification), an optimization that could safely be left to implementation (i.e., don't recompute which jets to fire if none of the inputs have changed since the previous frame), is instead promoted to the requirements level, where it is expressed in an obscure manner that appears to admit the possibility of firing a failed jet (a very bad idea!) in certain circumstances.

11.10. Design Description: *The Design Description is a definition of the software architecture and the low-level requirements that will satisfy the software high-level requirements. This data should include:*

- **a.** *A detailed description of how the software satisfies the specified software high-level requirements, including algorithms, data structures, and how software requirements are allocated to processors and tasks.*

  ...

Here the rationale for formal methods would be to enhance communication among members of the design and implementation teams. Use of formal methods to specify (sequential) algorithms, data structures, and interfaces is fairly straightforward, and some widely publicized, and apparently successful applications of formal methods have been of this kind [HK91]. The concepts and notations employed can be closer to those of programming and digital design than is appropriate for requirements documentation and the consequences for certification seem slight.

Formal methods in place of some traditional processes

So far, I have envisaged that traditional reviews and analyses will continue to be performed, although some traditional data may be supplemented or replaced by formal specifications.

Larger savings may be possible if formal analyses can replace some of the traditional processes considered in certification. I have already suggested that formal methods of analysis such as typechecking could supplement or replace reviews for properties such as consistency of specifications. However, more significant savings might be possible if formal "proofs of correctness" could replace some of the onerous testing procedures required by DO-178B. (I have heard estimates that up to 30% of all software development costs may be consumed in unit testing and that it costs $100,000 for each bug found by structural unit test procedures—however, this latter figure is of little relevance, since the purpose of unit testing in DO-178B is assurance, not debugging.) The difficulty in doing this is the reason I put "correctness" in quotes: formal methods deal with models of the system, not the system itself. Consequently, some testing will be required to validate the modeling, though it is plausible that this could be accomplished as part of integration testing, without requiring unit testing. The Cleanroom methodology follows a very rigorous version of this approach: developers must provide correctness arguments for their code and are not allowed to run it; there are no unit tests, and integration tests use statistical sampling methods [Dye92].[33]

---

[33] In order to support statistical testing, Cleanroom developers have to document the expected "operational profile" of their software. Apparently, this extra requirement causes developers to

The realities of airplane certification are such that unit test criteria are unlikely to be relaxed in favor of formal verification or formal refinement without considerable evidence supporting a "*rationale for use of the alternative method which shows that the system safety objectives are satisfied*" required by Section 12.3 b(3) of DO-178B. Since one of the purposes of unit test is to exercise the compiled code[34] formal verification of source code, or formal refinement of specifications into source code, are unlikely to satisfy this safety objective (unless the compiler were verified).

However, it is feasible that formal methods could discharge one of the objectives of testing rather more convincingly than conventional tests, and this may be a niche that formal methods could occupy rather sooner than replacement of unit testing. According to DO-178B [RTCA92, Section 6.4]:

> "*Testing of airborne software has two complementary objectives. One objective is to demonstrate that the software satisfies its requirements. The second objective is to demonstrate with a high degree of confidence that errors which could lead to unacceptable failure conditions, as determined by the system safety assessment process, have been removed.*"

The second of these objectives is rather difficult to verify by testing—it means trying to demonstrate a negative. But if we can state precisely the properties that could lead to unacceptable failure conditions, then formal methods can be used to show the absence of those properties (we prove that the software entails the negation of those properties for all inputs). This approach is suggested in DO-178B itself [RTCA92, Section 12.3.1]:

> "*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).*"

One specific technique that addresses concern for "*inputs that are out of range*" is use of formal verification to prove that specific exceptions (e.g., array bounds violations) will not occur. Techniques of this kind can be considered a very strong form of typechecking; they were pioneered by German [Ger78] and are now available in commercial tools [Egg90]. Best and Cristian give a formal treatment of more general kinds of exceptions [BC81, Cri84].

---

think about their software from a novel perspective and has the unexpected side-effect of leading them to discover faults that would otherwise have gone unnoticed at that stage.

[34]For example, the onerous unit test criterion required by DO-178B called Modified Condition/Decision Coverage (MC/DC) is intended to exercise all possible code sequences that a compiler might generate from the Boolean expressions appearing in tests and loops [CM93].

3.4.2 Use of Formal Methods to Improve Quality Control and Assurance

Many of those who advocate use of formal methods for safety-critical systems do so because they believe that existing practices are inadequate and that (only) formal methods can significantly increase the assurance provided for these systems. Standards, such as the British Interim Defence Standard 00-55 [MOD91a] and the United States Trusted Systems Evaluation Criteria [DoD85], that mandate use of formal methods for certain classes of systems seem motivated by these beliefs. In this subsection I summarize the evidence in support of these beliefs, examine whether increased quality control and assurance is needed for airborne systems (and if so, for what aspects of those systems), and consider whether and how formal methods can contribute to increased quality control and assurance.

The distinction between quality *control* and quality *assurance* is that the first is concerned with methods for eliminating faults, while the second is concerned with methods for demonstrating that no faults remain (although the distinction is often somewhat blurred in practice). In considering how formal methods may contribute to improved quality control, I will enquire whether there are classes of faults that might escape detection using traditional processes (and might therefore be present in operational systems), and I will examine whether formal methods could help eliminate those faults. In considering their contribution to assurance, I will enquire whether there are classes of faults that are often detected much later than the lifecycle stage where they are introduced, and will examine whether formal methods could help detect those faults earlier, and give assurance that they have been detected.

Formal Methods to Increase Quality Control

As reported in Section 3.3, there is no documented evidence of inadequate quality control for airborne software in current operation—but since the generation of commercial airplanes with truly extensive flight-critical software has accumulated relatively few flight-hours to date (the Airbus A320 has about a million hours, the Airbus A330 and A340 are about to enter service, and the Boeing 777 has not yet flown), the software would have to be very bad indeed for any significant faults to have shown up. The evidence recounted in Section 3.3 from experimental aircraft and from spacecraft, where faults have been documented in flight test or in operation, suggests that *if* current quality control methods are inadequate, then they are most likely to be so in those areas of intrinsically high technical complexity concerning the management of concurrent, distributed activities in the presence of faults or other rare combinations of events. If this assumption is correct, then it is possible that definitive evidence for individual software faults occurring in operation will never be forthcoming, even if software quality *is* inadequate—for the circumstances in which residual faults will be activated will be those of multiple failure, where the exact sequence of events will be very difficult to determine and next to impossible to reproduce, and evidence that could implicate the software (e.g., a trace of its execution) is simply unavailable (the necessary instrumentation is seldom retained after flight test). It is possible, therefore, that the only evidence for the adequacy or otherwise of current quality control practices for flight-critical software will be statistical—and it may require a decade or more to accumulate the quantity of operational experience (i.e., in excess of 10^7 flying hours) that may be expected to reveal very small, but unacceptable, failure rates.

In summary, software quality control and assurance practices have apparently worked adequately in the past; if they are inadequate for the new generation of software-intensive airplanes, their inadequacies will be on the margins and slow to come to light, and the evidence will be equivocal. Therefore, in my opinion, judgments concerning the adequacy of quality control methods for flight-critical software must primarily be based on intellectual and introspective grounds. It is up to systems developers and certification authorities to make these judgments, but I think that if there are any areas where current processes for quality control might be judged inadequate, then they will concern complex problems such as the coordination, partitioning, timing, and synchronization of distributed and concurrent activities in the presence of faults and other combinations of rare events. These are areas of high and inherent technical complexity that must deal with vast numbers of possible behaviors and are difficult, if not impossible, to validate through testing.

But can formal methods validate these vastly complex behaviors, and if so, what type of formal methods? Once again, I believe the analysis must chiefly be an intellectual and introspective one. And on that basis, the reason why formal methods might be expected to help is that they provide a means for describing, specifying, calculating, and thinking about complex behaviors: they can render intellectually manageable a dimension of possibilities that otherwise exceeds our capacity. DO-178B appears to concur with this assessment, and in its Section 12.3.1 states

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.*

I suggest 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, fault tolerance, 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 have been calling 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 an improvement on current practice.

The mechanisms of fault tolerance, distributed coordination, and concurrency management employed in aircraft systems owe little to those studied by academic researchers.[35] This is not a criticism (there is little reason to suppose that academic researchers know more about flight-control systems than those who actually develop them), but the salient point is that, because they do not derive from academic traditions, the architectures and mechanisms used in aircraft systems cannot draw on the analyses and proofs that have been published and subjected to peer review in computer science journals and conferences. Hence, if it is not done already (and if it is, it is not described in the open literature), a very desirable first step towards increased quality control and assurance would be use of formal methods, at a modest level of rigor, to demonstrate the correctness of the basic mechanisms and algorithms concerned with complex behaviors. The intent of this exercise would be to contribute to satisfaction of the software verification process objectives stated in Section 6.1 of DO-178B:

> *The general objectives are to verify that*
>
> ...
>
> **b.** *The high-level requirements have been developed into a software architecture and low-level requirements that satisfy the high-level requirements ...*
>
> ...

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

---

[35] For example:

> "Not far from CNRS-LAAS [a French research establishment with considerable expertise in fault tolerance], Airbus Industrie builds the Airbus A320s. These are the first commercial aircraft controlled solely by a fault-tolerant, diverse computing system. Strangely enough this development owes little to academia" [Kir89].

The GEC architecture [DH90] for the primary flight computers of the Boeing 777 is similarly without academic forbears. An exception to this general rule is Allied Signal's MAFT architecture [KWFT88], which was proposed for some of the design studies in the 7J7, 767X series.

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

I suggest that the specific verification objective for formal methods would be to demonstrate that certain fundamental algorithms and architectural mechanisms involving complex behavior (i.e., behavior that consists almost exclusively of "*discontinuities*") satisfy their corresponding high-level requirements.

Since such requirements and behaviors may not be testable under all combinations of circumstances, this use of formal methods addresses the guidance for software verification activities given in Section 6.2 of DO-178B:

...

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 defined in the Software Verification Plan or Software Verification Results.*

...

In particular, I suggest that formal methods and proofs can provide the "*other means*" required to satisfy the verification process objectives in the case of complex behaviors.

I consider this objective an important one for formal methods since it addresses the source of the observed faults described earlier in this chapter, and I do not see how it can be accomplished by other means—though it seems prudent that formal methods should supplement, not replace, existing practice.

The specific benefit provided by formal methods is that they allow "*complex behaviors*" to be analyzed (by means of proofs), rather than merely reviewed—and analyzed in their totality, rather than merely sampled as by testing or simulation. Thus, the benefit derives from proof, 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 in order to ensure that they accomplish the desired goal.

Level 1 formal methods can provide assurance that critical algorithms and architectural mechanisms achieve their requirements, relative to the level and abstractness of description used, and subject to the fidelity of the modeling employed and accuracy of the proofs performed. Beyond the modest step just recommended, 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. In my opinion, the latter is most likely to be the case, since the assumptions built in to many notations (e.g., synchronous communication in LOTOS) require the very mechanisms (e.g., synchronization) whose correctness we will be trying to establish.

But if Level 2 formal methods add very little in this domain, Level 3 may add a great deal. We will be dealing with difficult problems, where a large number of potential behaviors must be considered—that is why we have 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 [RvH91], and one algorithm for distributed consensus that was published with a detailed proof of its correctness [TP88] nonetheless contains a bug [LR93b]. The flaws in the two examples just cited were discovered while undertaking formal analysis at Level 3 and suggest the benefits that may be derived from this level of rigor. As with Level 2, those Level 3 specification languages that provide built-in models of computation and of concurrency may hinder rather than help the kinds of analysis considered here; a specification language built on an expressive but neutral logic, in which can be constructed models suited to the problems at hand, may prove the most effective tool.

The value of undertaking mechanically-checked proofs is that the dialog with a proof checker leads us to examine all cases and combinations of 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 been proved correct" is not an obstacle to deriving great benefit and additional assurance by applying Level 3 formal methods in this domain (recall Section 3.2 where part of the *Qualification Criteria for Software Verification Tools* was reproduced from DO-178B). 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 semi-formal (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.[36]

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: we also need to validate the modeling employed. That is to say, we need to be sure 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. We also need to be sure that the algorithm and architectural mechanisms considered in the proof will be 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. In my opinion, since the assurance objective of this analysis is to ensure there are no conceptual flaws in the basic algorithms and mechanisms, 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 is accomplished by peer review, supported by analyses (recall the list given in the previous subsection) that demonstrate, for example, that axiomatic specifications are sound (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.[37] Concern that implementations are faithful to their verified specifications is a separate problem, and can be handled using either formal methods, or

[36]For this reason, I do not endorse the requirement in UK Interim Defence Standard 00-55 [MOD91a, paragraph 32.2.3] that a second mechanically-checked proof using a "diverse tool" should be required. The resources required would be better expended on diverse *analysis*, and on human scrutiny of the argument and modeling employed.

[37]UK Interim Defence Standard 00-55 places great stress on validation through "animation" [MOD91a, paragraph 29.3.1 and guidance paragraph 29.6]. By this is meant: (1) use of "formal arguments" to demonstrate consistency with higher-level requirements and to undertake what I call "challenges," and (2) construction of an "executable prototype" to "examine and investigate particular aspects of the specification that have been identified in advance, with particular emphasis on the safety features." While executable prototypes can be useful in some circumstances, I do not consider it likely that they can contribute to increased quality control and assurance for the difficult algorithms and architectural mechanisms under consideration here.

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.

A very reasonable concern about the type of formal methods activity I advocate for increased assurance is that it requires specialized skills of a very high order: I am suggesting nothing less than applying the most rigorous kinds of formal methods to the hardest and most difficult aspects of design. But if we accept that quality control and assurance of airborne software already achieves very high standards, we can hardly expect to push the standards still further without significant effort, training, and skill. However, although technically difficult and intellectually demanding, the applications of formal methods that I advocate are few in number (just the problems where concern is greatest) and small in scale (since they will be undertaken in the early lifecycle and can employ abstract models), and can therefore be accomplished using relatively few (but highly skilled) people. Thus, the challenge for technology-transfer of formal methods into high-assurance contexts is not to bring relatively large numbers of people up to relatively modest levels of accomplishment, but to train relatively few people to extremely high levels—and not just in formal methods, for to formalize the topics of interest and to work effectively with the senior systems, safety, and software engineers concerned will require expertise in many aspects of safety-critical systems design.

Another reasonable concern is that formal methods tools, and theorem provers in particular, will not be adequate to the challenge of the hardest problems. Those whose experience has been restricted to "proof assistants," or other systems with little automation, often have a very false impression of the capabilities of modern theorem proving systems: they extrapolate from their own experience and assume that theorems a little harder than those they have been able to prove must be at the limit of feasibility. In truth, the more automated of modern theorem-proving systems can, under the guidance of skilled and experienced users, prove rather hard theorems rather expeditiously. Certainly, however, problems can be posed that will stress the capability of any current theorem prover—or, rather, the patience of its user—but abstraction may allow us to "downsize" such problems to a manageable scale (or to divide them into several smaller problems). It takes great skill to do this without abstracting away matters of real significance, but the simplification achieved is likely to be of independent value (providing a better grasp on the real issues). Once a problem is reduced to a tractable form, mechanical theorem proving is not a major impediment: most of the time spent in dialog with the theorem prover is spent revising the specification in response to faults discovered, not in developing the final proof [Sha88].

Formal Methods to Increase Quality Assurance

Under the previous heading, I considered the potential contribution of formal methods to aspects of design where traditional methods of quality control might be inadequate and where faults could persist in operational software. In this section, I examine the case where quality control seems adequate, but quality assurance may be considered less than satisfactory. I am interested here in those lifecycle processes that have a high "leakage rate" (i.e., faults are discovered at a later stage of the lifecycle than that which introduced them), or where the assurance processes are poorly controlled, so that it is difficult to provide evidence for their coverage and effectiveness. As seen in Section 3.3, both these characteristics are most strongly expressed in the early lifecycle, and in requirements specification and analysis in particular.

The general problem in the validation of requirements specifications is that it is difficult to measure "coverage" in any objective sense. Requirements analysts collect and study as much pertinent information as possible and examine the written requirements from several perspectives in order to determine whether they are complete, consistent, and free from major faults. In addition, they identify and mentally execute "scenarios" that are rather like tests in that each one explores the ramifications of the requirements specification under a specific hypothesis. But apart from the scenarios and reports of any deficiencies discovered, there are few outputs from the requirements analysis process that are checkable or measurable—basically all that is available is a statement "I checked this to the best of my ability and apart from the items listed, I cannot find any other problems." The effectiveness of the process is thus heavily dependent on the experience, intelligence, and diligence of the personnel involved. When faults are discovered later in the lifecycle and those responsible for requirements analysis are asked how those faults escaped their detection, they generally answer "I didn't know it was meant to (not) do that," or "I didn't think of that scenario." While I do not suggest that formal methods can eliminate these deficiencies, I do believe they can alleviate them, and can contribute to a more analytical and manageable process. The objective would be to contribute to satisfaction of DO-178B Section 6.3.1. (Reviews and Analyses of High-Level Requirements):

a. (Compliance with system requirements): *The objective is to ensure that the system functions to be performed by the software are defined, that the functional, performance, and safety-related requirements of the system are satisfied by the software high-level requirements, and that derived requirements and the reason for their existence are correctly defined.* (Section 6.3.2., dealing with low-level requirements, adds the objective of ensuring that ... *derived requirements and the design basis for their existence are correctly defined.*)

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.).

c. (Compatibility with the target computer): *The objective is to ensure that no conflicts exist between the high-level requirements and the hardware/software features of the target computer, especially, system response times and input/output hardware.*

d. (Verifiability): *The objective is to ensure that each high-level requirement can be verified.*

...

f. (Traceability): *The objective is to ensure that the functional, performance, and safety-related requirements of the system that are allocated to software were developed into the software high-level requirements.*

...

The difficulties of requirements capture and documentation are larger than those addressed by formal techniques alone; what is really needed is a systematic *method* for approaching these problems (and for applying formal techniques to them). For control systems, the approach developed by Parnas and colleagues—variously known as the "A7" or "Software Cost Reduction" (SCR) method—seems most suitable. (A recent description of the A7 method is given by van Schouwen [vS90, vSPM93]; an example, in which a duration calculus [Cha93] is used to support the method, is given by Engel *et al.* [EKM$^+$93].) A more comprehensive approach, developed specifically for aerospace applications and combining object-oriented techniques with those of A7, is the "Consortium Requirements Engineering" (CoRE) method of the Software Productivity Consortium. (This method is currently being applied to Lockheed's C-130J (Hercules) avionics upgrade.)

I suggest that formal specifications and formal analyses should supplement traditional methods for requirements specification and analysis. Within the framework provided by a requirements engineering method, such as A7 or CoRE, formal methods could render quality control and assurance of requirements a more systematic, controlled, and repeatable process.

In particular, I believe that formal methods can help requirements authors state *what* the component concerned is intended to do, what it should *not* do, what *constraints* it must satisfy, and what *assumptions* may be made about its environment. Present techniques may lose these vital items of information; demands for increased precision only drive the specifier towards more operational descriptions of *how* the system is to be constructed. Nor do methodologies such as OMT [RBP+91] address these problems of stating *what* is required; while excellent for many purposes, they serve mainly to provide perspectives on how the system is to be put together.

Similarly, present techniques do not encourage requirements authors or analysts to state explicitly the general constraints and assumptions that apply to a component. Instead, scenarios are used to enumerate the intended circumstances of its operation. Formally stated assumptions and constraints, on the other hand, could capture general properties, and attempts to prove the conjecture

assumptions plus *requirements specification* imply *constraints*

should force systematic enumeration of all the significant scenarios. Work by de Lemos, Saeed, and Anderson [dLSA92] illustrates some steps in this direction.

In addition to systematic exploration of requirements specification in relation to its assumptions and constraints, a suitably detailed specification can be scrutinized with respect to a number of consistency and completeness criteria. A formal specification can be checked for a specific kind of consistency by typechecking, and by showing that its axioms are satisfiable (again, recall the list given in the previous subsection). Similarly, a certain form of completeness can be systematically checked using the methods of Jaffe, Leveson and Melhart [JLHM91].

It would not be necessary to use explicitly formal methods to follow some of the steps suggested above. Natural language, possibly supplemented by diagrams, tables, and some mathematical notation could be adequate for many purposes. And some of the proposed analyses could be reduced to checklists without losing all effectiveness (recall Lutz' report [Lut93b] that a checklist derived from Jaffe, Leveson and Melhart's proposals would have identified more than 75% of the safety-critical faults discovered in the Voyager and Galileo spacecraft). However, specifically formal methods would provide the additional benefit that some review steps could be replaced or supplemented by analyses: that is by repeatable processes amenable to independent, and possibly mechanical, checking. Such systematic analyses seem most likely to contribute to an improvement in assurance.

Item [c.] "*Compatibility with the target computer*" in DO-178B Section 6.3.1. (reproduced above) seems to raise the general question of interfaces (under "*input/output hardware*"). Recalling Lutz' data from Section 3.3 that 25% of all safety critical faults in Voyager and Galileo were interface faults, and that half of those concerned misunderstood hardware interfaces, it seems that more precise specification of such interfaces would be advantageous. Since faults seem to be associated with semantic attributes of the interfaces (e.g., initial states, fault modes, timing and sequencing constraints), a formal notation that permits the expression of such attributes is required; specification of syntactic attributes (e.g., using pseudocode) will be insufficient. æ

3.5 Selection of Formal Methods, Levels and Tools

Before applying formal methods to a system development, it is important to consider carefully a number of choices and decisions: for what purpose are formal methods to be applied, to what components and properties, and with what level of rigor? Once these basic decisions are taken, more specific ones can be addressed, such as the selection of individual methods and of tools.

In the previous section, I identified a number of ways in which formal methods could be used in support of certification. The common thread among those different ways is the ability of formal methods to reduce certain questions to calculation, thereby allowing analyses to replace or supplement reviews. Even when the intellectual substance of a formal specification is chiefly examined through reviews, it is surely the fact that *some* questions, such as syntactic or type consistency, can be settled by calculational processes that distinguishes it from a nonformal specification; if we do not make use of the fact that formal specifications permit formal calculations, then we have not used formal methods in any significant way and there is no need to consider their impact on certification. (This is not to say that the notational, educational, or mental aspects of formal methods might not benefit the software engineering process—merely that they produce little novel evidence that can be examined in support of certification.)

Therefore, in my opinion, the selection of a level of rigor, formal method, and tools, should be largely determined by the type of analysis we wish to perform. Once the kind of analysis is chosen, I believe it will generally be most useful to decide next the level of rigor. If the goal is to analyze the correctness of crucial algorithms and architectural mechanisms, then Level 1 rigor and carefully constructed traditional proofs may be adequate (and certainly superior to no analysis at all); if the proofs are intricate or numerous, then Level 3 rigor and mechanically-checked proofs may be preferable. When specifications are mainly to be used as documentation and examined by review, then a formalized specification notation and Level 2 rigor may be appropriate—provided the notation is supported by tools that perform effective detection of syntactic and semantic faults.

Once we have identified the type of analysis and level of rigor, we can select the most appropriate methods and tools. For Level 2 documentation of sequential components, a notation such as VDM or Z supported by a typechecker may be a good choice. Alternatively, we may prefer the richer mechanization and stronger guarantees of consistency afforded by the tools of a system that is normally used at Level 3. The problem domain will naturally influence the choice of methods and tools: for example, with real-time applications we have to decide whether to use a system that supports those concerns directly, or whether to model them within some more general framework.

For Level 3 applications (i.e., those involving mechanized proof checking), it will be necessary to match the capabilities of the tools used to the requirements of the problem domain: it will seldom be productive to use a process-algebra such as LOTOS for the description of sequential programs and low-level data structures (recall [GH93a]), nor is a program verification system likely to be the best choice for abstract design specifications (see [SB89]). The intended analyses must also be considered, and the methods and tools chosen appropriately. For example, if the goal is to examine whether mode-switching or other complex control logic admits undesirable properties, then a state-exploration system may be the most suitable. But for verification of the correctness of fault-tolerant algorithms, a more general-purpose theorem prover will be a better choice.

These points may seem obvious, but I know of projects where methods and tools were selected apparently by chance, resulting in some strange pairings. For example, one organization purchased a tool intended for the formal development of hardware designs—and it is a perfectly fine tool for that purpose. When a later project was started to formally analyze voting and redundancy management mechanisms, they used the same tool. Now although it is possible to drive screws with a hammer, the practice is unlikely to lead to sturdy joints, nor to refined appreciation of the capabilities of either screws or hammers. Accordingly, in the next few pages, I list some questions that should be considered when selecting tools to support formal methods.

The questions are largely technical ones bearing on the effectiveness of the automated analyses that can be supported by the tools under consideration—although I recognize that political and other nontechnical considerations may also be influential in the selection of methods and tools. Principal among these nontechnical factors are popularity, availability of training courses and textbooks, and standardization and endorsement by various national and international bodies. These factors will be important in many applications of formal methods, but I believe they should generally be outweighed by technical considerations when contemplating application to airborne systems. For flight software, we are less concerned with general improvements in software engineering and more concerned with assurance of software quality—in an industry that already achieves extremely high standards. The main contribution that formal methods can make in assurance for flight software is in providing mechanically-checked analyses for strong properties of requirements, specifications, algorithms, and programs. These mechanized analyses require state-of-the-art capabilities in formal methods and their support tools, and the art is advancing very rapidly. Popular and standardized methods have generally achieved that status over many years and therefore generally lag the state-of-the-art, and their very popularity and standardization make the processes of change and improvement rather ponderous.

3.5.1 Issues in Tool Selection

Most of the topics mentioned here were considered at some length in the previous chapter and are gathered together here mainly for convenience. I assume the tools under consideration provide a formal specification language, parser, typechecker, various utilities, and some support for mechanized proof checking or theorem proving. I generally refer to such a tool as a "verification system" (or simply "system"), and consider its capabilities in the order listed in the previous sentence.

Verification System

- *Does the system have adequate documentation and examples?*

  Clearly, these are required if productive use is to be made of the system. However, it should not be an absolute disqualification if the documentation is not precisely of the form desired: most aerospace projects develop substantial standards and procedures documents, and it is likely that some specially tailored description of the chosen formal method and system will be required, no matter how good its own documentation.

  Examples are important for evaluation and training. If examples directly relevant to the planned application are not available, then a pilot project to create them should be considered.

- *Has the system been used on real problems?*

  Embarking on a major project using an untried verification system is risky. Generally, users should expect that the chosen system has already shown its mettle on *some* significant project, and preferably has been used for purposes similar to those planned.

- *Is it easy to learn? And does it provide effective support for experienced users?*

  An attractive user-interface, perhaps with pop-up menus and other conveniences, can ease the process of learning a new system. It is more important, however, that the system should be a productive tool in the hands of experienced users. A key requirement for a system to be truly easy to use is that it should have a rational structure, so that users can form a mental model of its operation that corresponds to its actual behavior [Nor88].

  Users should be wary of opinions formed of verification systems on the basis of demonstrations or elementary examples. Greater familiarity and more substantial exercises are required in order to estimate how a system will serve on large projects.

- *Does the system support the selected formal methods and analyses effectively?*

  A verification system should be chosen to support the analyses desired, not vice-versa. If the desired analysis is very strong typechecking, then a system that uses an untyped logic will unsuitable, no matter how great its other capabilities. Conversely, if the goal is to analyze correctness of difficult algorithms, then a system with a feeble "proof assistant" will provide more frustration than assistance.

  In my opinion, ability to perform the desired analyses should take precedence over support for a particular formal method. For example, if proofs of desired properties are beyond the capabilities of the available proof assistants for VDM, say, then it will be better to adopt the notation supported by a system whose theorem prover is adequate to the task than to abandon the analysis. It should also be borne in mind that resources are generally finite and bounded: heroic efforts to perform an analysis with an unsuitable tool will consume resources that might have been better used elsewhere in the overall scheme of enhancing software quality control and assurance.

- *Is the system generic or specific to a particular logic and language?*

  While most systems are crafted around a particular specification language and its associated theorem-proving requirements, some provide a more generic foundation that can be adapted to different notations and logics. Isabelle [Pau88], which has been instantiated for several logics including type theory and ZF set theory, is an example of a system that provides a fairly general-purpose foundation. Instantiating such a foundation for a particular logic requires rather specialized skill, and is unlikely to be attempted by projects or users that are primarily interested in using, rather than building, support tools for formal methods. Most users, therefore, will find it more appropriate to evaluate the suitability to their needs of specific instantiations (e.g., type theory in Isabelle), rather than the general foundation itself. However, an advantage of systems instantiated on a general foundation is that it may be possible for users to extend or modify them to match special requirements; a potential disadvantage is that their capabilities may be less than those of one hand-crafted for the logic or specification language concerned.

  An alternative that can provide some of the advantages of a generic system is one based on a very rich foundation, such as higher-order logic (type theory). (More ambitious foundations, specifically intended to support a wide variety of logics, include the Edinburgh Logical Framework (LF) [AHMP92].) It is possible to encode the semantics of other logics (e.g., temporal logics, or Hoare logics) or specification notations (e.g., Z) in higher-order logic. A disadvantage of this approach is that some of the details of the encoding used may obtrude into specifications or proofs and, as with generic systems, the capabilities achieved may be less than those of a system hand-crafted for the logic or specification language concerned.

- *Does the system support the implementation language concerned?*

  If the desired analyses include verification of implementation-level descriptions, such as those written in Ada, then the verification system must support the programming language concerned. However, it may be worth reexamining the purpose of formal analysis of such low-level descriptions. The unit tests required by DO-178B are extremely onerous and cannot easily be eliminated, since they are intended to examine the code generated by the compiler and its operation in the context of the particular operating system, run-time support system, and hardware concerned. These tests accomplish many of the purposes that would be achieved by code-level verification. If, on reflection, it seems that the goals of the formal analysis are more concerned with correctness, or other properties, of the *algorithms* employed, then more abstract modeling can be used, and the link to a specific programming language will not be needed.

Specification Language

- *Does the language have an explicit semantics?*

  A formal specification language must itself have a secure grounding in logic. A complete formal semantics is probably not necessary, though it is desirable, if the language is built on a fairly standard foundation (e.g., classical first- or higher-order logic) without many embellishments. However, when novel or nonstandard constructions are employed, the user has a right to expect that an adequately detailed and formal semantic account has been subjected to peer scrutiny and is available for examination.

- *Is the logic underlying the specification language at least as expressive as first-order predicate calculus, or is it a more specialized logic?*

  First-order predicate calculus with equality is generally considered the minimum foundation for a conveniently expressive specification language; more restricted foundations may be adequate for specialized applications and may offer compensating benefits, such as executability, or very powerfully automated theorem proving. For maximum versatility and convenience of expression, more powerful foundations than pure predicate calculus can be desirable; examples include set theory and higher-order logic. The best choice will depend on the intended application.

- *Does the language have computer-science constructions (such as records, tuples, enumerations, updates)?*

Specification languages, as opposed to raw logics, generally provide built-in notations for these constructions, which are a great convenience in computer-science applications. It is often possible to simulate these constructions if they are absent, but considerable notational convenience will be lost, and theorem proving support will also be less automated. Updates (also known as overriding) are the way in which new values of structured types (such as functions, and records) are constructed from existing values in a purely functional manner (they correspond to assignment to array elements or record fields in imperative programming languages) and it is usually important for the theorem prover to deal with them in a very efficient manner.

- *Does the language have familiar and convenient syntax (e.g., infix + etc.)?*

  Although lisp-like prefix notation has some adherents, most users prefer more familiar notation (i.e., $x * y + z$ rather than `(plus (times x y) z)`). The notation used within the system assumes less importance if there are facilities for typesetting it in a more attractive form for documentation and review. Some systems use true mathematical notation (i.e., $\forall$, rather than a keyword such as `FORALL`); this is not an unmixed blessing, since it can require a nonstandard editor to create such symbols.

- *Is the specification language strongly typed? How rich is the type-system, and how stringent is the error-checking performed by the typechecker?*

  Strong typing is generally considered an advantage in specification languages, just as it is in programming languages. In specification languages, however, the type-system can be richer than is generally feasible for programming languages. Among the more powerful type-constructions are those for predicate subtypes, dependent types, and disjoint unions.

- *How does the logic deal with partial functions?*

  Partial functions are those whose value is undefined for some values of their arguments. Developing a semantics for a specification language that admits partial functions is a challenging problem. The main choices are multiple-valued logics (e.g., LPF [BCJ84]), or logics of partial terms (e.g., Beeson's LPT [Bee86]). The alternative is to treat all functions as total; this can be rather unnatural (e.g., allowing division by zero) in languages with only elementary type-systems, but becomes very effective when predicate subtypes and dependent types are available. Some examples of these approaches were presented in Subsection 2.3.1.2, and they are explained in more technical detail in Appendix subsection A.11.2. In most cases, rather sophisticated theorem proving is required for a sound treatment (LPF and LPT usually generate definedness goals as proof-time, predicate and dependent types generally require theorem proving during typechecking).

- *Does the specification language allow formulas to be introduced as axioms?*

  Specification languages that do not allow the introduction of axioms can impose a rather constructive style: requiring the user to define a function (say) having a required property, rather than being able to simply assert the property axiomatically. The extent to which this is a drawback depends on the application. Axioms are particularly useful when we wish to state assumptions about the environment, and when we wish only to constrain (rather than fix) certain properties.

  When axioms are allowed, the system should support methods for demonstrating their consistency.

- *Does the specification language have a definitional principle that guarantees conservative extension? Is it mechanically checked?*

  It is not always desirable or necessary to introduce concepts axiomatically. Properly constructed definitions have the advantage that they cannot introduce inconsistencies; they are also conducive to efficient theorem proving and can allow specifications to be directly executed. It is desirable for well-definedness to be enforced by the system, so that malformed definitions are not admitted. Checking the well-definedness of some constructions (e.g., recursive definitions) can require theorem proving.

- *Does the system support definition of recursively defined abstract datatypes?*

  Mechanisms for introducing recursively-defined abstract datatypes (such as lists, trees, etc.) are extremely useful; similar mechanisms can be provided for inductively-defined types and functions (such as transitive closure). Disjoint unions (which are very useful for adjoining an "error" value to some other type) can often be manufactured using an abstract datatype facility if they are not provided directly.

- *Are specifications required to be model-oriented, or property-oriented, or can both styles be accommodated?*

  This question is closely related to those concerning whether axioms or definitions are admitted. The model-oriented style is usually associated with definitional mechanism; the property-oriented with axioms. Ideally both styles should be supported.

- *Does the language have overloading and type inference?*

  Strongly-typed languages can become notationally complex if the system does not provide some automatic way to associate appropriate types with expressions. For example, addition on integers is often a different function from that on the reals in many specification languages, but it is generally convenient if both are denoted by the single symbol + (so that the symbol is "overloaded"), and the system uses context to determine whether real or integer addition is intended. Many degrees of sophistication are possible in type-inference mechanisms; it becomes especially challenging (and necessary) in systems that support higher-order logic, subtypes, dependent types, and parameterized modules.

- *Are specifications purely functional, or expressed in terms of pre- and post-conditions on a state, or can both styles be accommodated?*

  Purely functional specifications are closer to ordinary logic and it is generally easier to provide effective theorem proving for this case than for specifications involving state. When an implicit state is present, it is generally necessary to enrich the logic with Hoare sentences or some equivalent construction, or to reduce the problem to ordinary logic by generating verification conditions.

  When state is present, it is often necessary to distinguish variables and constants that are part of the state from those that are purely logical (e.g., bound variables). Some specification languages do not do this very well.

  Specifications involving state are often appropriate for late-lifecycle program verification, but can lead to overly complex and prescriptive specifications at earlier stages, when functional specifications may be preferable.

- *Does the specification language have an encapsulation or modularization mechanism? Can modules be parameterized? Can semantic constraints be placed on the instantiations of parameters? How are the constraints enforced?*

  Just as reuse is supported in programming by procedures, and packages, so it is also useful to encapsulate specifications in modules. And as with programming languages, parameterization greatly increases the utility and genericity of such specification modules: it is desirable to specify sorting, for instance, generically in the type of entity to be sorted, and the ordering relation that is to be used. When specification modules can be parameterized, it is usually essential to be able to place semantic constraints on the allowed instantiations; we should not be allowed to instantiate the sorting module with a relation that is not an ordering. Specifying these kinds of constraints can either be done directly, by attaching assumptions to the formal parameters that must be discharged whenever the module is instantiated, or by allowing modules to be parameterized by theories—OBJ [FGJM85] does this, and more elaborate ideas are discussed in the language proposal *Clear* [BG77].

- *Does the language have a built-in model of computation?*

In most applications, formal methods are used to reason about computational processes. Thus, the chosen specification language must be able to represent the kinds of computations concerned. Some specification languages have a particular model of computation built-in, in the form of programming-language constructs (e.g., Gypsy), or a process algebra (e.g., LOTOS). If such built-in mechanisms are lacking, it may still be possible to model them if the underlying logic is sufficiently rich. For example, imperative, concurrent, distributed, and real-time computations have all been specified quite successfully in specification languages based on higher-order logic. Almost any logic can represent sequential programs by means of function applications and compositions (i.e., they can model functional programming quite directly). As noted in Section 2.2.1.3, important properties of distributed systems can often be analyzed without requiring an explicit model of distributed or parallel computation—for example, important properties of distributed, fault-tolerant algorithms can be verified in models that treat them as recursive functions.

When a model of computation is built in, it is important to be sure that it is suitable for the analysis concerned: a notation based on synchronous communication is not a suitable foundation for studying synchronization algorithms.

- *Is the specification language executable, or does it have an executable subset, or are there some capabilities for animating specifications?*

  Execution and animation provide ways to validate specifications; another way to perform validation is by attempting to prove putative "challenge" theorems. The argument against executable specification languages is that the demands of executability may compromise its effectiveness as a specification medium [HJ89].

- *Is there support for state exploration, model checking, and related methods?*

  As explained in Section 2.2.3, some analyses can be performed efficiently using a sophisticated form of brute-force that allows exploration of all the reachable states entailed by certain kinds of specification. At present, theorem-proving and state-exploration systems tend to be distinct breeds, so that if an analysis will best yield to state-exploration, then it will advisable to use a system specialized to that purpose. If theorem proving is required as well, then it may be necessary to use two systems. An active area of research is concerned with finding effective ways to combine these techniques, so potential users will not be faced with quite such a sharp dichotomy in the future.

Utilities

- *Does the Formal Method have a comprehensive library of standard types, functions, and other constructions? How well validated is the library?*

  A frustration to many who use a verification system is the discovery that very little "background knowledge" is built-in or available in libraries. Each new problem area may require significant investment to formalize the basic building blocks of the field. Some systems do have libraries of such basic constructions as n-bit words, searching, sorting, maxima and minima, majority, and so on (those systems that do not have them built-in will also require libraries for basic arithmetic, sets, trees etc.), and it can be a substantial advantage if the appropriate foundation is already available in this form. Libraries can differ in quality, and it is important to know how well validated are those on which reliance is placed: an inconsistency in a library specification may undermine a significant verification.

  When it is necessary to construct or extend a library of basic definitions in order to support a particular project, the process should not generally be exposed to the "customer": those who want formal methods to help them understand some difficult design problem do not want to be asked to review axiomatizations of, say, the *majority* function. The senior hardware designer for one project refused to participate in further reviews of formal specifications after a morning spent laboring over the theory of n-bit words—that level of detail was not his concern.

- *Does the system provide editing tools? Are they free-form or structure based, or both?*

  Some form of editor is required in order to create and modify specifications. Some verification systems require or allow this to be done using a separate editor. The advantage of this arrangement is that it allows users to use their own preferred editor; its disadvantage is that the interface between the preparation and the analysis of specifications may not be completely smooth. Other systems require specifications to be created and modified using a particular editor that is integrated with the verification system. The editor provided may be a standard one such as Emacs or `vi`, or it may be special purpose. Since they are widely used, standard editors may be considered free of gross defects, but users often have strong preferences for one or the other, and this is a factor that should be considered.

  Special-purpose editors increase the time and effort required to learn the system, and may be less capable than standard ones. Special purpose editors are often structure-based (meaning that they "understand" the syntax of the language concerned). These are often impressive in demonstrations, and seem attractive on small examples. Skilled users can become very frustrated with such an interface, however, and it is very desirable that free-form text editing should also be supported.

Some preferences concerning style of editing and interaction with the system may be cultural: in North America, almost all engineers and scientists are fast touch-typists, whereas this is a very unusual accomplishment among technical personnel in Europe. Thus, the keyboard-intensive style of interaction preferred in North America may be very uncomfortable for Europeans; conversely, Americans, who may do all their interaction within Emacs and leave their workstation's pointing device untouched from one day to the next, can be driven to distraction by the need to traverse lengthy menus and to point and click on selections.

- *Does the system support reviews and formal inspections?*

  Aerospace and other application areas that require high assurance make extensive use of formal reviews, walkthroughs, or inspections. Formal specifications will probably be subjected to the same processes and some tool support may be useful. Basic support includes facilities for browsing and prettyprinting specifications, and for generating cross-reference tables. More sophisticated capabilities might include requirements tracing.

  Specifications that are to be subjected to formal inspections must generally adhere to rigid standards for style and presentation. It is useful if a prettyprinter can generate the required layout automatically. Note that it is often required that lines should be numbered—a simple capability that is often lacking from the prettyprinters of specification languages. Large projects often standardize on a single document-preparation system, and compatibility with the chosen documentation standard can be a make or break issue in selecting a formal methods tool for such projects. The typesetting and document-preparation tools that are familiar in academic and research environments (i.e., LaTeX and troff) are largely unknown in industry.

  Systematic renaming of the identifiers may be necessary in order to conform to style standards. This can play havoc with saved proofs, and it is valuable if a "rename identifier" command is provided to perform this elementary but vital function in a fully robust manner.

- *Does the system have facilities for producing nicely typeset specifications, or for presenting specification in the form of tables, or diagrams?*

  Purely logical formulations may be most effective for deduction, and straight ASCII representations may be necessary for mechanical processing, but alternative presentations may be best suited for human review, especially with those unused to formal specification.

At the very least, it should be possible to typeset specifications in an attractive manner, and mathematical or other compact and familiar notation should be available to the user. Graphical or tabular representations may require special-purpose support, but is desirable that the verification system provide the necessary "hooks" to allow users to provide such extensions.

- *Does the system provide facilities for cross-referencing, browsing, and requirements tracing?*

  Browsing capabilities, which allow the user to instantly refer to the definition or uses of a term, can boost productivity quite considerably during the development of large specifications or proofs. Cross-reference listings (preferably typeset) provide comparable information in a static form suitable for documentation. These capabilities provide some support for requirements tracing (which is usually a manual operation). It is possible that requirements tracing can be partially automated in some applications, given suitable standards and conventions. This will require special-purpose support and, again, it is preferable if the verification system provides "hooks" so that users can provide such extensions themselves.

- *Does the system record the state of a development (including proofs) from one session to the next, so that work can pick up where it left off?*

  This capability is obviously desirable. Naturally, it is best if saving the current state of development is done continuously (so that not everything will be lost if a machine crashes) and incrementally (so that work is not interrupted while the entire state is saved in a single shot).

- *Does the system support mechanisms for change control and version management?*

  Large specifications and verifications are likely to require the efforts of many people and some mechanism for coordinating access and changes will be required. Depending on their design, these capabilities can either be built-in and specialized to the verification system, or can employ a generic version management system such as RCS [Tic85].

- *Does the system propagate the consequences of changes to the specification appropriately? At what granularity are changes recognized? Are updates performed incrementally?*

  Version management is concerned with the *control* of changes to a formal development: ensuring that two people do not modify a component simultaneously, for example. A related, but different, issue concerns propagation of the *consequences* of changes. If a definition is changed, for example, then any proof or theorem which refers to that definition becomes suspect and so, transitively, do any further proofs or theorems that refer to those newly suspect entities. It is essential for a verification system to deal soundly with this propagation of changes: whenever a component is pronounced type-correct, or a theorem proved, these statements must be correct with respect to current versions of the specifications and proofs. Not all systems manage these matters properly. For example, HOL is often praised as a system with a very secure foundation, but its early version did not track changes—so that it was possible to prove a theorem, then modify definitions, and the system would still consider the theorem proved.

Tracking the propagation of changes can be performed at many levels of granularity. At the coarsest level, the state of the entire development can be reset when any part of it is changed; at a finer level, changes can be tracked at the module level; and at the finest level of granularity, they can be tracked at the level of individual declarations.

Once the consequences of changes have been propagated, another choice needs to be made: should the affected parts be reprocessed at once, or only when needed. Clearly, a more complex implementation is required to track changes at the finer levels of granularity, and to reprocess affected parts incrementally and as needed. However, the more complex implementation provides enormous improvements in user's productivity.[38]

Theorem Prover

- *Does the system allow lemmas to be used before they are proved?*

  Lemmas are small theorems that are used in the proof of larger ones (rather like subroutines in programming). When a lemma is first proposed, one generally wants to know whether it is useful before attempting to prove it. Thus, it is generally convenient to be able to cite unproven lemmas in the proof of a larger theorem. The lemma can then be proven later, when its utility has been established. Some systems do not allow things to be done in this order, instead requiring that all lemmas are proved before they are used. In systems that do allow use of unproven lemmas, it is necessary to provide some kind of macroscopic "proof tree" analysis which checks that all postponed proofs are eventually performed.

[38] These and other statements about productivity gains are derived from my own experience: over the years we at SRI have incorporated new capabilities in our EHDM and PVS systems and observed the benefits. These observations are qualitative and informal, however: we have not conducted scientific measurements.

- *Does the system allow new definitions to be introduced during proof? Does it allow existing definitions to be modified?*

  Beyond even the use of unproved lemmas is the ability to invent and introduce lemmas or definitions during an ongoing proof; such flexibility is very valuable, but requires tight integration between the theorem prover and the rest of the verification system.

  A yet more daring freedom is the ability to modify the statement of a lemma or definition during an ongoing proof. Much of what happens during a proof attempt is the discovery of inadequacies, oversights, and faults in the specification that is intended to support the theorem. Having to abandon the current proof attempt, correct the problem, and then get back to the previous position in the proof, can be very time consuming. A system that allows the underlying specification to be extended and modified during a proof confers enormous gains in productivity. Needless to say, the mechanisms needed to support this in a sound way are quite complex.

- *Does the theorem prover allow cases to be postponed, and to be tackled in any order?*

  Most interactive theorem provers work backwards from the theorem to be proved, generating smaller and smaller subcases. Often, the user will be most interested in the main line of the proof, and may wish to postpone minor cases and boundary conditions until satisfied that the overall argument is likely to succeed. Provers that do not allow cases to be postponed and reordered are likely to disrupt the user's thought processes.

- *Does the theorem prover provide automated support for proofs by induction?*

  Proofs by induction are very common, and it is often quite easy to generate the necessary lemmas automatically, once the user has proposed the induction scheme to be used. Doing so can be a very useful service to the user. Some theorem provers (notably that of Boyer and Moore) go further and provide very sophisticated heuristics that are able to propose induction schemes in an automated manner. Although impressive to behold, these may not provide large productivity gains in general.[39]

  Extensionality arguments (i.e., proving that two sets or functions are equal by showing that they have the same members, or the same values for every argument) are another example where the system can provide very useful service by filling in the details of a proof step once the user has indicated the sets or functions concerned.

[39] Because it lacks quantification, the Boyer-Moore prover must often use induction to deal with cases that would otherwise be disposed of by quantificational reasoning.

- *Does the theorem prover provide facilities for comprehending the overall structure of a proof?*

  Losing sight of the forest is easy when grappling among the trees of a lengthy proof. The system should provide mechanisms for displaying the overall structure of the ongoing proof.

- *Does the system identify all the axioms, definitions, assumptions and lemmas used in the proof of a formula (and so on recursively, for all the lemmas used in the proof)?*

  This is another aspect of the "proof tree" analysis mentioned above. Even in systems that require all lemmas to be proved before they are used, it is useful, at the end of a proof, to be able to discover its foundations. Such information helps eliminate unnecessary axioms and definitions from theories, and identifies the assumptions that must be validated by external means.

- *Does the theorem prover provide information that will help construct a human-readable journal-style proof?*

  Proofs of significant theorems should be subjected to human review as well as mechanical proof checking. It is beneficial if there is some similarity between the human- and machine-checked proofs: a preliminary informal proof should help guide interaction with the theorem prover, and the mechanically-checked proof should help construct a compelling journal-level argument. Not all theorem-proving systems work in ways that resemble the steps of a journal-style proof, and for some purposes may be considered unacceptable on that account. Note that what are required are steps that resemble those of traditional mathematical demonstrations, not the primitive steps of textbook treatments of formal deduction.

- *Is it easy to reverify a theorem following slight changes to the specification?*

  Assumptions and requirements may change after a verification has been completed; often, too, it is necessary to revisit the earlier parts of a large verification as later work reveals the need for changes and extensions. Ideally, the investment in completed portions of the verification should assist and encourage these improvements by making it relatively easy to explore the consequences of changes. This requires that proofs should be robust in the face of small changes to the specification. Theorem provers are inherently more robust than proof checkers in this regard, since they perform more of the proof search themselves. However, apparently minor design choices can have a significant impact on the robustness of a proof: for example, are quantified variables recorded by name (so that a simple renaming of identifiers in the specification may derail a saved proof) or by position? And if the latter, is the order sensitive to the number or order of presentation of other quantified variables (so that adding a new variable will throw a proof off track), or to deeper measures that may be expected to remain invariant over minor changes to the specification?

When a change to the specification is sufficiently large that saved proofs cannot automatically adjust to their new circumstances, the user must take over to fix things. Once again, small design decisions can have great impact on the ease of proof maintenance.[40] For example, is the prover able to present the user with just the failing proof branches, or must the user redevelop the whole proof? And does the prover have "fast-forward," "single step," and "undo" facilities so that the user can interactively "zoom in" on the failed parts of the proof to splice in corrections?

While some of the facilities required for proof development are also useful in proof maintenance, a verification system will be better suited to large tasks if its developers have explicitly considered the maintenance issue.

- *How is the theorem prover controlled and guided?*

  The distinction between a theorem prover and a proof checker is in the quantity and nature of guidance that must be supplied by the user: theorem provers require less guidance, and less direct guidance than proof checkers. Apart from those based on exhaustive search[41], even the most powerful theorem provers require some direction from the user. The Boyer-Moore prover, for example, is guided through the order in which lemmas are presented, and by the form of those lemmas (for example, $A = B$ has a different effect than $B = A$). The advantage of a very automatic theorem prover is that the user has to provide less information, and proofs tend to be robust in the face of changes. The disadvantage is that guidance is provided in a very indirect manner, which requires intimate understanding of the operation of the prover: users must anticipate how the theorem prover will search for a proof, rather than instruct it to execute their proof.

  Proof checkers can require differing degrees of guidance, and the guidance may be presented interactively, or up front. Those that require guidance to be presented up front, and thereafter operate autonomously, generally cause the user to break the proof down into many small lemmas, for it is difficult to plan a proof in complete detail without seeing some of the intermediate steps.

[40] Life imitates art: Dijkstra's series of EWD memos includes several parodies from the Chairman of Mathematics Inc., such as "our Proof of the Riemann Hypothesis has been brought into the field, contrary to the advice of our marketing manager who felt it still required too much maintenance. . . at the end of March, we transferred fifty mathematicians from Production to Field Support" [Dij82].

[41] Resolution provers, for example, perform exhaustive search. Even these require human control, in the form of selecting strategies, and assigning values to various parameters.

Proof checkers that operate interactively are generally the easiest for users to understand. Interactive checkers that require considerable guidance are often called "proof assistants."

My experience and that of my colleagues is that the most powerful interactive proof checkers are comparable to the most powerful automatic theorem provers in terms of human productivity, and are easier to learn and to use. Noninteractive proof checkers require a great deal of skill to use effectively. The productivity of proof checkers that lack effective low-level automation (such as integrated arithmetic and rewriting) is orders of magnitude less than that of more powerfully automated systems; the productivity of some proof assistants may be so low as to render them unsuited to realistic applications.

- *Does the theorem prover perform rewriting?*

  Rewriting (replacing an instance of one side of an equation by the corresponding instance of the other) is one of the most useful strategies in theorem proving and a theorem prover that lacks this capability is unlikely to prove very productive.[42] For maximum productivity, however, much more than just a straightforward rewriter is required. *Conditional* equations (those of the form $P \supset A = B$) are very common, and require effective strategies[43]; the *matching* that selects applicable instances of equations for rewriting must often be performed modulo some theory[44]; and it is usually very important to silently solve the conditional part when expanding definitions so that the user is not overwhelmed with irrelevant cases[45].

- *Does the theorem prover provide automated support for arithmetic reasoning?*

  Most proofs require some reasoning involving the properties of numbers. Simple facts about the natural numbers and the integers arise in almost every proof (often as a result of induction or counting arguments);[46] more sophisticated properties, and also properties involving rational or real numbers, arise in explicitly numerical algorithms (which are common in flight-control applications). It is possible to provide powerful automation for arithmetic reasoning in the form of "decision procedures" that *guarantee* to solve all arithmetic

[42] Some technical background on rewriting is provided in Appendix section A.5.1.2.

[43] The most conservative strategy will only do the rewrite if P can be discharged immediately; another strategy will do the rewrite and carry P along as an additional hypothesis.

[44] For example, an equation $f(x+1) = g(x) + \ldots$ can be matched against $\ldots f(x+2-1)\ldots$ only in the presence of arithmetic. Cases of this kind arise very frequently indeed.

[45] For example, expanding a definition of the form $f(x) = \textbf{if } x = 0 \textbf{ then } A \textbf{ else } B \textbf{ endif}$ in the context $f(z+1)$, where z is a natural number, should cause the prover to discharge the condition $z \geq 0 \supset z+1 \neq 0$, thereby allowing $f(z+1)$ to be rewritten directly to B.

[46] Some technical background is provided in Appendix section A.5.2.

problems within a certain class (e.g., Presburger arithmetic). As with rewriting, this facility needs to be closely integrated with other reasoning procedures for maximum benefit. Effective automation of arithmetic reasoning provides an enormous boost to user productivity in all application domains, and is essential for any work involving explicitly numerical calculations (e.g., clock synchronization algorithms).

- *Can the theorem prover handle large propositional expressions efficiently? Does it employ BDDs?*

  Some problem areas (e.g., hardware design, or mode-switching and alarm logic in avionics) can generate very large propositional formulas (i.e., boolean expressions), involving dozens or even hundreds of terms. Simple truth-table or tableau-based procedures for propositional tautology checking or simplification are easily overwhelmed by such formulas and may take hours to return a result. There are, however, alternative procedures that generally operate much faster (the speed-ups can be orders of magnitude). Those based on ordered Boolean (or Binary) Decision Diagrams (BDDs) are generally the most efficient.[47] As always, it is not sufficient for the theorem prover simply to call a BDD package: efficient propositional reasoning needs to be tightly integrated with the other basic inference mechanisms.

- *Does the theorem prover present users with their own formulas or with canonical representations? Are quantifiers retained?*

  Interactive theorem provers or proof checkers must display the evolving state of a proof so that the user can study it and propose the next step. It is generally much easier for the user to comprehend the proof display if it is expressed in the same terms as the original specification. Provers that display formulas in some canonical or "simplified" form are likely to complicate the user's task. Obviously, formulas change as proof steps are performed, but it is usually best if each transformation in the displayed proof corresponds to an action explicitly invoked by the user. Thus, for example, universal quantifiers are almost always eliminated at some stage by the process of Skolemization, but it is usually best to perform this only when the user explicitly requests it be done.

- *Does the theorem prover minimize the quantity and maximize the relevance of information presented to the user?*

  Interactive theorem provers must avoid overwhelming the user with information. That is why, as noted earlier, irrelevant cases should be silently discarded when expanding definitions. Ideally, the user should be expected to examine

[47] Some references are given in Appendix section A.3.1.

less than a screenful of information at each interaction. It can require powerful low-level automation to prune (only) irrelevant information effectively. The system can assist comprehension by highlighting what has changed since the last display.

- *Does the theorem prover provide automated support for instantiation of quantified variables?*

  The major step in many proofs consists in supplying the appropriate instantiations for existentially quantified variables.[48] Often, however, plausible instantiations can be found through the mechanism of *unification*. Productivity can be considerably enhanced if the system assists the user with mechanizations of this sort.

- *Can proofs be "cut and pasted" from one formula to another?*

  Often several formulas are sufficiently similar that a proof for one will also work (perhaps following small adjustments) on the others. It is therefore useful if the proof for one can be "cut and pasted" on to the others. The proof must be rerun to see if it works in the new context; if it does not, the facilities for proof maintenance can be used to make the necessary adjustments.

- *Does the theorem prover allow the user to compose proof steps into larger ones?*

  Within a given application, proofs often take on a repetitive pattern: for example, "rewrite with these formulas, then use induction and arithmetic." The "cut and paste" model of proof development described above can deal with very repetitive cases, but more varied problems may best be dealt with by user-written proof "macros" or "subroutines" (often called "tactics" or "strategies") that can be applied as single steps. Provers differ in the sophistication of the "control language" that guides the application of such macros—some providing only elementary selection ("apply these proof steps in order until one succeeds") and iteration ("repeat this proof step until it fails") operators, while others provide full programming languages (e.g., the programming language ML was originally developed as the control language for the LCF theorem prover). Generally speaking, a prover whose primitive steps are powerful and robust can use a less elaborate control language than one whose primitive steps are small and fragile.

[48] The Herbrand-Skolem-Gödel theorem reduces first-order theorem proving to instantiation plus propositional calculus; since propositional calculus is decidable, it can be argued that instantiation is the essentially hard part of first-order theorem proving.

- *Can users extend the capabilities of the theorem prover with their own code? Does the theorem prover have a "safe" mode that performs only the built-in inference procedures?*

  Beyond macros lie abilities to extend the theorem prover itself. The advantage of prover extensions over simple macros is that a prover extension can examine the data structures maintained by the prover in order to perform specialized search procedures. Clearly, any code that manipulates the data structures maintained by the prover has the capability to introduce unsoundness. Soundness can be retained, however, if user-written code is used only for search and is constrained to call the (presumably sound) standard inference functions of the prover in order to change the data structures.[49]

  More ambitious capabilities allow new proof procedures to be added, provided that are proven correct beforehand. The metafunctions of the Boyer-Moore prover [BM81] have this character. "Reflective" theorem provers, a research topic, support reasoning about their own proof procedures and can be extended in quite general ways.

3.6 Conclusion

> *"In science, nothing capable of proof ought to be accepted without proof."* [Ded63, page 31]

> *"The virtue of a logical proof is not that it compels belief but that it suggests doubts."* [Lak76, page 48]

This report has covered a lot of material, and I have tried to present it in a balanced and neutral manner. In this section I will state my personal conclusions and 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

[49] In systems based on the LCF model, there is no distinction between proof macros (called "tactics" and "tacticals" in LCF) and prover extensions, and soundness is enforced in a very clever way using the type system of ML. In PVS, on the other hand, macros are written in a limited "strategy language," while prover extensions are written in a restricted, purely applicative, fragment of lisp and soundness is maintained because prover extensions cannot have side-effects.

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 airborne software—and unnecessary too, since the industry seems to be doing a mostly satisfactory job using nonformal methods.

Nonetheless, I believe the 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 seem 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 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, 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 (and that 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.

In principle, all formal methods support the notion that properties of designs should be *calculated* (i.e., proved) from their formal specification; in practice, formal methods differ greatly in the extent to which they support the reality of that notion. In my opinion, formal methods can provide benefit to industries that already employ stringent processes for quality control and assurance only to the extent that they exploit the opportunity to reduce questions to calculation; the unique attribute of formal methods is that they allow reviews (i.e., activities based on consensus) to be replaced by analysis (i.e., repeatable, checkable, calculational activities). Thus, I consider that only Level 3 formal methods are likely to be fully effective in the role described: that means methods supported by mechanized proof-checking or state-exploration tools. To be maximally effective, these methods need to be applied early in the lifecycle, to the hardest and most critical aspects of design, expressed in relatively abstract form. By focusing on the hardest and most critical problems, formal methods may best contribute to assurance; by using abstract modeling, we render the verification and validation problems tractable; and by working in the early lifecycle, the cost benefits of early debugging may offset the expense of applying formal methods.

These conclusions and recommendations 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), but I believe that the first step is to get them used at all, and for serious purposes. 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. 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 (existing) forms of assurance. In my opinion, arguments for more extensive use of some kinds of formal methods (e.g., formal specification of requirements) can be made, and indeed are better made, on grounds of cost: for although the traditional practices do seem effective, they are undeniably expensive. This represents an opportunity for formal methods, but it is not one that requires regulatory encouragement. However, if more extensive application of formal methods is undertaken, then I believe that the material offered for consideration in support of certification should include substantial evidence of an analytic, calculational character. In particular, it should include evidence for type-correctness (extending beyond individual modules to their interdependencies), for well-formedness of definitions, for consistency of axiomatizations, and for validity of the modeling employed.

On the other hand, my 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 be expected to improve a process that is already very effective? And although these challenges are tough, they are relatively few in number and small in scale, and can be undertaken by a small team of highly skilled people. The tools that are currently available to support these ambitious applications of formal methods are not ideal, but I believe they are adequate.

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.

Bibliography

[ABHM85] T. Anderson, P. A. Barrett, D. N. Halliwell, and M. R. Moulding. An evaluation of software fault tolerance in a practical system. In *Fault Tolerant Computing Symposium 15*, pages 140–145, Ann Arbor, MI, June 1985. IEEE Computer Society.

[ABL89] A. Frank Ackerman, Lynne S. Buchwald, and Frank H. Lewski. Software inspections: An effective verification process. *IEEE Software*, 6(3):31–36, May 1989.

[Abr80a] J. R. Abrial. *The Specification Language Z: The Basic Library.* Programming Research Group, Oxford University, Oxford, UK, 1980.

[Abr80b] J. R. Abrial. *Specification of Some Aspects of a Simple Batch Operating System.* Programming Research Group, Oxford University, Oxford, UK, May 1980.

[AD91] Christine Anderson and Merlin Dorfman, editors. *Aerospace Software Engineering: A Collection of Concepts*, volume 136 of *Progress in Astronautics and Aeronautics.* American Institute of Aeronautics and Astronautics, Washington, DC, 1991.

[Add91] Edward Addy. A case study on isolation of safety-critical software. In COMP [COMP91], pages 75–83.

[AHMP92] Arnon Avron, Furio Honsell, Ian A. Mason, and Robert Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9(3):309–354, December 1992.

[AHW+90] G. H. Archinoff, R. J. Hohendorf, A. Wassyng, B. Quigley, and M. R. Borsch. Verification of the shutdown system software at the Darlington nuclear generating station. In *International Conference on Control and Instrumentation in Nuclear Installations*, Glasgow, UK, May 1990. The Institution of Nuclear Engineers.

[AINP88] Peter B. Andrews, Sunil Issar, Daniel Nesmith, and Frank Pfenning. The TPS theorem proving system. In Lusk and Overbeek [LO88], pages 760–761.

[AJ90] Heather Alexander and Val Jones. *Software Design and Prototyping using me too*. Prentice Hall International, Hemel Hempstead, UK, 1990.

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

[AL90] T. Anderson and P. A. Lee. *Fault-Tolerance: Principles and Practice (Second, revised edition)*. Springer-Verlag, Vienna, Austria, 1990.

[ALN+91] J.-R. Abrial, M. K. O. Lee, D. S. Neilson, P. N. Scharbach, and I. H. Sørensen. The B-method. In S. Prehn and W. J. Toetenel, editors, *VDM '91: Formal Software Development Methods*, pages 398–405, Noordwijkerhout, The Netherlands, October 1991. Volume 552 of *Lecture Notes in Computer Science*, Springer-Verlag. Volume 2: Tutorials.

[AN89] Katsuji Akita and Hideo Nakamura. Safety and fault-tolerance in computer-controlled railway signalling systems. In A. Avižienis and J. C. Laprie, editors, *Dependable Computing for Critical Applications*, pages 107–131, Santa Barbara, CA, August 1989. Volume 4 of *Dependable Computing and Fault-Tolerant Systems*, Springer-Verlag, Vienna, Austria.

[Ano89] Anonymous. Reprogramming capability proves key to extending Voyager 2's journey. *Aviation Week and Space Technology*, page 72, August 7, 1989.

[AP93] Stephen Austin and Graeme I. Parkin. Formal methods: a survey. Technical report, Division of Information Technology and Computing, National Physical Laboratory, Teddington, Middlesex, UK, March 1993.

[Art91] R. D. Arthan. On formal specification of a proof tool. In Prehn and Toetenel [PT91], pages 356–370.

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

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

[Bar78] Jon Barwise. An introduction to first-order logic. In Jon Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, chapter A1, pages 5–46. North-Holland, Amsterdam, Holland, 1978.

[Bar89a] Geoff Barrett. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering*, 15(5):611–621, May 1989.

[Bar89b] Jon Barwise. Mathematical proofs of computer system correctness. *Notices of the American Mathematical Society*, 36:844–851, September 1989.

[BBH+74] H. Bekič, D. Bjørner, W. Henhapl, C. B. Jones, and P. Lucas. A formal definition of a PL/1 subset. Technical Report 25.139, IBM Laboratory, Vienna, Austria, 1974.

[BC81] Eike Best and Flaviu Cristian. Systematic detection of exception occurrences. *Science of Computer Programming*, 1(1):115–144, 1981.

[BC85] Jean-François Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of **while**-programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, January 1985.

[BCAG94] *Statistical Summary of Commercial Jet Aircraft Accidents, Worldwide Operations, 1959–1993.* Published annually by: Airplane Safety Engineering (B-210B), Boeing Commercial Airplane Group, Seattle, WA, March 1994.

[BCD91] Ricky W. Butler, James L. Caldwell, and Ben L. Di Vito. Design strategy for a formally verified reliable computing platform. In COMP [COMP91], pages 125–133.

[BCJ84] H. Barringer, J. H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[Bee86] Michael J. Beeson. Proving programs and programming proofs. In *International Congress on Logic, Methodology and Philosophy of Science VII*, pages 51–82, Amsterdam, 1986. North-Holland. Proceedings of a meeting held at Salzburg, Austria, in July, 1983.

[Bev89] William R. Bevier. Kit and the short stack. *Journal of Automated Reasoning*, 5(4):519–530, December 1989.

[BF86] Robin E. Bloomfield and Peter K. D. Froome. The application of formal methods to the assessment of high integrity software. *IEEE Transactions on Software Engineering*, SE-12(9):988–993, September 1986.

[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.

[BG77] Rod Burstall and Joseph Goguen. Putting theories together to make specifications. In Raj Reddy, editor, *Proceedings, Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058. Department of Computer Science, Carnegie-Mellon University, 1977.

[BH90] Bishop Brock and Warren A. Hunt, Jr. Report on the formal specification and partial verification of the VIPER microprocessor. Technical Report 46, Computational Logic Incorporated, Austin, TX, January 1990.

[BH91] Bishop Brock and Warren A. Hunt, Jr. Report on the formal specification and partial verification of the VIPER microprocessor. In COMP [COMP91], pages 91–98.

[Bis93] P. G. Bishop. The variation of software survival time for different operational input profiles (or why you can wait a long time for a big bug to fail). In *Fault Tolerant Computing Symposium 23* [IEE93], pages 98–107.

[BJ92] Neil A. Brock and David M. Jackson. Formal verification of a fault tolerant computer. In *11th AIAA/IEEE Digital Avionics Systems Conference*, pages 132–137, Seattle, WA, October 1992. The Institute of Electrical and Electronics Engineers.

[Bjø81] D. Bjørner. The VDM principles of software specification and program design. In *TC2 Working Conference on Formalization of Programming Concepts*, pages 44–74. Volume 107 of *Lecture Notes in Computer Science*, IFIP, Springer-Verlag, 1981.

[BL76] D. E. Bell and L. J. La Padula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, Mitre Corporation, Bedford, MA, March 1976.

[BL92] Sarah Brocklehurst and Bev Littlewood. New ways to get accurate reliability measures. *IEEE Software*, 9(4):34–42, July 1992.

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, NY, 1979.

[BM81] R. S. Boyer and J S. Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In R. S. Boyer and J S. Moore, editors, *The Correctness Problem in Computer Science*. Academic Press, London, 1981.

[BM86] R. S. Boyer and J S. Moore. Integrating decision procedures into heuristic theorem provers: A case study with linear arithmetic. In *Machine Intelligence*, volume 11. Oxford University Press, 1986.

[BM88] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, New York, NY, 1988.

[Boe88] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, May 1988.

[Bou68] N. Bourbaki. *Elements of Mathematics: Theory of Sets*. Addison-Wesley, Reading, MA, 1968.

[BR91] J. C. Bicarregui and B. Ritchie. Reasoning about VDM using the VDM support tool in Mural. In Prehn and Toetenel [PT91], pages 371–388.

[Bro87] Frederick P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[BS88] G. Birtwistle and P. A. Subrahmanyam, editors. *VLSI Specification, Verification and Synthesis*. Kluwer Academic Publishers, Boston, MA, 1988.

[BS89] G. Birtwistle and P. A. Subrahmanyam, editors. *Current Trends in Hardware Verification and Theorem Proving*. Springer-Verlag, New York, NY, 1989.

[BSH86] Victor R. Basili, Richard W. Selby, and David H. Hutchens. Experimentation in software engineering. *IEEE Transactions on Software Engineering*, SE-12(7):733–743, July 1986.

[Bun83] Alan Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press, London, UK, 1983.

[Bus90] Marilyn Bush. Improving software quality: The use of formal inspections at the Jet Propulsion Laboratory. In *12th International Conference on Software Engineering*, pages 196–199, Nice, France, March 1990. IEEE Computer Society.

[BW84] Victor R. Basili and David M. Weiss. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, SE-10(11):728–738, November 1984.

[BY92] William R. Bevier and William D. Young. Machine checked proofs of the design of a fault-tolerant circuit. *Formal Aspects of Computing*, 4(6A):755–775, 1992.

[CAB+86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Car89] Bernard Carré. Program analysis and validation. In Sennett [Sen89], chapter 8.

[CDD90] Flaviu Cristian, Bob Dancey, and Jon Dehn. Fault-tolerance in the advanced automation system. In *Fault Tolerant Computing Symposium 20*, pages 6–17, Newcastle upon Tyne, UK, June 1990. IEEE Computer Society.

[CDM86] P. Allen Currit, Michael Dyer, and Harlan D. Mills. Certifying the reliability of software. *IEEE Transactions on Software Engineering*, SE-12(1):3–11, January 1986.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CFL+91] Jeffrey V. Cook, Ivan V. Filippenko, Beth H. Levy, Leo G. Marcus, and Telis K. Menas. Formal computer verification in the state delta verification system (SDVS). In *AIAA Computing in Aerospace VIII*, pages 77–87, Baltimore, MD, October 1991. AIAA paper 91-3715.

[CG88] Robert F. Cmelik and Narain H. Gehani. Dimensional analysis with C++. *IEEE Software*, 5(3):21–27, May 1988.

[CGH+92] E. M. Clarke, O. Grumberg, H. Haraishi, S. Jha, D. Long, K. L. McMillan, and L. Ness. Verification of the Futurebus+ cache coherence protocol. Technical Report CMU-CS-92-206, School of Computer Science, Carnegie Mellon University, 1992.

[CGR93a] Dan Craigen, Susan Gerhart, and Ted Ralston. Formal methods reality check: Industrial usage. In J. C. P. Woodcock and P. G. Larsen, editors, *FME '93: Industrial-Strength Formal Methods*, pages 250–267, Odense, Denmark, April 1993. Volume 670 of *Lecture Notes in Computer Science*, Springer-Verlag.

[CGR93b] Dan Craigen, Susan Gerhart, and Ted Ralston. An international survey of industrial applications of formal methods; Volume 1: Purpose, approach, analysis and conclusions; Volume 2: Case studies. Technical Report NIST GCR 93/626, National Institute of Standards and Technology, Gaithersburg, MD, April 1993.

[Cha92] P. Chapront. Vital coded processor and safety related software design. In Frey [Fre92], pages 141–145.

[Cha93] Zhou Chaochen. Duration calculi: An overview. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Proceedings of the International Conference on Formal Methods in Programming and Their Applications*, pages 256–266, Novosibirsk, Russia, June/July 1993. Volume 735 of *Lecture Notes in Computer Science*, Springer-Verlag.

[CHB79] R. H. Campbell, K. H. Horton, and G. G. Belford. Simulations of a fault-tolerant deadline mechanism. In *Fault Tolerant Computing Symposium 9*, pages 95–101, Madison, WI, June 1979. IEEE Computer Society.

[Che89] Mikhail Chernyshov. Post-mortem on failure. *Nature*, 339:9, May 4, 1989.

[CHJ86] B. Cohen, W. T. Harwood, and M. I. Jackson. *The Specification of Complex Systems*. Addison-Wesley, Wokingham, England, 1986.

[CJ90] J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In Carroll Morgan and J. C. P. Woodcock, editors, *Proceedings of the Third Refinement Workshop*, pages 51–69. Springer-Verlag Workshops in Computing, 1990.

[CKM+91] Dan Craigen, Sentot Kromodimoeljo, Irwin Meisels, Bill Pase, and Mark Saaltink. EVES: An overview. In Prehn and Toetenel [PT91], pages 389–405.

[CLS88] Stephen S. Cha, Nancy G. Leveson, and Timothy J. Shimeall. Safety verification in Murphy using fault tree analysis. In *10th International Conference on Software Engineering*, pages 377–386, Singapore, April 1988. IEEE Computer Society.

[CLSK87] S. Cha, N. G. Leveson, T. J. Shimeall, and J. C. Knight. An empirical study of software error detection using self-checks. In *Fault Tolerant Computing Symposium 17*, pages 156–161, Pittsburgh, PA., July 1987. IEEE Computer Society.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1988.

[CM93] John Joseph Chilenski and Steven P. Miller. Applicability of modified condition/decision coverage to software testing. Issued for information under FAA memorandum ANM-106N:93-20, August 1993.

[COCD86] B. A. Carré, I. M. O'Neill, D. L. Clutterbuck, and C. W. Debney. SPADE: the Southampton program analysis and development environment. In I. Sommerville, editor, *Software Programming Environments.* Peter Pereginus, 1986.

[Coh88] A. J. Cohn. A proof of correctness of the VIPER microprocessors: The first level. In Birtwistle and Subrahmanyam [BS88], pages 27–72.

[Coh89a] A. J. Cohn. Correctness properties of the VIPER block model: The second level. In Birtwistle and Subrahmanyam [BS89], pages 1–91.

[Coh89b] Avra Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(2):127–139, June 1989.

[COMP89] *COMPASS '89 (Proceedings of the Fourth Annual Conference on Computer Assurance)*, Gaithersburg, MD, June 1989. IEEE Washington Section.

[COMP90] *COMPASS '90 (Proceedings of the Fifth Annual Conference on Computer Assurance)*, Gaithersburg, MD, June 1990. IEEE Washington Section.

[COMP91] *COMPASS '91 (Proceedings of the Sixth Annual Conference on Computer Assurance)*, Gaithersburg, MD, June 1991. IEEE Washington Section.

[Coo90] Henry S. F. Cooper Jr. Annals of space (the planetary community)—part 1: Phobos. *New Yorker*, pages 50–84, June 11, 1990.

[Cou93] Costas Courcoubetis, editor. *Computer-Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, Elounda, Greece, June/July 1993. Springer-Verlag.

[CP85] W. J. Cullyer and C. H. Pygott. Hardware proofs using LCF-LSM and ELLA. Memorandum 3832, Royal Signals and Radar Establishment, September 1985.

[Cra92] R. H. Crane. Experience gained in the production of licensable safety-critical software for Darlington NGS. In *EPRI Workshop on Methodologies for Cost-Effective, Reliable Software Verification and Validation*, pages 5–1 to 5–37, Palo Alto, CA, January 1992. Electric Power Research Institute (EPRI). (Meeting held in Chicago, August, 1991; proceedings published as EPRI TR-100294).

[Cri84] Flaviu Cristian. Correct and robust programs. *IEEE Transactions on Software Engineering*, SE-10(3):163–174, March 1984.

[Cri89] Flaviu Cristian. Exception handling. In T. Anderson, editor, *Dependability of Resilient Computers*. Blackwell Scientific Publications, 1989.

[Cul88] W. J. Cullyer. Implementing safety critical systems: The VIPER microprocessor. In Birtwistle and Subrahmanyam [BS88], pages 1–26.

[CV91] Vinod Chandra and M. R. Verma. A fail-safe interlocking system for railways. *IEEE Design & Test of Computers*, 8(1):58–66, March 1991.

[Dav92] Alan M. Davis. Why industry often says 'no thanks' to research. *IEEE Software*, 9(6):97–99, November 1992.

[dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematica*, 34(5):381–392, 1972. (Journal published by North-Holland, supercedes *Koninklijke Nederlandse Akademie van Wetenschappen, Series A*).

[dB91] N. G. de Bruijn. Checking mathematics with computer assistance. *Notices of the American Mathematical Society*, 38(1):8–16, January 1991.

[DB92] Ben L. Di Vito and Ricky W. Butler. Formal techniques for synchronized fault-tolerant systems. In Landwehr et al. [LRS92], pages 163–188.

[DD77a] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[DD77b] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[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.

[Ded63] Richard Dedekind. *Essays on the Theory of Numbers*. Dover Publications, Inc., New York, NY, 1963. Reprint of the 1901 Translation by Wooster Woodruff Beman; the original essay was written 1887.

[DF90] Janet R. Dunham and George B. Finelli. Real-time software failure characterization. In COMP [COMP90], pages 39–45.

[DG90] Norman Delisle and David Garlan. A formal specification of an oscilloscope. *IEEE Software*, 7(5):29–36, September 1990.

[DG93] B. Terry Devlin and R. David Girts. MD-11 automatic flight system. *IEEE Aerospace and Electronic Systems Magazine*, 8(3):53–56, March 1993.

[DGK+90] Ben Di Vito, Cristi Garvey, Davis Kwong, Alex Murray, Jane Solomon, and Amy Wu. The Deductive Theory Manager: A knowledge based system for formal verification. In *Proceedings of the Symposium on Research in Security and Privacy*, pages 306–318, Oakland, CA, May 1990. IEEE Computer Society.

[DH90] R. W. Dennis and A. D. Hills. A fault tolerant fly by wire system for maintenance free applications. In *9th AIAA/IEEE Digital Avionics Systems Conference*, pages 11–20, Virginia Beach, VA, October 1990. The Institute of Electrical and Electronics Engineers.

[DHB91] Flemming Damm, Bo Stig Hansen, and Hans Bruun. On typechecking in VDM and related consistency issues. In Prehn and Toetenel [PT91], pages 45–62.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[Dij82] Edsger W. Dijkstra. EWD539: Mathematics inc., a private letter from its chairman. In *Selected Writings on Computing: A Personal Perspective*, Texts and Monographs in Computer Science, pages 184–187. Springer-Verlag, New York, NY, 1982.

[Dij89] Edsger W. Dijkstra. On the cruelty of really teaching computer science. *Communications of the ACM*, 32(12):1398–1404, December 1989. Comments by seven colleagues follow Dijkstra's paper on pp. 1405–1414.

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

[dLSA92] R. de Lemos, A. Saeed, and T. Anderson. A train set as a case study for the requirements analysis of safety-critical systems. *Computer Journal*, 35(1):30–40, February 1992.

[DoD84] *Military Standard MIL-STD-882B: System Safety Program Requirements.* Department of Defense, Washington, DC, March 1984. Revised 1 July 1987.

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

[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.

[Duk89] Eugene L. Duke. V&V of flight and mission-critical software. *IEEE Software*, 6(3):39–45, May 1989.

[DV89] Michel Diaz and Chris Vissers. SEDOS: Designing open distributed systems. *IEEE Software*, 6(6):24–33, November 1989.

[Dye92] Michael Dyer. *The Cleanroom Approach to Quality Software Development.* John Wiley and Sons, New York, NY, 1992.

[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.

[Egg90] Paul Eggert. Toward special-purpose program verification. In Mark Moriconi, editor, *ACM Sigsoft International Workshop on Formal Methods in Software Development*, pages 25–29, Napa, CA, May 1990. Published as ACM Software Engineering Notes, Vol. 15, No. 4, Sept. 1990.

[EH93] Carl T. Eichenlaub and C. Douglas Harper. Using Penelope to assess the correctness of NASA Ada software: A demonstration of formal methods as a counterpart to testing. NASA Contractor Report 4509, NASA Langley Research Center, Hampton, VA, May 1993. (Work performed by ORA Corporation).

[EKM+93] Marcin Engel, Marcin Kubica, Jan Madey, David Lorge Parnas, Anders P. Ravn, and A. John van Schouwen. A formal approach to computer systems requirements documentation. In Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors, *Hybrid Systems*, pages 452–474, New York, NY, 1993. Volume 736 of *Lecture Notes in Computer Science*, Springer-Verlag.

[EL85] Dave E. Eckhardt, Jr. and Larry D. Lee. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Transactions on Software Engineering*, SE-11(12):1511–1517, December 1985.

[End93] Albert Endres. Lessons learned in an industrial software lab. *IEEE Software*, 10(5):58–61, September 1993.

[FAA-F] Fault-tolerant software. Chapter 9 of [FAA89].

[FAA-L] Latent faults. Chapter 10 of [FAA89].

[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.

[FAA93] *RTCA Inc., Document RTCA/DO-178B.* Federal Aviation Administration, January 11, 1993. Advisory Circular 20-115B.

[Fag76] Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, March 1976.

[Fag86] Michael E. Fagan. Advances in software inspection. *IEEE Transactions on Software Engineering*, SE-12(7):744–751, July 1986.

[Fai85] Richard E. Fairley. *Software Engineering Concepts.* McGraw-Hill, New York, NY, 1985.

[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.

[FEG92] Bob Fields and Morten Elvang-Gøransson. A VDM case study in *mural*. *IEEE Transactions on Software Engineering*, 18(4):279–295, April 1992.

[Fei80] R. J. Feiertag. A technique for proving specifications are multilevel secure. Technical Report CSL-109, Computer Science Laboratory, SRI International, Menlo Park, CA, January 1980.

[Fen93] Norman Fenton. How effective are software engineering methods? *Journal of Systems and Software*, 22:141–146, 1993.

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

[FGJM85] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian K. Reid, editor, *12th ACM Symposium on Principles of Programming Languages*, pages 52–66. Association for Computing Machinery, 1985.

[FGT93] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. *Journal of Automated Reasoning*, 11(2):213–248, October 1993.

[Flo67] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, pages 19–32, Providence, RI, 1967. American Mathematical Society.

[FLR77] R. J. Feiertag, K. N. Levitt, and L. Robinson. Proving multilevel security of a system design. In *Sixth ACM Symposium on Operating System Principles*, pages 57–65, November 1977.

[Fre92] Heinz H. Frey, editor. *Safety of Computer Control Systems (SAFECOMP '92)*, Zurich, Switzerland, October 1992. International Federation of Automatic Control.

[Fro85] Robert A. Frosch. Getting it all under control. *IEEE Control Systems Magazine*, 5(1):3–8, February 1985. Keynote speech to the 1984 American Control Conference.

[GAO92a] *Embedded Computer Systems: Significant Software Problems On C-17 Must Be Addressed.* United States General Accounting Office, Washington, DC, May 1992. GAO/IMTEC-92-48.

[GAO92b] *Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia.* United States General Accounting Office, Washington, DC, February 1992. GAO/IMTEC-92-26.

[GAO93] *Aircraft Certification: New FAA Approach Needed to Meet Challenges of Advanced Technology.* United States General Accounting Office, Washington, DC, September 1993. GAO/RCED-93-155.

[Gar81] John R. Garman. The "bug" heard 'round the world. *ACM Software Engineering Notes*, 6(5):3–10, October 1981.

[GAS89] Donald I. Good, Robert L. Akers, and Lawrence M. Smith. Report on Gypsy 2.05. Technical Report 1, Computational Logic Inc., Austin, TX, January 1989.

[GCR93] Susan Gerhart, Dan Craigen, and Ted Ralston. Observations on industrial practice using formal methods. In *15th International Conference on Software Engineering*, pages 24–33, Baltimore, MD, May 1993. IEEE Computer Society.

[Ger78] S. German. Automating proofs of the absence of common runtime errors. In *Proceedings, 5th ACM Symposium on the Principles of Programming Languages*, pages 105–118, Tucson, AZ, January 1978.

[GH78] John V. Guttag and James J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10(1):27–52, 1978.

[GH80] John Guttag and J. J. Horning. Formal specification as a design tool. In *7th ACM Symposium on Principles of Programming Languages*, pages 251–261, Las Vegas, NV, January 1980.

[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.

[GH93a] Hubert Garavel and René-Pierre Hautbois. An experience with the LOTOS formal description technique on the flight warning computer of the Airbus 330/340 aircrafts. In *First AMAST International Workshop on Real-Time Systems*, Iowa City, IA, November 1993. Springer-Verlag Workshops in Computing. (To appear).

[GH93b] John V. Guttag and James J. Horning with S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, Cambridge, UK, 1993.

[GMT+80] S. L. Gerhart, D. R. Musser, D. H. Thompson, D. A. Baker, R. L. Bates, R. W. Erickson, R. L. London, D. G. Taylor, and D. S. Wile. An overview of Affirm: A specification and verification system. In S. H. Lavington, editor, *Information Processing '80*, pages 343–347, Australia, October 1980. IFIP, North-Holland Publishing Company.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[Gol87] J. Goldberg. A history of research in fault-tolerant computing at SRI International. In A. Avižienis, H. Kopetz, and J. C. Laprie, editors, *The Evolution of Fault-Tolerant Computing*, volume 1 of *Dependable Computing and Fault-Tolerant Systems*, pages 101–119. Springer-Verlag, Vienna, Austria, 1987.

[Goo70] Donald I. Good. *Toward a Man-Machine System for Proving Program Correctness.* PhD thesis, University of Wisconsin, 1970.

[Gor86] M. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 153–177. Elsevier, 1986. Reprinted in Yoeli [Yoe90, pp. 57–77].

[GS93] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math.* Texts and Monographs in Computer Science. Springer-Verlag, New York, NY, 1993.

[Gup92] Aarti Gupta. Formal hardware verification methods: A survey. *Formal Methods in Systems Design*, 1(2/3):151–238, October 1992.

[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.

[H⁺78] K. L. Heninger et al. Software requirements for the A-7E aircraft. NRL Report 3876, Naval Research Laboratory, November 1978.

[Hal90] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.

[Ham92] Dick Hamlet. Are we testing for true reliability? *IEEE Software*, 9(4):21–27, July 1992.

[Har86] David Harel. Statecharts: A visual approach to complex systems. Technical report, MCC, Austin, TX, February 1986.

[Har91] W. T. Harwood. Proof rules for Balzac. Technical Report WTH/P7/001, Imperial Software Technology, Cambridge, UK, 1991.

[Hay93] Ian Hayes, editor. *Specification Case Studies*. Prentice-Hall International Ltd., Hemel Hempstead, UK, second edition, 1993.

[HB86] James R. Hoelscher and James B. Balliet. Microprocessor-based interlocking: Concept to application. In *Proceedings, Rail Transit Conference*, Miami, FL, June 1986. American Public Transit Association.

[HD93] Kenneth Hoyme and Kevin Driscoll. SAFEbus™. *IEEE Aerospace and Electronic Systems Magazine*, 8(3):34–39, March 1993.

[HDHR91] Kenneth Hoyme, Kevin Driscoll, Jack Herrlin, and Kathie Radke. ARINC 629 and SAFEbus™: Data buses for commercial aircraft. *Scientific Honeyweller*, pages 57–70, Fall 1991.

[HDL89] F. Keith Hanna, Neil Daeche, and Mark Longley. Specification and verification using dependent types. *IEEE Transactions on Software Engineering*, 16(9):949–964, September 1989.

[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.

[Hel86] K. A. Helps. Some verification tools and methods for airborne safety-critical software. *IEE/BCS Software Engineering Journal*, 1(6):248–253, November 1986.

[Hen80] K. L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering*, SE-6(1):2–13, January 1980.

[Hen93] Peter Henderson. *Object-Oriented Specification and Design with C++*. McGraw-Hill, New York, NY, October 1993.

[HI88] Sharam Hekmatpour and Darrel Ince. *Software Prototyping, Formal Methods, and VDM*. International Computer Science Series. Addison-Wesley, Wokingham, England, 1988.

[Hil88] Paul N. Hilfinger. An Ada package for dimensional analysis. *ACM Transactions on Programming Languages and Systems*, 10(2):189–203, April 1988.

[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.

[HK91] Iain Houston and Steve King. CICS project report: Experiences and results from the use of Z in IBM. In Prehn and Toetenel [PT91], pages 588–596.

[HL91] Richard E. Harper and Jaynarayan H. Lala. Fault-tolerant parallel processor. *AIAA Journal of Guidance, Control, and Dynamics*, 14(3):554–563, May–June 1991.

[Hoa69] C. A. R. Hoare. An axiomatic basis of computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, 1985.

[Hoa89] C. A. R. Hoare. *Essays in Computing Science*, edited by C. B. Jones. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, 1989.

[Hod83] Wilfred Hodges. Elementary predicate logic. In Dov M. Gabbay and Franz Guenthner, editors, *Handbook of Philosophical Logic–Volume I: Elements of Classical Logic*, volume 164 of *Synthese Library*, chapter I.1, pages 1–131. D. Reidel Publishing Company, Dordrecht, Holland, 1983.

[Hol86] E. Keith Holt. The application of microprocessors to interlocking logic. In *Proceedings, Rail Transit Conference*, Miami, FL, June 1986. American Public Transit Association.

[Hol87] Gergory J. Holt. The certification challenge of the high technology aircraft of the 1990's. In *Aerospace Technology Conference and Exposition*, Long Beach, CA, October 1987. SAE Technical Paper Series, Paper 871842, Society of Automotive Engineers.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[HOPW87] J. D. Halpern, S. Owre, N. Proctor, and W. F. Wilson. Muse—a computer assisted verification system. *IEEE Transactions on Software Engineering*, 13(2):151–156, February 1987.

[HS91] M. C. McElvany Hugue and P. David Stotts. Guaranteed task deadlines for fault-tolerant workloads with conditional branches. *Real-Time Systems*, 3(3):275–305, September 1991.

[Hue80] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–812, October 1980.

[HV93] Dick Hamlet and Jeff Voas. Faults on its sleeve: Amplifying software reliability testing. In Thomas Ostrand and Elaine Weyuker, editors, *1993 International Symposium on Software Testing and Analysis (ISSTA)*, pages 89–98, Cambridge, MA, June 1993. Association for Computing Machinery. Published as part of ACM Software Engineering Notes, Vol. 18, No. 3, July 1993.

[HW73] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2(4):335–355, 1973.

[IEC86] *Software for Computers in the Safety Systems of Nuclear Power Stations*. International Electrotechnical Commission, Geneva, Switzerland, first edition, 1986. IEC Standard 880.

[IEE93] *Fault Tolerant Computing Symposium 23*, Toulouse, France, June 1993. IEEE Computer Society.

[ILL75] S. Igarishi, R. L. London, and D. C. Luckham. Automatic program verification I: A logical basis and its implementation. *Acta Informatica*, 4:145–182, 1975.

[Inc88] D. C. Ince. *An Introduction to Discrete Mathematics and Formal System Specification*. Oxford Applied Mathematics and Computing Science Series. Oxford University Press, Oxford, UK, 1988.

[IRM84] Stephen D. Ishmael, Victoria A. Regenie, and Dale A. Mackall. Design implications from AFTI/F16 flight test. NASA Technical Memorandum 86026, NASA Ames Research Center, Dryden Flight Research Facility, Edwards, CA, 1984.

[ISO88] *LOTOS—A Formal Description Technique Based on the Temporal Ordering of Observational Behavior*. International Organization for Standardization—Information Processing Systems—Open Systems Interconnection, Geneva, Switzerland, September 1988. ISO Standard 8807.

[JB92] Sally C. Johnson and Ricky W. Butler. Design for validation. *IEEE Aerospace and Electronic Systems Magazine*, 7(1):38–43, January 1992. Also in 10th AIAA/IEEE Digital Avionics Systems Conference, Los Angeles, CA, Oct. 1991, pp. 487–492.

[JJLM91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A formal Development Support System*. Springer-Verlag, London, UK, 1991.

[JL89] Matthew S. Jaffe and Nancy G. Leveson. Completeness, robustness, and safety in real-time software requirements specification. In *11th International Conference on Software Engineering*, pages 302–311, Pittsburgh, PA, May 1989. IEEE Computer Society.

[JLHM91] Matthew S. Jaffe, Nancy G. Leveson, Mats P. E. Heimdahl, and Bonnie E. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, March 1991.

[Jon86] Cliff B. Jones. *Software Development: A Rigorous Approach*. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, 1986.

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

[Jon92] Cliff B. Jones. The search for tractable ways of reasoning about programs. Technical Report UMCS-92-4-4, Department of Computer Science, University of Manchester, Manchester, UK, March 1992.

[K+89] Hermann Kopetz et al. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, 9(1):25–40, February 1989.

[KB70] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–293. Pergamon, New York, NY, 1970.

[Kem86] Richard A. Kemmerer. Verification assessment study final report. Technical Report C3-CR01-86, National Computer Security Center, Ft. Meade, MD, 1986. 5 Volumes (Overview, Gypsy, Affirm, FDM, and EHDM). US distribution only.

[Keu91] Kurt Keutzer. The need for formal verification in hardware design and what formal verification has not done for me lately. In Windley [Win91], pages 77–86.

[Kin69] J. C. King. *A Program Verifier*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1969.

[Kir89] Hubert Kirrmann. Fault-tolerant computing in Europe. *IEEE Micro*, 9(2):5–7, April 1989.

[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.

[Kle91] Israel Kleiner. Rigor and proof in mathematics: A historical perspective. *Mathematics Magazine*, 64(5):291–314, December 1991. Published by the Mathematical Association of America.

[Kli90] Morris Kline. *Mathematics in Western Culture*. Penguin Books, 1990. (Originally published 1953).

[KM86] Deepak Kapur and David R. Musser. The RSRE crypto controller example in Affirm-85. In *Verification Assessment Study Final Report* [Kem86], pages 309–318.

[KS80] Deepak Kapur and Mandayam Srivas. Expressiveness of the operation set of a data abstraction. In *7th ACM Symposium on Principles of Programming Languages*, pages 139–153, Las Vegas, NV, January 1980.

[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.

[KSQ92] Brian W. Kowal, Carl J. Scherz, and Richard Quinliven. C-17 flight control system overview. *IEEE Aerospace and Electronic Systems Magazine*, 7(7):24–31, July 1992.

[Kur90] R. P. Kurshan. Analysis of discrete event coordination. In *Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*. Volume 430 of *Lecture Notes in Computer Science*, Springer-Verlag, 1990.

[KWFT88] R. M. Kieckhafer, C. J. Walter, A. M. Finn, and P. M. Thambidurai. The MAFT architecture for distributed fault tolerance. *IEEE Transactions on Computers*, 37(4):398–405, April 1988.

[KZ88] D. Kapur and H. Zhang. RRL: A rewrite rule laboratory. In Lusk and Overbeek [LO88], pages 768–769.

[Lak76] Imre Lakatos. *Proofs and Refutations*. Cambridge University Press, Cambridge, England, 1976.

[Lam87] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

[Lam91] Leslie Lamport. The temporal logic of actions. Technical Report 79, DEC Systems Research Center, Palo Alto, CA, December 1991. To Appear in ACM Transactions on Programming Languages and Systems.

[Lap91] J. C. Laprie, editor. *Dependability: Basic Concepts and Terminology in English, French, German, Italian and Japanese*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, Vienna, Austria, February 1991.

[Lap92] Jean-Claude Laprie. For a product-in-a-process approach to software reliability evaluation. In *3rd International Symposium on Software Reliability Engineering*, pages 134–139, Research Triangle Park, NC, October 1992. IEEE Computer Society.

[LC86] Arthur L. Liestman and Roy H. Campbell. A fault-tolerant scheduling problem. *IEEE Transactions on Software Engineering*, SE-12(11):1089–1095, November 1986.

[Lea88] David Learmount. A320 certification: The quiet revolution. *Flight International*, pages 21–24, February 27, 1988.

[Lea93] David Learmount. SAS unaware of key system on crash MD-81. *Flight International*, pages 8–9, November 10, 1993.

[Lei69] A. C. Leisenring. *Mathematical Logic and Hilbert's ε-Symbol*. Gordon and Breach Science Publishers, New York, NY, 1969.

[Les86] Pierre Lescanne. REVE: a rewrite rule laboratory. In J. H. Siekmann, editor, *8th International Conference on Automated Deduction (CADE)*, pages 695–696, Oxford, England, July 1986. Volume 230 of *Lecture Notes in Computer Science*, Springer-Verlag.

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

[Lev91] Nancy G. Leveson. Software safety in embedded computer systems. *Communications of the ACM*, 34(2):34–46, February 1991.

[LGvH+79] D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. Stanford Pascal Verifier user manual. CSD Report STAN-CS-79-731, Stanford University, Stanford, CA, March 1979.

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

[Lin88] Kenneth S. Lindsay. A taxonomy of the causes of proof failures in applications using the HDM methodology. In *Fourth Aerospace Computer Security Applications Conference*, pages 419–423, Orlando, FL, December 1988. IEEE Computer Society. Reprinted in [COMP89, pp. 79–83].

[LM89] B. Littlewood and D. R. Miller. Conceptual modeling of coincident failures in multiversion software. *IEEE Transactions on Software Engineering*, 15(12):1596–1614, December 1989.

[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.

[LO88] E. Lusk and R. Overbeek, editors. *9th International Conference on Automated Deduction (CADE)*, volume 310 of *Lecture Notes in Computer Science*, Argonne, IL, May 1988. Springer-Verlag.

[Loc92] C. Douglass Locke. Software architecture for hard real-time applications: Cyclic executives vs. priority executives. *Real-Time Systems*, 4(1):37–53, March 1992.

[LR93a] Patrick Lincoln and John Rushby. Formal verification of an algorithm for interactive consistency under a hybrid fault model. In Courcoubetis [Cou93], pages 292–304.

[LR93b] Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Fault Tolerant Computing Symposium 23* [IEE93], pages 402–411.

[LRS92] C. E. Landwehr, B. Randell, and L. Simoncini, editors. *Dependable Computing for Critical Applications—3*, volume 8 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, Vienna, Austria, September 1992.

[LS91] Matthew Lee and Ib H. Sørensen. B-Tool. In Prehn and Toetenel [PT91], pages 695–696.

[LS93] Bev Littlewood and Lorenzo Strigini. Validation of ultrahigh dependability for software-based systems. *Communications of the ACM*, pages 69–80, November 1993.

[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.

[LSSE80] R. Locasso, J. Scheid, D. V. Schorre, and P. R. Eggert. *The Ina Jo Specification Language Reference Manual*. System Development Corporation (now Paramax), Santa Monica, CA, November 1980. TM-6889/000/01.

[LSST83] Nancy G. Leveson, Timothy J. Shimeall, Janice L. Stolzy, and Jeffrey C. Thomas. Design for safe software. In *Proc. AIAA 21st Aerospace Sciences Meeting*, Reno NV, January 1983. American Institute of Aeronautics and Astronautics.

[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.

[Luc90] David Luckham. *Programming with Specifications: An Introduction to ANNA, A Language for Specifying Ada Programs.* Texts and Monographs in Computer Science. Springer-Verlag, New York, NY, 1990.

[Lut93a] 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.

[Lut93b] Robyn R. Lutz. Targeting safety-related errors during software requirements analysis. In *SIGSOFT '93: Symposium on the Foundations of Software Engineering*, pages 99–106, Los Angeles, CA, December 1993. Published as ACM Software Engineering Notes, Vol. 18, No. 5, Dec. 1993.

[LvHKBO87] David C. Luckham, Friedrich W. von Henke, Bernd Krieg-Brückner, and Olaf Owe. *ANNA: A Language for Annotating Ada Programs*, volume 260 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[MA89a] Dale A. Mackall and James G. Allen. A knowledge-based system design/information tool for aircraft flight control systems. In *AIAA Computers in Aerospace Conference VII*, pages 110–125, Monterey, CA, October 1989. Collection of Technical Papers, Part 1.

[MA89b] John D. Musa and A. Frank Ackerman. Quantifying software validation: When to stop testing? *IEEE Software*, 6(3):19–27, May 1989.

[Mac84] Dale A. Mackall. AFTI/F-16 digital flight control system experience. In Gary P. Beasley, editor, *NASA Aircraft Controls Research 1983*, pages 469–487. NASA Conference Publication 2296, 1984. Proceedings of workshop held at NASA Langley Research Center, October 25–27, 1983.

[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.

[Mac91] Donald MacKenzie. The fangs of the VIPER. *Nature*, 352(6335):467–468, August 8, 1991.

[Mar92] M. Jean Martin. Vital processing by single coded unit. In Frey [Fre92], pages 147–149.

[MB90] C. Mason and D. Bushaus. Software problem cripples AT&T long-distance network. *Telephony*, 218(4):10, January 22, 1990.

[MC90] Derek P. Mannering and Bernard Cohen. The rigorous specification and verification of the safety aspects of a real-time system. In COMP [COMP90], pages 68–85.

[McA89] David A. McAllester. *ONTIC: A Knowledge Representation System for Mathematics*. MIT Press, Cambridge, MA, 1989.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):184–195, 1960.

[McC90] W. W. McCune. OTTER 2.0 users guide. Technical Report ANL-90/9, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, March 1990.

[McD93] John McDermid, editor. *Software Engineer's Reference Book*. CRC Press, Boca Raton, FL, 1993. Hardback edition published in UK by Butterworth-Heinemann Ltd., 1991.

[McG90] John G. McGough. The Byzantine Generals Problem in flight control systems. In *AIAA Second International Aerospace Planes Conference*, Orlando, FL, October 1990. AIAA paper 90-5210.

[MD93] Ralph Melton and David L. Dill. *Murϕ Annotated Reference Manual*. Computer Science Department, Stanford University, Stanford, CA, March 1993.

[MDL87] Harlan D. Mills, Michael Dyer, and Richard Linger. Cleanroom software engineering. *IEEE Software*, 4(5):19–25, September 1987.

[MG78] D. L. Martin and D. Gangsaas. Testing the YC-14 flight control system software. *AIAA Journal of Guidance and Control*, 1(4):242–247, July–August 1978.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[MIO87] John D. Musa, Anthony Iannino, and Kazuhira Okumoto. *Software Reliability—Measurement, Prediction, Application*. McGraw Hill, New York, NY, 1987.

[MJ84] F. L. Morris and C. B. Jones. An early program proof by Alan Turing. *IEEE Annals of the History of Computing*, 6:139–143, 1984.

[MMN+92] Keith W. Miller, Larry J. Morell, Robert E. Noonan, Stephen K. Park, David M. Nicol, Branson W. Murrill, and Jeffrey M. Voas. Estimating the probability of failure when testing reveals no failures. *IEEE Transactions on Software Engineering*, 18(1):33–43, January 1992.

[MOD91a] *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.

[MOD91b] *Interim Defence Standard 00-56: Hazard Analysis and Safety Classification of the Computer and Programmable Electronic System Elements of Defence Equipment.* UK Ministry of Defence, April 1991.

[Mon92] G. Mongardi. Dependable computing for railway control systems. In Landwehr et al. [LRS92], pages 255–277.

[MP67] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Mathematical Aspects of Computer Science, Proc. Symp. Appl. Math., Vol. XIX*, pages 33–41. American Mathematical Society, 1967.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Volume 1: Specification.* Springer-Verlag, New York, NY, 1992.

[MS91] K. L. McMillan and J. Schwalbe. Formal verification of the Gigamax cache-consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 242–251. Information Processing Society of Japan, 1991.

[MSS82] P. M. Melliar-Smith and R. L. Schwartz. Formal specification and verification of SIFT: A fault-tolerant flight control system. *IEEE Transactions on Computers*, C-31(7):616–630, July 1982.

[Mus80] D. R. Musser. Abstract data type specification in the Affirm system. *IEEE Transactions on Software Engineering*, 6(1):24–32, January 1980.

[MW93] Zohar Manna and Richard Waldinger. *The Deductive Foundations Computer Programming.* Addison-Wesley, 1993. The One-Volume version of “The Logical Basis for Computer Programming”.

[NAS83] *Peer Review of a Formal Verification/Design Proof Methodology.* NASA Conference Publication 2377, July 1983.

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

[Neu92] Peter G. Neumann. Illustrative risks to the public in the use of computer systems and related technology. *ACM Software Engineering Notes*, 17(1):23–32, January 1992.

[Nic89] J. E. Nicholls, editor. *Z User Workshop*, Oxford, UK, December 1989. Springer-Verlag Workshops in Computing.

[NK91] Takeshi Nakajo and Hitoshi Kume. A case history analysis of software error cause-effect relationships. *IEEE Transactions on Software Engineering*, 17(8):830–838, August 1991.

[NO79] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[Nor88] Donald A. Norman. *The Psychology of Everyday Things.* Basic Books, New York, NY, 1988. (Also available in paperback under the title "The Design of Everyday Things.").

[NY82] R. Nakajima and T. Yuasa, editors. *The IOTA Programming System*, volume 160 of *Lecture Notes in Computer Science.* Springer-Verlag, 1982.

[OCF+88] I. M. O'Neil, D. L. Clutterbuck, P. F. Farrow, P. G. Summers, and W. C. Dolman. The formal verification of safety-critical assembly code. In W. D. Ehrenberger, editor, *Safety of Computer Control Systems (SAFECOMP '88)*, pages 115–120, Fulda, Germany, 1988. International Federation of Automatic Control, IFAC Proceedings Series No. 16.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, pages 748–752, Saratoga, NY, June 1992. Volume 607 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag.

[Par72] David L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.

[Par93] David Lorge Parnas. Some theorems we should prove. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications (6th International Workshop, HUG '93)*, pages 155–162, Vancouver, Canada, August 1993. Number 780 in Lecture Notes in Computer Science, Springer-Verlag.

[Pat92] Ozello Patrick. The coded microprocessor certification. In Frey [Fre92], pages 185–190.

[Pau87] L. C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, Cambridge, England, 1987.

[Pau88] Lawrence C. Paulson. Isabelle: The next seven hundred theorem provers. In Lusk and Overbeek [LO88], pages 772–773.

[Pau92] Lawrence C. Paulson. Designing a theorem prover. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science; Volume 2 Background: Computational Structures*, pages 415–475. Oxford Science Publications, Oxford, UK, 1992.

[Per84] Charles Perrow. *Normal Accidents: Living with High Risk Technologies*. Basic Books, New York, NY, 1984.

[Pet85] Henry Petroski. *To Engineer is Human: The Role of Failure in Successful Design*. St. Martin's Press, New York, NY, 1985. Also Vintage Books paperback edition, 1992.

[Phi93] Edward H. Phillips. Wing slat control cited in Chinese MD-11 accident. *Aviation Week and Space Technology*, pages 37–38, November 8, 1993.

[Pla93] Nico Plat. *Experiments with Formal Methods in Software Engineering*. PhD thesis, Technische Universiteit Delft, The Netherlands, 1993.

[Pnu81] Amir Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–80, 1981.

[Pot93] Colin Potts. Software-engineering research revisited. *IEEE Software*, 10(5):19–28, September 1993.

[Pow92] David Powell. Failure mode assumptions and assumption coverage. In *Fault Tolerant Computing Symposium 22*, pages 386–395, Boston, MA, July 1992. IEEE Computer Society.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[PT91] S. Prehn and W. J. Toetenel, editors. *VDM '91: Formal Software Development Methods*, volume 551 of *Lecture Notes in Computer Science*, Noordwijkerhout, The Netherlands, October 1991. Springer-Verlag. Volume 1: Conference Contributions.

[PvSK90] David L. Parnas, A. John van Schouwen, and Shu Po Kwan. Evaluation of safety-critical software. *Communications of the ACM*, 33(6):636–648, June 1990.

[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.

[Pyg88] C. H. Pygott. NODEN: An engineering approach to hardware verification. Technical Report 415-88, Royal Signals and Radar Establishment, 1988.

[Pyl91] I. C. Pyle. *Developing Safety Systems: A Guide Using Ada.* Prentice Hall International (UK) Ltd, Hemel Hempstead, UK, 1991.

[RAISE92] The RAISE Language Group. *The RAISE Specification Language.* BCS Practioner Series. Prentice-Hall International, Hemel Hempstead, UK, 1992.

[Ran75a] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[Ran75b] Brian Randell, editor. *The Origins of Digital Computers: Selected Papers.* Springer-Verlag, second edition, 1975.

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

[Rei86] Constance Reid. *Hilbert—Courant.* Springer-Verlag, New York, NY, 1986. Originally two volumes: *Hilbert*, 1970, and *Courant*, 1976.

[RL76] Lawrence Robinson and Karl N. Levitt. Proof techniques for hierarchically structured programs. *Communications of the ACM*, 20(4):271–283, April 1976.

[RLS79] L. Robinson, K. N. Levitt, and B. A. Silverberg. *The HDM Handbook.* Computer Science Laboratory, SRI International, Menlo Park, CA, June 1979. Three Volumes.

[RM90] Harold E. Roland and Brian Moriarty. *System Safety Engineering Management*. John Wiley and Sons, New York, NY, second edition, 1990.

[Roy70] W. W. Royce. Managing the development of large software systems. In *Proceedings WESCON*, August 1970.

[Roy91] Winston Royce. Current problems. In Anderson and Dorfman [AD91], chapter 1.1, pages 5–15.

[RS92] Mark Ryan and Martin Sadler. Valuation systems and consequence relations. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science; Volume 1 Background: Mathematical Structures*, pages 1–78. Oxford Science Publications, Oxford, UK, 1992.

[RS93] John Rushby and Mandayam Srivas. Using PVS to prove some theorems of David Parnas. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications (6th International Workshop, HUG '93)*, pages 163–173, Vancouver, Canada, August 1993. Number 780 in Lecture Notes in Computer Science, Springer-Verlag.

[RTCA89] *Minimum Operational Performance Standards for Traffic Alert and Collision Avoidance System (TCAS) Airborne Equipment*. Radio Technical Commission for Aeronautics, Washington, DC, volume 1, consolidated edition edition, September 1989. DO-185.

[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.

[Rus81] John Rushby. The design and verification of secure systems. In *Eighth ACM Symposium on Operating System Principles*, pages 12–21, Asilomar, CA, December 1981. (ACM *Operating Systems Review*, Vol. 15, No. 5).

[Rus82] John Rushby. Proof of Separability—a verification technique for a class of security kernels. In *Proc. 5th International Symposium on Programming*, pages 352–367, Turin, Italy, April 1982. Volume 137 of *Lecture Notes in Computer Science*, Springer-Verlag.

[Rus84] John Rushby. A trusted computing base for embedded systems. In *Proceedings 7th DoD/NBS Computer Security Initiative Conference*, pages 294–311, Gaithersburg, MD, September 1984.

[Rus86] John Rushby. The crypto example. In *Verification Assessment Study Final Report* [Kem86], pages 251–255.

[Rus89] John Rushby. Kernels for safety? In T. Anderson, editor, *Safe and Secure Computing Systems*, chapter 13, pages 210–220. Blackwell Scientific Publications, 1989. (Proceedings of a Symposium held in Glasgow, October 1986).

[Rus93] John Rushby. A fault-masking and transient-recovery model for digital flight-control systems. In Jan Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Kluwer International Series in Engineering and Computer Science, chapter 5, pages 109–136. Kluwer, Boston, Dordecht, London, 1993. An earlier version appeared in [Vyt92, pp. 237–257].

[Rut90] David Rutherford. A vital digital control system with a calculable probability of an unsafe failure. In COMP [COMP90], pages 1–11.

[RvH91] John Rushby and Friedrich von Henke. Formal verification of the Interactive Convergence clock synchronization algorithm using EHDM. Technical Report SRI-CSL-89-3R, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1989 (Revised August 1991). Original version also available as NASA Contractor Report 4239, June 1989.

[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.

[RW85] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[SB89] Paul Smith and Nick Bleech. Practical experience with a formal verification system. In Sennett [Sen89], chapter 4.

[SB91] Mandayam Srivas and Mark Bickford. Verification of the FtCayuga fault-tolerant microprocessor system, volume 1: A case-study in theorem prover-based verification. Contractor Report 4381, NASA Langley Research Center, Hampton, VA, July 1991. (Work performed by ORA Corporation).

[SB92] Mandayam Srivas and Mark Bickford. Moving formal methods into practice: Verifying the FTPP Scoreboard: Phase 1 results. NASA Contractor Report 189607, NASA Langley Research Center, Hampton, VA, May 1992. (Work performed by ORA Corporation).

[Sch89] P. N. Scharbach, editor. *Formal Methods: Theory and Practice*. CRC Press, Boca Raton, FL, 1989.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[Sco92] William B. Scott. C-17 first flight triggers Douglas/Air Force test program. *Aviation Week and Space Technology*, page 21, September 23, 1992.

[Sen89] C. T. Sennett, editor. *High-Integrity Software*. Software Science and Engineering. Plenum Press, New York, NY, 1989.

[SGD88] Michael K. Smith, Donald I. Good, and Benedetto L. Di Vito. Using the Gypsy methodology. Technical Report 1, Computational Logic Inc., January 1988.

[SGGH91] James B. Saxe, Stephen J. Garland, John V. Guttag, and James J. Horning. Using transformations and verification in circuit design. Technical Report 78, DEC Systems Research Center, Palo Alto, CA, September 1991.

[Sha88] N. Shankar. Observations on the use of computers in proof checking. *Notices of the American Mathematical Society*, 35(6):804–805, July/August 1988.

[Sha92] Natarajan Shankar. Mechanical verification of a generalized protocol for Byzantine fault-tolerant clock synchronization. In Vytopil [Vyt92], pages 217–236.

[Sha93] Natarajan Shankar. Verification of real-time systems using PVS. In Courcoubetis [Cou93], pages 280–291.

[Sho67] Joseph R. Shoenfield. *Mathematical Logic*. Addison-Wesley, Reading, MA, 1967.

[Sho77] Robert E. Shostak. On the SUP-INF method for proving Presburger formulas. *Journal of the ACM*, 24(4):529–543, October 1977.

[Sho78] Robert E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7):583–585, July 1978.

[Sho79] Robert E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, April 1979.

[Sho84] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[SIG91] *SIGSOFT '91: Software for Critical Systems*, New Orleans, LA, December 1991. Published as ACM SIGSOFT Engineering Notes, Volume 16, Number 5.

[Sil83] J. M. Silverman. Reflections on the verification of the security of an operating system kernel. In *Ninth ACM Symposium on Operating System Principles*, pages 143–154, Bretton Woods, NH, October 1983. (ACM Operating Systems Review, Vol 17, No. 5).

[SLR78] Jay M. Spitzen, Karl N. Levitt, and Lawrence Robinson. An example of hierarchical design and proof. *Communications of the ACM*, 21(12):1064–1075, December 1978.

[SM92] Tom Sadeghi and Farzin Motamed. Evaluation and comparison of triple and quadruple flight control architectures. *IEEE Aerospace and Electronic Systems Magazine*, 7(3):20–31, March 1992.

[SMT92] G. Michael Schneider, Johnny Martin, and W. T. Tsai. An experimental study of fault detection in user requirements. *ACM Transactions on Software Engineering Methodology*, 1(2):188–204, April 192.

[Sol82] Jane O. Solomon. Specification-to-code correlation. In *Proceedings of the Symposium on Security and Privacy*, pages 81–83, Oakland, CA, April 1982. IEEE Computer Society.

[Sol90] Daniel Solow. *How to Read and do Proofs: An Introduction to Mathematical Thought Processes*. John Wiley and Sons, New York, NY, second edition, 1990.

[Sør81] Ib Holm Sørensen. A specification language. In J. Staunstrup, editor, *Program Specification*, pages 381–401, Aarhus, Denmark, August 1981. Volume 134 of *Lecture Notes in Computer Science*, Springer-Verlag.

[Spi87] Cary R. Spitzer. *Digital Avionics Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[Spi88] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics.* Cambridge Tracts in Theoretical Computer Science 3. Cambridge University Press, Cambridge, UK, 1988.

[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.

[SR90] John A. Stankovic and Krithi Ramamritham. What is predictability for real-time systems? *Real-Time Systems*, 2(4):247–254, November 1990. (Editorial).

[SS84] D. V. Schorre and J. Stein. *The Interactive Theorem Prover (ITP) Manual.* System Development Corporation (now Paramax), Santa Monica, CA, 1984. TM-6889/006/01.

[Tho92] Haydn A. Thompson. *Parallel Processing for Jet Engine Control.* Advances in Industrial Control. Springer-Verlag, London, UK, 1992.

[Tic85] Walter F. Tichy. *RCS—A System for Version Control.* Department of Computer Sciences, Purdue University, West Lafayette, IN, July 1985.

[Tom87] James E. Tomayko. *Computers in Spaceflight: The NASA Experience*, volume 18, supplement 3 of *Encyclopaedia of Computer Science and Technology.* Marcel Dekker, New York, NY, 1987.

[TP88] Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *7th Symposium on Reliable Distributed Systems*, pages 93–100, Columbus, OH, October 1988. IEEE Computer Society.

[Tra90] Paul Travis. Why the AT&T network crashed. *Telephony*, 218(4):11, January 22, 1990.

[Tur92] A. M. Turing. Checking a large routine. In D. C. Ince, editor, *Collected Works of A. M. Turing: Mechanical Intelligence*, pages 129–131. North-Holland, Amsterdam, The Netherlands, 1992. (Originally presented at EDSAC Inaugural Conference on High Speed Automatic Calculating Machines, 24 June, 1949).

[Voa92] Jeffrey M. Voas. A dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, August 1992.

[vS90] A. John van Schouwen. The A-7 requirements model: Re-examination for real-time systems and an application to monitoring systems. Technical Report 90-276, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, May 1990.

[vSPM93] A. John van Schouwen, David Lorge Parnas, and Jan Madey. Documentation of requirements for computer systems. In *IEEE International Symposium on Requirements Engineering*, pages 198–207, San Diego, CA, January 1993.

[VW84] Iris Vessey and Ron Weber. Research on structured programming: An empiricist's evaluation. *IEEE Transactions on Software Engineering*, SE-10(4):397–407, July 1984.

[Vyt92] J. Vytopil, editor. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, Nijmegen, The Netherlands, January 1992. Springer-Verlag.

[WAH+93] Stephanie White, Mack Alford, Julian Holtzman, Stephen Kuehl, Brian McCay, David Oliver, David Owens, Colin Tully, and Allan Willey. Systems engineering of computer-based systems. *IEEE Computer*, 26(11):54–65, November 1993.

[Wei88] C. Weissman. Blacker: Security for the DDN. Examples of A1 security engineering trades. In *Proceedings of the Symposium on Security and Privacy*, page 26, Oakland, CA, April 1988. IEEE Computer Society. Publication clearance for this paper was rescinded; it finally appeared in the Proceedings of the 1992 Symposium on Research in Security and Privacy, pages 286–292.

[Wel93] Edward F. Weller. Lessons from three years of inspection data. *IEEE Software*, 10(5):38–45, September 1993.

[Wey88] Elaine J. Weyuker. The evaluation of program-based software test data adequacy criteria. *Communications of the ACM*, 31(6):668–675, June 1988.

[WHCC80] A. Y. Wei, K. Hiraishi, R. Cheng, and R. H. Campbell. Application of the fault-tolerant deadline mechanism to a satellite on-board computer system. In *Fault Tolerant Computing Symposium 10*, pages 107–109, Kyoto, Japan, June 1980. IEEE Computer Society.

[Wie93] Lauren Ruth Wiener. *Digital Woes: Why We Should Not Depend on Software*. Addison-Wesley, Reading, MA, 1993.

[Wil90] John Williams. Built to last. *Astronomy Magazine*, 18(12):36–41, December 1990.

[Win91] Phillip Windley, editor. *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, Davis, CA, August 1991. IEEE Computer Society.

[Wit83] Ludwig Wittgenstein. *Remarks on the Foundations of Mathematics*, edited by G. H. von Wright, R. Rhees, and G. E. M. Anscombe. MIT Press, Cambridge, MA, revised edition, 1983. Translated by G. E. M. Anscombe.

[WKP80] B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, February 1980.

[WL89] Jim Woodcock and Martin Loomes. *Software Engineering Mathematics*. SEI Series in Software Engineering. Addison Wesley, Reading, MA, 1989.

[WLG+78] John H. Wensley, Leslie Lamport, Jack Goldberg, Milton W. Green, Karl N. Levitt, P. M. Melliar-Smith, Robert E. Shostak, and Charles B. Weinstock. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, October 1978.

[Wor92] J. B. Wordsworth. *Software Development with Z*. Addison-Wesley, Wokingham, England, 1992.

[WW93] Debora Weber-Wulff. Proof Movie—a proof with the Boyer-Moore prover. *Formal Aspects of Computing*, 5(2):121–151, 1993.

[XP91] Jia Xu and David Lorge Parnas. On satisfying timing constraints in hard-real-time systems. In SIG [SIG91], pages 132–146.

[YM87] William D. Young and John McHugh. Coding for a believable specification to implementation mapping. In *Proceedings of the Symposium on Security and Privacy*, pages 140–148, Oakland, CA, April 1987. IEEE Computer Society.

[YN85] T. Yuasa and R. Nakajima. IOTA: A modular programming system. *IEEE Transactions on Software Engineering*, SE-11(2):179–187, February 1985.

[Yoe90] Michael Yoeli, editor. *Formal Verification of Hardware Design*. IEEE Computer Society, Los Alamitos, CA, 1990.

[You92] William D. Young. Verifying the Interactive Convergence clock-synchronization algorithm using the Boyer-Moore prover. NASA Contractor Report 189649, NASA Langley Research Center, Hampton, VA, April 1992. (Work performed by Computational Logic Incorporated).

Appendix A

A Rapid Introduction to Mathematical Logic

> *"...our interest in formalized languages being less often in their actual and practical use as languages than in the general theory of such use and its possibilities in principle."* [Chu56, page 47]
>
> *"The formal systems of logic were created in order to be studied, not in order to be used. It is an interesting exercise to try to formalize Hardy and Wright's number theory book in Peano Arithmetic. Any logician will see that it can theoretically be done. But to do it in practice is far too cumbersome... This has not bothered logicians, who...have not been interested in actually formalizing anything, but only in the possibilities of doing so."* [Bee86, pages 53–54]

Formal methods are grounded in mathematical logic, and some familiarity with the main concepts and terminology of that field are essential to an understanding of formal methods. Unfortunately, most textbooks on formal methods introduce only the fragments of logic relevant to their particular methods and leave the reader uninformed of alternatives. Textbooks on mathematical logic are also unsatisfactory for those seeking an introduction to the background of formal methods: elementary texts such as [Hod77, Cop67, Lem87], while excellent for their intended purpose, omit much that is of importance concerning logic as a foundation for formal methods in computer science, while more advanced texts such as [Chu56, End72, Kle52, Men64, Mon76, Sho67] are rather challenging, yet still omit some of the topics necessary for an appreciation of the mechanization of mathematical logic in support of formal methods (while including many that are unnecessary for that purpose).

To help readers gain the necessary background, I provide in this appendix a minimal introduction to mathematical logic without all the technicalities usually introduced in logic texts. My main goal is simply to acquaint the reader with the main concepts and terminology of the field. There is, however, one important topic where the going does get a little technical: this is the explanation of *interpretation* and *model* in Sections A.2, A.3, and A.4. An appreciation of these concepts is necessary in order to understand the relationship between "true" and "provable," and between the notions "sound" and "complete," and I urge readers to whom this material is new to persevere with it. Recent controversies, such as that surrounding the British Viper microprocessor (see Section 2.5.2 in the main text) show that the formal notion of "proof" is no longer of merely specialist interest: all those concerned with formal methods and certification should understand the technical meaning of "proof" as used in logic. Later sections that deal with somewhat esoteric topics that can be skipped at first reading are marked with the "dangerous bend" sign ☡.

A secondary goal of this presentation is to sketch the origins and motivations for some of the foundational approaches taken in mathematical logic, and to identify some of the alternatives that are often omitted from elementary introductions to logic. The reason for doing this can be found in the quotations at the head of this chapter: mathematical logic was developed by mathematicians to address matters of concern to them. Initially, those concerns were to provide a minimal and self-evident foundation for mathematics; later, technical questions about logic itself became important. For these reasons, much of mathematical logic is set up for *meta*mathematical purposes: to show that certain elementary concepts allow some parts of mathematics to be formalized *in principle*, and to support (relatively) simple proofs of properties such as soundness and completeness. Formal methods in computer science, on the other hand, is concerned with formalizing requirements, designs, algorithms and programs, and with developing formal proofs *in practice*. Thus, formal systems and approaches that are very suitable for mathematicians' purposes may not be so suitable as a foundation for formal methods in computer science—especially when mechanized checking of formalizations is desired. This is particularly the case, in my opinion, with axiomatic set theory[1]—yet many books on formal methods take their lead from the mathematicians and use axiomatic set theory and Hilbert-style formalizations of predicate calculus as a foundation. Those who are curious about alternatives and might want to learn something about type theory, or the sequent calculus, say, must generally turn to rather advanced texts on mathematical logic. So although this Appendix cannot provide an adequate introduction to these alternatives, it does at least sketch them, and describe the motivations behind the various different approaches, and some of their strengths and weaknesses. I hope that readers who persevere to the end will find this mate-

[1]This is not to say that the notation of set theory is not useful in formal methods; the question is whether set theory should be the *foundation* for formal methods.

rial helps them to become more knowledgeable and critical in their use of formal methods.

A.1 Introduction

Logic is concerned with methods for sound reasoning. More particularly, *mathematical* or *formal* logic is concerned with methods that are sound because of their *form*, and independent of their content. For example, the deduction

> That jaberwocky is a tove;
> All toves are slithy;
> Therefore that jaberwocky is slithy

seems perfectly sound, because of its form, even though we have no idea what a *jaberwocky* or a *tove* might be, nor what it means to be *slithy*. On the other hand, the deduction

> That plane is a Boeing 737;
> Therefore it has two engines

is not logically sound, even though its conclusion is true, because the line of reasoning employed is not generally valid—it jumps to a conclusion that is not supported by the facts *explicitly* enumerated. The argument used has the same form as

> That car is a Chrysler;
> Therefore it has two engines,

which is palpable nonsense. We can correct the problem by adding a premise to make explicit the "knowledge" used in coming to the conclusion:

> That plane is a Boeing 737;
> All Boeing planes, except the 747, have two engines;
> Therefore that plane has two engines.

The line of reasoning is now sound; it doesn't depend on the meaning of "Boeing 737" and the other terms appearing in its statement. To see this, we can abstract the argument to the form

> That A is an X;
> P is true of all A's, except Y;
> Therefore P is true of that A

and observe that it seems to be sound independently of what A, X, Y, or P mean.[2]

Note, however, that although the soundness of this line of reasoning does not depend on what a "Boeing 737" is, it depends crucially on what "all," "except," and "and" mean. Logic chooses some elements of language (such as "all," and "and"), gives them a very precise meaning, and then investigates the lines of reasoning and argument that can be built up from those elements; in particular, logic seeks to identify those lines of reasoning that are valid independently of the meaning of the other (so-called proper) terms that appear in the argument. Logics differ from one another in their choice of primitive elements, in the ways in which they allow terms to be combined, and in the lines of reasoning that they permit.

Notice that a valid or sound argument does not guarantee that the conclusion we draw will be true. For example, if the plane we saw was a Boeing 707 (or 727), the perfectly valid pattern of deduction given above would cause us to deduce, incorrectly, that the plane has only two engines. A sound method of reasoning guarantees true conclusions only if the premises are true statements. The truth of a statement such as

All Boeing planes have two engines, except the 747

depends on the real-world interpretation we place on the terms such as "Boeing planes." If the interpretation we choose is "all Boeing passenger planes in current service," then this premise to our argument is false, and a perfectly valid line of reasoning can lead us to a false conclusion.

Two sources of error can lead us to draw false conclusions about the world: we can reason correctly from incorrect premises, and we can reason incorrectly from correct premises (and, of course, we can also go doubly wrong and reason incorrectly from bad premises). The contribution of mathematical logic is that it gives us the tools to eliminate the second of these ways to draw incorrect conclusions. Logic tells us what follows from what, so that we can be sure that our conclusions will be at least as good as our premises. However, no formal method can assure us that our premises are true statements in the intended interpretation; we have to introspect, discuss them with others, test their consequences, and generally use informal methods to accomplish this important task.[3]

[2] Though it does depend on knowing that an X is not a Y (i.e., a 737 is not a 747). When we come to formalize this (in Section A.4), we will have to make even this "obvious" fact explicit.

[3] However, formal methods provide a precise and standardized notation, a repertoire of standard constructions, and well-accepted sets of premises (i.e., axiomatizations) for many familiar notions. In addition, formal methods can help us establish that our premises are mutually consistent, and can support systematic exploration of their consequences (through attempts to prove properties that ought to be true if the premises are correct). Thus the informal process of validating premises can be made more systematic using formal methods.

The practical value of mathematical logic arises when we use long or difficult lines of reasoning to draw an important conclusion. If we can satisfy ourselves, by informal methods, that the premises to our argument are true in our intended interpretation, then mathematical logic can assure us that the conclusion will also be true in that interpretation.

> "The true method should provide us with...a certain sensible material means which will lead the intellect, like the lines drawn in geometry and the forms of operations which are prescribed to novices in arithmetic. Without this, our minds will not be able to travel a long path without going astray." Leibniz, quoted in [Bou68, page 303].

Mathematical logic—Leibniz' "true method"—provides us with rules of deduction, analogous to those of arithmetic, that allow the validity of an argument to be checked by a process akin to calculation.

A.2 General Concepts

In this section I give more detailed and precise explanations to some of the notions introduced above.[4] The explanations are given in terms of the abstract notion of "formal system" and are generic to all logics; I give examples of specific logics and theories in the subsections that follow.

The first thing that must be specified about a particular formal system is the language in which its statements are expressed. Just as with a programming language, there must be some grammar that defines the syntactic form of the things we can write in the system. Mathematicians tend to prefer very terse, succinct languages, so the statements in most of their formal systems look very cryptic to the untrained eye. Legal strings of symbols in the language of a formal system are called *well-formed formulas* or *wff*s. Wffs that are "self contained" (technically, contain no free variables) are called sentences.

Formal systems come in families, and there are usually two parts to a formal system: one part that is common to all the systems in a given family, and one part that changes from one member of the family to another. The first part is called the *logical* part of the system, and the second the *nonlogical* or *extralogical* or *proper* part. (I will use only the first of these alternative terms). For example, all *first-order theories* have a common logical part variously called *first-order logic* or *first-order predicate calculus*, or *elementary logic* or *quantification theory*. (I will use the simple name *predicate calculus*.)

[4] My treatment draws on [Tar76, Dav89].

Those symbols of the system that belong to the logical part are called the *logical* symbols of the system. The other symbols are called the *nonlogical* symbols of the formal system. The logical symbols generally include the *propositional connectives* "not" (a monadic operator, usually written as prefix $\neg$ or $\sim$ by mathematicians), "and" (a dyadic operator, often called *conjunction* and written as infix $\wedge$), "or" (dyadic, often called *disjunction* and written as infix $\vee$), "implies" (dyadic, written as infix $\supset$, $\rightarrow$, or $\Rightarrow$), and "if and only if" (dyadic, sometimes called *equivalence*, and written as infix *iff*, $\leftrightarrow$, $\Leftrightarrow$, or $\equiv$). More powerful logics include *function* symbols (such as f, g, and some that may be given a built-in interpretation such as $+$), predicate symbols (such as P, Q, and some that may be given a built-in interpretation such as $=$ and $<$), the *quantifiers* "for all" ($\forall$) and "there exists" ($\exists$), and/or *modalities* such as "necessarily" ($\Box$) and "possibly" ($\Diamond$).

The propositional connectives developed out of attempts to understand and codify principles of sound argument, and were intended to capture important constructions used in natural language. Prior to the development of modern formal logic, the usages of natural language were arbiters of what the connectives should mean, and there could be disagreement over this: for example, some medieval logicians treated "or" as inclusive ("either one or the other or both"), while others regarded it as exclusive ("either one or the other, but not both"). In formal logic, the meanings of the operators are determined by the rules of deduction (as explained shortly) and do not depend on intuitive notions.

Although the meanings of the logical operators do not *depend* on intuitive notions, they should clearly *correspond* to them—since logic is intended to support and strengthen our innate capabilities, not oppose them. Most of the logical operators do directly correspond to familiar notions, but the "implies" operator is a little less straightforward.

The symbol $\supset$ of logic is sometimes called *material* implication in order to distinguish it from the unadorned term "implication" used in ordinary discourse. The intent is that $A \supset B$ should capture conditional constructions of the form "*if* A *then* B." From this intuition, it follows that if A is a true statement, then $A \supset B$ should be considered true only if B is also a true statement. Thus

$$2 + 2 = 4 \supset \text{Paris is the Capital of France}$$

is a true sentence (because both the *antecedent* or *condition* "$2+2=4$" and *consequent* or *conclusion* "Paris is the Capital of France" are true). Observe that this example indicates one of the unintuitive aspects of material implication: there need be no "causal" connection between the antecedent and consequent. In natural language, we generally expect some connection or association between the clauses of a conditional, such as "*if* Paris is the seat of French Government *then* Paris is the Capital of France."

Leaving these doubts aside, and proceeding with our examination of material implication, we can see that

$$2 + 2 = 4 \supset \text{London is the Capital of France}$$

is a false sentence (because the antecedent is true but the consequent is false). But what if A, the antecedent is false? Should

$$2 + 2 = 5 \supset \text{Paris is the Capital of France}$$

be considered a true sentence? And how about

$$2 + 2 = 5 \supset \text{London is the Capital of France?}$$

In natural language one could make a plausible case for any of the alternatives (or even argue that a conditional need be neither true nor false when its antecedent is false), but in logic we adopt the convention that if A is false, then $A \supset B$ is true, whatever the value of B. One way to see that this is reasonable is to consider the alternatives. We know that when A is true, $A \supset B$ must be true exactly when B is true. To complete the definition, we need to assign truth values to $A \supset B$ when A is false, and there are just four possible ways to do this: (1) it could be false whatever the value of B, (2) it could be true exactly when B is true, (3) it could be true exactly when B is false, and (4) it could be true whatever the value of B. If we choose the first of these alternatives, then $A \supset B$ is exactly the same as $A \wedge B$ (i.e., "implies" is the same as "and"); if we choose the second, then $A \supset B$ is the same as B and the value of A is irrelevant; and if we choose the third, then $A \supset B$ turns out to be the same as $A \equiv B$. None of these corresponds to any reasonable interpretation of "implies" or "*if*... *then* ...," and so we are left with the fourth alternative.

Although this treatment of material implication works perfectly well, some logicians remain unhappy that it does not correspond exactly to informal usages and have sought different formulations, such as the notions of "strict" and "relevant" implication. The consideration of alternatives that we conducted above shows that any satisfactory formulation that is different to material implication will require modification of the basic logic, not just tinkering with the operators. Consequently, strict implication requires modal logic (which is described later), and relevant implication requires relevance logic [Dun85]. Material implication may

> seem a little strange at first, but it soon becomes familiar.[5] Alternative formalizations of conditional sentences are of philosophical and technical interest, but do not rival the practical utility and importance of the classical formulation in terms of material implication.

In addition to its language, a formal system specifies a selected (possibly infinite) set of sentences that are taken as given and called *axioms*, and a set of *rules of inference*, which are ways of deducing new sentences from given sets of sentences. Axioms are divided into logical and nonlogical axioms; rules of inference usually belong to the logical part of the system. The axioms and the rules of inference constitute the *deductive system* of the formal system considered.

Axioms are generally specified by *axiom schemes* (mathematicians often say *schema* and *schemata*), such as

$$\phi \vee \neg\phi,$$

where ϕ stands for an arbitrary sentence. This example (it is the *law of the excluded middle*) states that for any sentence ϕ, either ϕ or its negation must be true (there is no middle ground). When we put a particular sentence (e.g., "it will rain tomorrow") in place of ϕ, we obtain a specific axiom as an *instance* of the axiom scheme, such as

$$\text{it will rain tomorrow} \vee \neg \text{ it will rain tomorrow}.$$

Axiom schemes are convenient, because they allow an infinite number of axioms to be specified in a finite manner.

Similarly, rules of inference are often written in the form of schemes, exemplified by

$$\frac{\phi,\ \phi \supset \psi}{\psi},$$

where ϕ and ψ stand for arbitrary sentences. This example (it is the rule known as *modus ponens*) states that from a pair of sentences, one of the form ϕ and one of the form $\phi \supset \psi$, we may deduce the sentence ψ. We say that ϕ and $\phi \supset \psi$ are the *premises* or *antecedents* to this rule of inference, and ψ is its *conclusion* or *consequent*.

Notice that there are two mathematical languages in use here. One, the *object language*, is the formal system we are trying to define; the other is the *metalanguage* that we are using to do the definition. Here, the object language is a fragment of propositional calculus, and our metalanguage is the traditional language of informal

[5] Particularly once some important laws are learned, such as that $A \supset B \supset C$ is parsed as $A \supset (B \supset C)$, and is equivalent to $(A \wedge B) \supset C$. Some of these laws are listed in Section A.3.

mathematics expressed in English. It is necessary to keep the distinction between object language and metalanguage clearly in mind, especially when the same symbols are used in both; logicians use the terms *use* and *mention* to distinguish these cases. When we talk *about* some object, we are said to mention it; when we use an object to *refer* to some other object, we are said to use it. No confusion is likely with physical objects (we do not use the planet Mars to refer to something else), but it is possible with names or symbols. For example, from

> "Toby is a short haired cat" and
> "Toby is a name with four letters"

we might conclude that a certain name with four letters is a cat [Cur77]. In order to avoid this absurdity, we must recognize that "Toby" is used in quite different senses in these two sentences: in the first it is *used* to refer a certain cat, while in the second it is *mentioned* as a particular word. In order to reduce confusion, symbols are often enclosed in quotes when they are mentioned rather than used:

> "'Toby' is a name with four letters"

The variables ϕ and ψ used in the examples given earlier are *metalinguistic* variables, since they stand for arbitrary sentences of the object language. Axiom schemes and the operation of substituting sentences for metavariables are elements of the metalanguage; the resulting axioms are part of the object language. Similarly, it is necessary to distinguish proofs and theorems *about* the object language (i.e., metaproofs and metatheorems) from proofs and theorems *in* the object language. I am now ready to define the latter kind of proofs and theorems.

A *proof* of a sentence ψ from a set of sentences Γ is a finite sequence $\phi_1, \ldots, \phi_n$ of sentences with $\phi_n = \psi$, and in which each ϕ_i is either an axiom, or a member of Γ, or else follows from earlier ϕ_j's by one of the rules of inference. We say that ψ is provable from the *assumptions* Γ (or simply "Γ proves ψ") and write $\Gamma \vdash \psi$.[6] A *theorem* is a sentence that is provable without assumptions, and we write simply $\vdash \psi$.[7] The *theory* of a given formal system is the set of all its theorems.

A system is *consistent* if it contains no sentence ϕ such that both ϕ and its negation are theorems (i.e., if its theory does not contain both ϕ and $\neg\phi$). It is a metatheorem of all the logics considered here that *all* sentences are provable in an inconsistent system, and such systems are therefore useless.

[6] The symbol $\vdash$ is usually pronounced "turnstile."

[7] The axioms can be thought of as assumptions that come "for free." The reason we add (nonlogical) axioms to a formal system, rather than simply carry them as assumptions to all our theorems, is that there is generally an infinite number of axioms, and we need the metalinguistic notion of scheme to represent them conveniently. Of course, we could introduce a notion of "assumption scheme," but this would not be the way things have traditionally been done.

A system is *decidable* if there is an algorithm (i.e., a computer program that is guaranteed to terminate) that can determine whether or not any given sentence is a theorem. A system is semidecidable if there is an algorithm that can recognize the theorems (i.e., the algorithm is guaranteed to halt with the answer "yes" if given a theorem, but that need not halt if given a nontheorem—though if it does halt, it must give the answer "no" in this case). A system is *undecidable* if it is neither *decidable* nor *semidecidable*.

Notice that "proof" and "theorem" are purely syntactic notions: that is, they are marks on paper (or symbols in a computer). The formal system provides rules for transforming one set of marks into another; a proof is simply a sequence of formulas that conforms to the rules and ends up with a series of marks that we call the theorem. We do not have to know what any of these marks "mean" in order to tell whether or not a purported proof really is a correct proof of a given theorem; we simply have to check whether it follows the rules of the formal system concerned. Of course, the rules of formal systems are not chosen arbitrarily; they are chosen in the hope that if we interpret the marks on paper in some consistent way as statements about the real world, then the theorems of the system will be *true* statements about the world. The relationship between language and the world is the concern of *semantics*, and the goal of mathematical logic is to set things up so that the syntactic notion of *theorem* coincides with the semantic notion of *truth*. If this can be done, and if our axioms and assumptions correspond to true statements about (some aspect of) the real world, then the theorems that we can prove will also correspond to true statements about the world. In other words, we will be able to deduce (new) true facts about the world from given true facts simply by following the syntactic rules of our chosen formal system. We do not need to "understand" the world in order to carry out these syntactic operations; all our "understanding" (of the particular domain of interest) is encoded in the nonlogical axioms of the formal system used (and in the assumptions, if any, of the particular proofs performed). Since no "understanding" of the world is required to construct (or check) the purely syntactic objects that are proofs, these operations can, in principle, be performed by machine. Understanding of the world is needed only to choose the nonlogical axioms (and assumptions) and to interpret the utility of the theorems proved. Thus, the attraction of formal methods is that the necessary intuition and understanding of the world (i.e., the properties of the application domain concerned) are recorded *explicitly* in axioms and assumptions that can be subjected to intense scrutiny, and all the rest follows by deductions that can be mechanically checked.

The connection between syntax and semantics is established by an *interpretation* that identifies some real-world "domain" $\mathcal{D}$ and associates a true or false statement about $\mathcal{D}$ with each sentence. The statement associated with sentence ϕ is called its *valuation* in $\mathcal{D}$. Interpretations must satisfy certain structural properties that depend on the formal system concerned. (For example, if the formal system has function symbols, then the interpretation of $f(x, y)$ must be determined by the interpretations of f, x, and y.)

An interpretation is a *model* for a formal system if it valuates all its axioms to true; in addition, an interpretation is a model for a set of sentences T if, as well as being a model for the formal system concerned, it also valuates all the sentences in T to true. A set of sentences is *satisfiable* if it has a model. We say that a set of sentences T *entails* a sentence ψ and write $T \models \psi$ if every model of T is also a model of ψ—that is, if ψ valuates to true in every model of T. We say a sentence ψ is (universally) *valid* and write $\models \psi$ if it valuates to true in all models of its formal system. This connection between syntax and semantics, and the corresponding relationship between "provable" and "valid" were known and used informally (e.g., by Gödel and Skolem) in the 1920s; they were formalized by Tarski in the early 1930s.

A formal system is *sound* if $\Gamma \models \psi$ whenever $\Gamma \vdash \psi$; it is *complete* if $\Gamma \vdash \psi$ whenever $\Gamma \models \psi$. Roughly speaking, soundness ensures every provable fact is true, completeness ensures every true fact is provable. Notice that an inconsistent system cannot be sound. Obviously, only sound systems are of use in formal methods (or mathematics).[8] It would be nice if they were also complete (and even nicer if they were decidable), but I will observe later that most interesting formal systems are incomplete (and very few are decidable).

> In some formal systems, additional constraints are placed on the selection of models, and it is sometimes necessary to distinguish between soundness and completeness results relative to *all* models and those relative to certain *preferred* models. Other formal systems are constructed in the hope that they capture some aspect of reality exactly—that is, in the hope that they will have only one model, the *intended* or *standard* one. For example, the postulates of Euclid were intended to capture "ordinary" geometry exactly. Formal systems that have only one interpretation (up to isomorphism) are called *categorical*. In general, it is not possible to limit theories to just a single model; most interesting theories have *unintended* or *nonstandard* models in addition to the intended or standard ones (see section A.5.2).

Having established the general framework, I will now consider some specific formal systems that are of particular importance.

[8] It was a goal of the formalist school, led by Hilbert, to demonstrate (using only elementary methods) that the formal systems proposed as foundations for mathematics (such as set theory with the Axiom of Choice—see Section A.5.4) are sound. Gödel's results (see Section A.5.2) destroyed this plan. Soundness of the formal systems underlying mathematics cannot be demonstrated conclusively within those same systems, and must ultimately appeal to our experience and intuition.

A.3 Propositional Calculus

The simplest formal system generally considered is the *propositional calculus*, whose purpose is to provide a meaning for the propositional connectives listed earlier (Subsection A.2).[9] In its leanest form, the nonlogical part of the propositional calculus is empty, and its logical part considers only the connectives "implies" (written here as infix $\supset$) and "not" (written here as prefix $\neg$). The rest of its language consists of the parentheses, and some set of *proposition symbols*, which mathematicians generally write as P, Q, R, but which can be arbitrary strings such as "the cat is on the mat" that are considered atomic (i.e., without internal structure). The grammar of propositional calculus defines its sentences as follows:

- Any proposition symbol is a sentence, and
- If ϕ and ψ are sentences, then so are $(\neg\phi)$ and $(\phi \supset \psi)$.

(This is the mathematicians' description of propositional calculus, in which every compound sentence is fully parenthesized. Mathematicians drop parentheses—and so will I—when they are deemed unnecessary according to some informal rules. Computer scientists might give a context-free grammar for the language and make all these details explicit.)

There are three axiom schemes for propositional calculus:

A1. $\phi \supset (\psi \supset \phi)$,

A2. $(\phi \supset (\psi \supset \rho)) \supset ((\phi \supset \psi) \supset (\phi \supset \rho))$,

A3. $(\neg(\neg\phi)) \supset \phi$[10]

and one rule of inference

$$\frac{\phi,\ \phi \supset \psi}{\psi}$$

(modus ponens, often abbreviated MP).

The axioms of propositional calculus are infinite in number, since there are instances of each of the three schemes for all possible combinations of propositions. The symbols ϕ, ψ, and ρ appearing in the axiom schemes (and rule of inference) above are metalinguistic variables (or *metavariables*). An axiom *instance* is obtained from one of the axiom schemes by replacing its metalinguistic variables by propositions in a consistent fashion. I indicate the axiom instance of scheme 1 in which P replaces ϕ and Q replaces ψ by writing $A1[\phi \leftarrow P, \psi \leftarrow Q]$.

[9] Sources used for this section include Barwise [Bar78b] and De Long [DeL70].

[10] Sometimes $((\neg\phi) \supset (\neg\psi)) \supset (\psi \supset \phi)$ is used instead.

Using this machinery, we can prove the theorem $\vdash (P \supset P)$.

1.	$((P \supset ((P \supset P) \supset P)) \supset ((P \supset (P \supset P)) \supset (P \supset P)))$	
		$A2[\phi \leftarrow P, \psi \leftarrow (P \supset P), \rho \leftarrow P]$
2.	$(P \supset ((P \supset P) \supset P))$	$A1[\phi \leftarrow P, \psi \leftarrow (P \supset P)]$
3.	$((P \supset (P \supset P)) \supset (P \supset P))$	MP on 1 and 2
4.	$(P \supset (P \supset P))$	$A1[\phi \leftarrow P, \psi \leftarrow P]$
5.	$(P \supset P)$	MP on 3 and 4.

This derivation is completely unintuitive; the advantage of the "Hilbert-style" axiomatization I have been using is that it allows the interpretations of the propositional calculus to be defined in a concise manner. This is done in the next couple of pages. Following that, I introduce an alternative system that allows proofs in propositional calculus to be developed in an intuitive manner.

The domain of the interpretations of the propositional calculus is the set of truth values $\{\mathcal{T}, \mathcal{F}\}$ ($\mathcal{T}$ is an abbreviation for truth, $\mathcal{F}$ for falsehood). An interpretation of the propositional calculus assigns exactly one of these truth values (called its *valuation*) to every proposition symbol. Let $v(\phi)$ denote the valuation given to the proposition ϕ in a particular interpretation. Then we require that the valuation given to compound propositions in that interpretation is derived by recursive application of the following two *truth tables*:

ϕ	$\neg\phi$
$\mathcal{T}$	$\mathcal{F}$
$\mathcal{F}$	$\mathcal{T}$

ϕ	ψ	$\phi \supset \psi$
$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$
$\mathcal{T}$	$\mathcal{F}$	$\mathcal{F}$
$\mathcal{F}$	$\mathcal{T}$	$\mathcal{T}$
$\mathcal{F}$	$\mathcal{F}$	$\mathcal{T}$.

Sentences of the propositional calculus that valuate to $\mathcal{T}$ in all interpretations (i.e., the universally valid sentences) are called *tautologies*.

Interpretations constructed as described above provide models for the propositional calculus and it is possible to prove that the theorems and the tautologies coincide. Thus, the propositional calculus is sound (all theorems are tautologies) and complete (all tautologies are theorems). Furthermore, the propositional calculus is decidable—simply transfer into the semantic domain and use truth tables. Thus, another way to demonstrate $\vdash (P \supset P)$ is to construct the truth table for $(P \supset P)$ by substituting P for both ϕ and ψ in the general truth table for $\supset$ given above (note the second and third lines of the general table disappear, since they are impossible when ϕ and ψ have to take the same value)

P	$P \supset P$
$\mathcal{T}$	$\mathcal{T}$
$\mathcal{F}$	$\mathcal{T}$

and observe that its value is always $\mathcal{T}$. Hence, $\models (P \supset P)$, and so, by the completeness of propositional calculus, $\vdash (P \supset P)$.

The version of the propositional calculus I have given so far is rather impoverished. It is easy to add new connectives, either by treating them as abbreviations (i.e., macros), or by extending the formal system. Using macros, we would, for example, regard disjunction ($\phi \vee \psi$) as simply an abbreviation for $(\neg\phi) \supset \psi$ and would replace all instances of $\phi \vee \psi$ by their corresponding expansions in both the syntactic and semantic domains. Observe that this is a metalinguistic approach.

Using the alternative approach of expanding the formal system, we would add $\vee$ to the language as a logical symbol, and add the corresponding rules of inference

$$\frac{\phi \vee \psi}{(\neg\phi) \supset \psi} \qquad\qquad \frac{(\neg\phi) \supset \psi}{\phi \vee \psi}.$$

Similarly, the interpretation of $\vee$ would be supplied by the truth table

ϕ	ψ	$\phi \vee \psi$
$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$
$\mathcal{T}$	$\mathcal{F}$	$\mathcal{T}$
$\mathcal{F}$	$\mathcal{T}$	$\mathcal{T}$
$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$.

It is usually a matter of taste whether the properties of new connectives are added to the formal system by a rule of inference or by an axiom scheme. For example, if the equivalence ($\equiv$) connective has already been introduced, we can define $\vee$ by the axiom scheme

$$(\phi \vee \psi) \equiv ((\neg\phi) \supset \psi)$$

instead of the rule of inference given earlier. (Usually things are done in the opposite order, and $\phi \equiv \psi$ is viewed as an abbreviation for $(\phi \supset \psi) \wedge (\psi \supset \phi)$.)

There are a number of tautologies (often called "laws") that are worth learning by heart for use in systems that contain a generous collection of connectives:

		$\phi \vee \neg\phi$	law of the excluded middle
		$\neg(\phi \wedge \neg\phi)$	law of contradiction
$(\phi \vee \phi)$	$\equiv$	ϕ	laws of
$(\phi \wedge \phi)$	$\equiv$	ϕ	tautology
$\neg\neg\phi$	$\equiv$	ϕ	law of double negation
$\phi \supset \psi$	$\equiv$	$\neg\phi \vee \psi$	*verum sequitur ad quodlibet*
$\phi \supset \psi$	$\equiv$	$\neg\psi \supset \neg\phi$	law of contraposition
$\neg(\phi \vee \psi)$	$\equiv$	$\neg\phi \wedge \neg\psi$	De Morgan's
$\neg(\phi \wedge \psi)$	$\equiv$	$\neg\phi \vee \neg\psi$	laws
$\phi \vee (\psi \vee \mu)$	$\equiv$	$(\phi \vee \psi) \vee \mu$	Associative
$\phi \wedge (\psi \wedge \mu)$	$\equiv$	$(\phi \wedge \psi) \wedge \mu$	laws
$\phi \vee (\psi \wedge \mu)$	$\equiv$	$(\phi \vee \mu) \wedge (\phi \vee \mu)$	Distributive
$\phi \wedge (\psi \vee \mu)$	$\equiv$	$(\phi \wedge \mu) \vee (\phi \wedge \mu)$	laws
$\phi \supset (\psi \supset \mu)$	$\equiv$	$(\phi \wedge \psi) \supset \mu$	law of exportation
$(\phi \wedge \psi) \supset \mu$	$\equiv$	$(\phi \supset \mu) \vee (\psi \supset \mu)$	unnamed
$(\phi \vee \psi) \supset \mu$	$\equiv$	$(\phi \supset \mu) \wedge (\psi \supset \mu)$	laws

Notice that most of these laws come in pairs in which one member can be obtained from the other by interchanging $\wedge$ and $\vee$. Such pairs are called *duals*.

It is also worth remembering the derived inference rule

$$\frac{\neg\psi,\ \phi \supset \psi}{\neg\phi}$$

known as *modus tollens*. It follows from modus ponens by the law of contraposition.

Conventionally, the equivalence $\phi \equiv \psi$ is regarded as an abbreviation for a pair of "simpler" expressions, such as $(\phi \supset \psi) \wedge (\psi \supset \phi)$ or $(\neg\phi \wedge \neg\psi) \vee (\phi \wedge \psi)$ and it is often expanded into one of these forms whenever it is encountered during a proof. However, equivalence satisfies some beautiful laws, such as the *Golden Rule*:

$$(\phi \wedge \psi) \equiv \phi \equiv \psi \equiv (\phi \vee \psi)$$

(one of whose beautiful properties is that it does not matter how we associate—i.e., parenthesize—the equivalences). The book by Dijkstra and Scholten [DS90] make extensive and effective use of laws for equivalence.

The propositional connectives introduced so far are not the only ones. Those called **nand** and **nor** are defined by

$$\begin{aligned} \phi \ \mathbf{nand}\ \psi &\equiv \neg(\phi \wedge \psi) \\ \phi \ \mathbf{nor}\ \psi &\equiv \neg(\phi \vee \psi) \end{aligned}$$

and are often used in digital circuits. All the other propositional connectives can be defined in terms of either one of these two—a property first discovered by Pierce but often attributed to Sheffer (who wrote **nor** in the form $\phi \mid \psi$, where $\mid$ is known as "Sheffer's stroke").

Computer scientists generally like to have the three-way *if... then... else* connective available. It can be defined by

$$\textbf{if } \phi \textbf{ then } \psi \textbf{ else } \mu \equiv (\phi \supset \psi) \wedge (\neg\phi \supset \mu).^{11}$$

Finally, note that the constants *true* and *false* can be introduced by

$$true \equiv (\phi \vee \neg\phi),$$

$$false \equiv (\phi \wedge \neg\phi).$$

A.3.1 Propositional Sequent Calculus

The axiom schemes and rule of inference I have given for propositional calculus make for rather tedious proofs and are "barbarously unintuitive" [Hod83]. Usually, it is simpler to exploit the soundness and completeness of the propositional calculus and transfer into the semantic domain where theorems can be established using truth tables.[12] A case against doing this is that a proof should represent an *argument* for the truth a theorem and should lead to improved understanding (not least when the putative theorem is actually false and the proof attempt fails), whereas the truth-table method merely says "true" or "false." On the other hand, the propositional calculus is so limited, and its theorems so straightforward, that the opportunity

[11] In richer logics, *if... then... else* is often extended to a *polymorphic* form that denotes a value of a different sort (or type—these notions are explained in Sections A.9 and A.10) than a proposition. For example, in

$$|x| = \textbf{if } x < 0 \textbf{ then } -x \textbf{ else } x$$

the *if... then... else* denotes a numerical value. These polymorphic forms can be converted to the propositional kind by "lifting" the *if... then... else* over adjacent relations:

$$\textbf{if } x < 0 \textbf{ then } |x| = -x \textbf{ else } |x| = x.$$

[12] Large propositional expressions can arise in the specification and analysis of hardware circuits. Methods based on Boolean (or Binary) Decision Diagrams (BDDs) [Ake78, Bry86, Bry92] are generally the most efficient in practice by a considerable margin, and should be a basic component of any tool for formal methods that is intended to address such problems. Although BDDs are usually superior, methods derived from those of Davis and Putnam [DP60] can outperform BDDs on some problems [US93].

to gain insight from proofs is distinctly limited.[13] In richer systems, however, the ability to construct reasonably natural proofs in a reasonably effective manner will clearly be desirable, so I use the propositional calculus to illustrate one approach to doing this.

First, note that proofs can be simplified by using previously-proved theorems (which effectively enriches the set of sentences that can be used as axioms), and by deriving additional rules of inference. For example, there is an important metatheorem about propositional calculus called the *deduction theorem*, which I state as follows:

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash (\phi \supset \psi)}$$

where Γ is any set of sentences and Γ, ϕ is interpreted as the union $\Gamma \cup \{\phi\}$. Notice that I have stated the Deduction Theorem as an extended rule of inference: formerly rules of inference were relations on sets of sentences; now I allow them to be relations on proofs as well. This proves extremely convenient.

The deduction theorem is a *metatheorem* because it is a theorem *about* propositional calculus, not a theorem *in* propositional calculus. In order to prove the deduction theorem, we can either use informal mathematical arguments based on the informal definitions of proof and so on given earlier, or we can formalize propositional calculus within some more powerful formal system and do the proof within that system. In either case, the proof is by induction on the length of the derivation of ψ from S, ϕ.

Using the deduction theorem, the proof of $\vdash (P \supset P)$ becomes trivial:

1.	$P \vdash P$	Definition of proof
2.	$\vdash (P \supset P)$	Deduction Theorem$[S \leftarrow \emptyset, \phi \leftarrow P, \psi \leftarrow P]$.

But even with additions such as these, it seems clear that it will not be easy or natural to develop formal proofs using the formulation of propositional calculus that we have seen so far. This is not surprising, for that formulation is a "Hilbert-style" calculus designed for metamathematical convenience—that is, it is chosen to allow short statements of notions such as "interpretation," and relatively simple proofs of soundness and completeness. It is not, however, designed for actually *doing* proofs. For that purpose, alternative formulations due to Gentzen are much more convenient.

Gentzen's *natural deduction* and *sequent calculus* formulations use fewer axioms and more rules of inference than Hilbert-style formulations, and allow more "natural" proofs.[14] Of Gentzen's two formulations, natural deduction is quite convenient

[13] This is not to say that errors in using propositional calculus are not common in formal methods, only that the line by line reasoning of a formal proof is seldom needed to detect or explain them.

[14] Sources used for this section include Barwise [Bar78b], Gallier [Gal86], and Sundholm [Sun83].

on the (hand)written page or whiteboard, but the sequent calculus is much to be preferred for computer-assisted activity since all the information relevant to the local part of a proof is gathered together in a convenient way. A *sequent* is written the form $\Gamma \rightarrow \Delta$[15] where Γ and Δ are lists of sentences called the *antecedents* and *succedents*, respectively. The intuitive interpretation is that the conjunction of the antecedents should imply the disjunction of the succedents. Equivalently, we want to show that one (or more) of the succedents is true, or that one (or more) of the antecedents is false.

A proof in sequent calculus is a tree of sequents, whose root (at the bottom) is a sequent of the form $\rightarrow \phi$ (i.e., with empty antecedent), where ϕ is the sentence to be proved. The proof tree is "grown" upwards from the root by applying *inference rules* of the form

$$\frac{\Gamma_1 \rightarrow \Delta_1 \quad \cdots \quad \Gamma_n \rightarrow \Delta_n}{\Gamma \rightarrow \Delta}\ \mathbf{N}.$$

This says that the rule named N takes a leaf node of a proof tree of the form $\Gamma \rightarrow \Delta$, and adds n new leaves of the form given (n may be zero, in which case that branch of the proof tree terminates). The goal is to apply rules of inference to construct a proof tree whose branches are all terminated. This is sometimes called "backwards" proof, since we start with the conclusion to be proved. (Natural deduction, on the other hand, is an example of a "forwards" approach to proof in which we start with assumptions and work towards the conclusion.) Although the proof is *constructed* "backwards," it can subsequently be *read* "forwards."

I now give the inference rules for the propositional sequent calculus with the operators $\wedge, \vee, \supset$, and $\neg$ (omitting the "structural rules," which simply say that the order of sentences within an antecedent or succedent is unimportant, and that sentences can always be deleted from the antecedent or succedent). In the following, upper case greek letters Γ and Δ represent lists of sentences, and lower case letters ϕ, and ψ represent individual sentences. I will often write sequents in a form such as $\Gamma, \phi \supset \psi \rightarrow \Delta$, meaning that the antecedents consist of some (possibly empty) list Γ and a sentence of the form $\phi \supset \psi$.

The *Propositional Axiom* rule is the one that terminates a branch of the proof tree. It applies when the same sentence appears in both the antecedent and succedent.

$$\frac{}{\Gamma,\ \phi\ \rightarrow\ \phi,\ \Delta}\ \mathbf{Ax.}$$

This rule can be seen to be plausible by thinking of the sequent as expressing the tautology $(\Gamma \wedge \phi) \supset (\phi \vee \Delta)$ where the lists Γ and Δ are interpreted as the

[15] $\Gamma \vdash \Delta$ is also used, but this may be confusing here.

conjunction and disjunction, respectively, of their elements. Equivalently, under the interpretation that we need to show that one (or more) of the succedents is true, or that one (or more) of the antecedents is false, the propositional axiom rule can be justified by noting that ϕ appears as both an antecedent and a succedent and must be either true or false.

There are two rules for each of the propositional connectives, corresponding to the antecedent and succedent occurrences of these connectives. The negation rules simply state that the appearance of a sentence in the antecedent is equivalent to the appearance of its negated form in the succedent, and vice-versa.

$$\frac{\Gamma \rightarrow \phi, \Delta}{\Gamma,\ \neg\phi \rightarrow \Delta}\ \neg\rightarrow \qquad\qquad \frac{\Gamma, \phi \rightarrow \Delta}{\Gamma \rightarrow \neg\phi,\ \Delta}\ \rightarrow\neg.$$

The inference rules $\neg \rightarrow$ and $\rightarrow \neg$ are often called the rules for "negation on the left" and "negation on the right," respectively. Similar naming conventions are used for the other rules. One way to see that the rule $\neg \rightarrow$, say, is reasonable is to consider the sequent above the line as $\Gamma \supset \phi \vee \Delta$, which is equivalent to $\neg\Gamma \vee (\phi \vee \Delta)$, which is equivalent to $(\neg\Gamma \vee \phi) \vee \Delta$, which is equivalent to $\neg(\neg\Gamma \vee \phi) \supset \Delta$, which is equivalent to $(\Gamma \wedge \neg\phi) \supset \Delta$, which is an interpretation of the sequent below the line.[16] Equivalently, under the interpretation that we need to show that one (or more) of the succedents is true, or that one (or more) of the antecedents is false, the negation rules can be seen as simply trading demonstration of the truth of ϕ for that of the falsity of $\neg\phi$, and vice-versa.

The rule for conjunction on the left is a consequence of the fact that the antecedents of a sequent are implicitly conjoined; the rule for conjunction on the right causes the proof tree to divide into two branches, in which we separately prove each of the two conjuncts.

$$\frac{\phi,\ \psi,\ \Gamma \rightarrow \Delta}{\phi \wedge \psi,\ \Gamma \rightarrow \Delta}\ \wedge\rightarrow \qquad\qquad \frac{\Gamma \rightarrow \phi,\ \Delta \qquad \Gamma \rightarrow \psi,\ \Delta}{\Gamma \rightarrow \phi \wedge \psi,\ \Delta}\ \rightarrow\wedge.$$

The rules for disjunction are "duals" of those for conjunction.

$$\frac{\phi,\ \Gamma \rightarrow \Delta \qquad \psi,\ \Gamma \rightarrow \Delta}{\phi \vee \psi,\ \Gamma \rightarrow \Delta}\ \vee\rightarrow \qquad\qquad \frac{\Gamma \rightarrow \phi,\ \psi,\ \Delta}{\Gamma \rightarrow \phi \vee \psi,\ \Delta}\ \rightarrow\vee.$$

The rule for implication on the right is a consequence of the implication "built in" to the interpretation of a sequent; the rule for implication on the left splits the proof into two branches: on one we must prove the antecedent to the implication, and on the other we may assume the consequent.

[16] In this chain of equivalences, I used the identities $(\phi \supset \psi) \equiv (\neg\phi \vee \psi)$, and one of De Morgan's Laws.

$$\frac{\psi,\ \Gamma \rightarrow \Delta \qquad \Gamma \rightarrow \phi,\ \Delta}{\phi \supset \psi,\ \Gamma \rightarrow \Delta} \supset\rightarrow \qquad \frac{\Gamma,\ \phi \rightarrow \psi,\ \Delta}{\Gamma \rightarrow \phi \supset \psi,\ \Delta} \rightarrow\supset.$$

Notice how the rules of the sequent calculus have an intuitively plausible character, and entirely symmetric construction. This makes proof construction relatively systematic and natural. To see how it works, consider a proof of the sentence

$$(P \supset Q \supset R) \supset (P \wedge Q \supset R).$$

(Note, the implies symbol $\supset$ associates to the right and binds less tightly than $\wedge$, so this sentence is actually an instance of the law of exportation.)

First we construct the goal sequent

$$\rightarrow (P \supset Q \supset R) \supset (P \wedge Q \supset R).$$

Then we must seek an inference rule whose part "below the line" has an instance equal to this sequent. There is only one choice in this case, since the sole top level operator is an implication on the right. Hence, we apply the rule for implication on the right (with $[\phi \leftarrow (P \supset Q \supset R), \psi \leftarrow (P \wedge Q \leftarrow R)]$ and Γ and Δ empty):

$$\frac{(P \supset Q \supset R) \rightarrow (P \wedge Q \supset R)}{\rightarrow (P \supset Q \supset R) \supset (P \wedge Q \supset R)} \rightarrow\supset.$$

Now we focus our attention on the sequent above the line:

$$(P \supset Q \supset R) \rightarrow (P \wedge Q \supset R)$$

and look for a rule whose part below the line can be made to match this sequent. There are two choices here: implication on the left or implication on the right. The former rule will cause the proof tree to branch and it is usually best to put this off as long as possible. So we again choose implication on the right (this time with $[\Gamma \leftarrow (P \supset Q \supset R), \phi \leftarrow P \wedge Q, \psi \leftarrow R]$ and Δ empty):

$$\frac{(P \supset Q \supset R), (P \wedge Q) \rightarrow R}{(P \supset Q \supset R) \rightarrow (P \wedge Q \supset R)} \rightarrow\supset.$$

Again we focus on the sequent above the line

$$(P \supset Q \supset R), (P \wedge Q) \rightarrow R$$

and seek an applicable rule. There are two candidates: implication on the left, or conjunction on the left. The former rule will cause the proof tree to branch, so we use conjunction on the left:

$$\frac{(P \supset Q \supset R), P, Q \rightarrow R}{(P \supset Q \supset R), (P \wedge Q) \rightarrow R} \wedge\rightarrow$$

Now the sequent above the line is

$$(P \supset Q \supset R), P, Q \to R$$

and we have no choice but to use the rule for implication on the left:

$$\frac{(Q \supset R), P, Q \to R \qquad P, Q \to P, R}{(P \supset Q \supset R), P, Q \to R} \supset\to$$

The right-hand branch can be closed immediately:

$$\frac{}{P, Q \to P, R}\ \mathbf{Ax.}$$

The left-hand branch requires another application of the rule for implication on the left:

$$\frac{R, P, Q \to R \qquad P, Q \to Q, R}{(Q \supset R), P, Q \to R} \supset\to.$$

Its left and right branches can then be closed:

$$\frac{}{R, P, Q \to R}\ \mathbf{Ax}$$

$$\frac{}{P, Q \to Q, R}\ \mathbf{Ax}$$

Since all the branches are closed, the theorem is proved.

We can collect all the steps together into the following representation of the proof tree:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{R, P, Q \to R}\,\mathbf{Ax} \qquad \cfrac{}{P, Q \to Q, R}\,\mathbf{Ax}}{(Q \supset R), P, Q \to R}\supset\to \qquad \cfrac{}{P, Q \to P, R}\,\mathbf{Ax}}{(P \supset Q \supset R), P, Q \to R}\supset\to.}{(P \supset Q \supset R) \to (P \wedge Q \supset R)}\to\supset}{\to (P \supset Q \supset R) \supset (P \wedge Q \supset R)}\to\supset$$

Notice how we were able to develop this proof in an entirely "logical" and orderly way. The reader might want to compare this proof with one using the Hilbert-style formulation of the propositional calculus introduced at the beginning of Section A.3.

A.4 Predicate Calculus

Propositional calculus is a very limited formal system and allows us to express only very simple ideas. For example [Dav89, page 43], we cannot express the idea "if x is even, then $x + 1$ is odd" in propositional calculus. To see this, suppose we try and split the idea up as two propositions: $\boxed{x}_e$ will mean "x is even" and $\boxed{x+1}_o$ will mean "$x + 1$ is odd." Then it might seem that we could express the intended idea with the compound proposition $\boxed{x}_e \supset \boxed{x+1}_o$. But this does not work: $\boxed{x}_e$ and $\boxed{x+1}_o$ are independent propositions without internal structure (which is why I have written them in boxes), whereas the sense we want to convey requires that both refer to the same x. The *predicate calculus* (also called *first-order logic* or *elementary logic* or *quantification theory*) is an extension to propositional calculus that gives us *individual variables* and the expressive power we need to make this and very many other statements concerning mathematics and the world.

The formal system of the basic predicate calculus has no nonlogical part. Its logical symbols include the propositional connectives and the *quantifiers* "for all" (also called the *universal* quantifier and written $\forall$) and "there exists" (also called the *existential* quantifier and written $\exists$). The other symbols include *constants* (generally written as lowercase letters near the front of the alphabet: $a, b, \ldots$), *variables* (generally written as lowercase letters near the end of the alphabet: $\ldots, y, z$), *function symbols* (generally written as the lowercase letters: $f, g, \ldots$), and *predicate symbols* (generally written as uppercase letters: $P, Q, \ldots$).

The grammar of predicate calculus defines its language as follows:

- A *term* is a constant symbol, a variable symbol, or a *function application* $f(\sigma, \tau, \ldots, \nu)$ where f is a function symbol and $\sigma, \tau, \ldots, \tau$ are terms. (Actually, the symbols $\sigma, \tau, \ldots, \tau$ are metavariables *representing* terms). For convenience, I often write function applications using infix notation, such as $2 + 3$, rather than $+(2, 3)$.

- An *atomic formula* has the form $P(\sigma, \tau, \ldots, \nu)$ where P is a predicate symbol and $\sigma, \tau, \ldots, \nu$ are terms. (As before, $\sigma, \tau, \ldots, \tau$ are actually metavariables *representing* terms). Computer scientists may find it more familiar to think of predicates as functions that return truth values (Russell and other early writers actually called them "propositional functions"). For convenience, I often write atomic formulas using infix notation, such as $2 < 3$, rather than $< (2, 3)$.

  There are obvious consistency constraints on the sentences we should write: if f appears somewhere applied to two arguments (e.g., in a term of the form

$f(x, y)$) we should not use it somewhere else applied to fewer or greater numbers of arguments. The same applies to predicates.[17]

- A *well-formed formula* (wff) is an atomic formula, or a series of wffs combined using the propositional connectives, or an expression of the form $(\forall x : \phi)$ or $(\exists x : \phi)$ where x is a variable symbol and ϕ is a metavariable representing a wff.

  The *scope* of a quantifier is the wff to which it applies (i.e., the one following the colon). For example, in

  $$(\forall x : even(x) \supset odd(x+1)) \tag{A.1}$$

  the scope of the $\forall$ is the expression $even(x) \supset odd(x+1)$. A variable is *bound* if it appears in an expression in the scope of a quantifier naming that variable; otherwise it is *free*. A wff is *open* if it contains free variables, otherwise it is *closed*.

- The *sentences* of the predicate calculus are the closed wffs. By convention, open wffs are usually interpreted as sentences by automatically providing outer levels of universal quantification to bind any free variables (this is called taking the *universal closure*). Thus the open wff $(\exists y : y = x + 1)$ is closed by interpreting it as $(\forall x : (\exists y : y = x + 1))$.[18]

Numerous syntactic shorthands are generally applied to make predicate calculus sentences more readable. For example, in the examples above, I used *even* and *odd* as predicate symbols, and used infix $=$ to denote a predicate symbol, infix $+$ to denote a function symbol, and 1 to denote a literal constant. In the most primitive predicate calculus notation, we might have to write (A.1) as

$$(\forall x : E(x) \supset O(s(x)))$$

where s is intended to indicate the successor function on the integers, and E and O are meant to indicate the "even" and "odd" predicates, respectively. No matter how they are written, this example assumes that numbers, together with certain functions (e.g., $+$ or s) and predicates (e.g., *odd* or O) on them, are part of the formal system concerned. In fact, these concepts are *not* part of the predicate calculus, but must be axiomatized or defined within it. I examine how this can

[17] To be really thorough, we should distinguish different sets of n-ary function and predicate symbols, for each n.

[18] Notice the difference between $(\forall x : (\exists y : y = x + 1))$ and $(\exists x : (\forall y : y = x + 1))$. The former says that for each x, there is some y such that $y = x + 1$—this is obviously valid. The latter says that there is an x such that, for every y, $y = x + 1$, i.e., x is one less than every y—which is obviously unsatisfiable (in the usual interpretation of arithmetic).

be done in later sections; first I examine deduction and interpretation in the basic predicate calculus.

In a Hilbert-style formulation, the axiom schemes and rules of inference for predicate calculus are those of propositional calculus (whose metavariables are now to be replaced by formulas of predicate calculus, rather than by proposition symbols, in order to construct axioms and rule instances), together with the two axiom and two rule schemes shown below. In these schemes, the notation $\phi(x)$ means a formula ϕ in which x is a free variable, and $\phi[x \leftarrow t]$ means the substitution instance of ϕ with all free occurrences of x replaced by the term symbolized by t.[19]

The following axiom schemes are added to those of propositional calculus given on page 230:

A4. $(\forall x : \phi(x)) \supset \phi[t \leftarrow x]$, provided no free occurrences of x in ϕ lie in the scope of any quantifier for a free variable appearing in the term t (we say that t is free for x in ϕ).[20] This axiom scheme simply says that if some formula ϕ is true for all x, then it is certainly true when some particular term t is substituted for x in ϕ (provided no free variables get "captured").

A5. $\phi[x \leftarrow t] \supset (\exists x : \phi(x))$, provided t is free for x in ϕ. This axiom scheme says that we can conclude that there exists some x satisfying the formula ϕ if some substitution instance of ϕ is true.[21]

The following rule schemes of *generalization* are added to those of propositional calculus:

$$\frac{\psi \supset \phi(v)}{\psi \supset (\forall x : \phi(x))},$$

and

$$\frac{\phi(v) \supset \psi}{(\exists x : \phi(x)) \supset \psi}$$

where the variable v is not free in ψ. The rule of universal generalization can best be understood by considering the simpler case where ψ is true. Then the rule becomes

$$\frac{\phi(v)}{(\forall x : \phi(x))}$$

[19]Mathematicians usually indicate a substitution instance by $\phi[t/x]$ rather than $\phi[x \leftarrow t]$. I prefer my notation because the arrow makes it easy to remember the direction of the substitution. Substitution is an operation in the object language (it transforms one formula into another). I have also been using the notation $A1[\phi \leftarrow P]$ to indicate the metalevel operation in which an axiom or rule instance is constructed by letting ϕ be P in the scheme $A1$. This ambiguous notation is nonstandard, but I find it convenient.

[20]This caveat is called the "occurs check." To see why it is necessary, let ϕ be $(\exists y : x + 1 = y)$ and let t be y. Prolog interpreters ignore the occurs check in the interest of speed, and are unsound for this reason among others [NS93, Chapter III].

[21]Some formulations include only the universal quantifier; in these cases, the existential quantifier can be introduced by the definition $(\exists x : \phi(x)) \equiv \neg(\forall x : \neg\phi(x))$.

which says that if ϕ is true for some arbitrary v, then it must be true for all x. Existential generalization can be derived from universal generalization by the transformation mentioned in the previous footnote.

The models of predicate calculus are constructed as follows.[22] Given a nonempty set $\mathcal{D}$ as the domain of interpretation, an interpretation associates an n-place total function on $\mathcal{D}$ with each n-place function symbol, an n-place relation with each n-place predicate symbol, and an element of $\mathcal{D}$ to each constant symbol.[23] The interpretation of $P(t_1, t_2, \ldots, t_n)$ for predicate symbol P is true if and only if the tuple $(v(t_1), v(t_2), \ldots v(t_n))$ is in the set $v(P)$ where $v(P)$ is the relation interpreting P, and $v(t_i)$ is the interpretation of the term t_i. The interpretation of $f(t_1, t_2, \ldots, t_n)$ for function symbol f is the value $v(f)(v(t_1), v(t_2), \ldots v(t_n))$ where $v(f)$ is the function interpreting f, and $v(t_i)$ is the interpretation of the term t_i. The propositional connectives have their usual (truth-table) meaning; $(\forall x : \phi(x))$ is interpreted to be true if and only if the interpretation of ϕ is true for all values of the variable x in the domain $\mathcal{D}$, and $(\exists x : \phi(x))$ is interpreted to be true if and only if the interpretation of ϕ is true for some value of the variable x taken from the domain $\mathcal{D}$. (If ϕ does not contain the variable x, the interpretations of $(\forall x : \phi(x))$ and $(\exists x : \phi(x))$ are the same as the interpretation for ϕ.)

Like propositional calculus, predicate calculus is sound and complete (soundness is relatively straightforward; completeness was established by Gödel in his Ph.D. thesis in 1930[24]). That is, the theorems (i.e., the sentences that can be deduced using the axioms and rules of inference given above) coincide with the universally valid sentences (i.e., those that valuate to true in all interpretations).

We have now covered enough material to formalize and interpret the example that began this appendix:

> That plane is a Boeing 737;
> All Boeing planes, except the 747, have two engines;
> Therefore that plane has two engines.

[22] The definitions that follow use the terminology of set theory. An n-place *relation* on the set $\mathcal{D}$ is any set of n-tuples on $\mathcal{D}$, that is any set of objects of the form $(a_1, a_2, \ldots, a_n)$, where each a_i is a member of $\mathcal{D}$. An n-place *function* f on the set $\mathcal{D}$ can be considered as $n+1$-place relation with the property that if $(a_1, a_2, \ldots, a_n, x)$ and $(a_1, a_2, \ldots, a_n, y)$ are both members of f, then $x = y$ (i.e., the last member of the tuple is uniquely determined by the first n). In this case, the *application* of f to the *arguments* $(a_1, a_2, \ldots, a_n)$ is written $f(a_1, a_2, \ldots, a_n)$ and yields the value x. A function is *total* if it has a value for every combination of arguments.

[23] I am simplifying a bit here. Interpretations are correctly defined relative to a sequence of values from $\mathcal{D}$ that supplies arbitrary, but fixed, interpretations to free variables.

[24] Skolem had essentially demonstrated consistency some years earlier, but did not realize it [Hod83].

We can formalize this in first-order predicate calculus

$$\frac{\begin{array}{l}\texttt{is\_a\_Boeing\_plane}(p) \wedge \texttt{is\_a\_737}(p) \\ \forall x : \texttt{is\_a\_Boeing\_plane}(x) \wedge \neg\texttt{is\_a\_747}(x) \supset \texttt{has\_two\_engines}(x)\end{array}}{\texttt{has\_two\_engines}(p)}$$

where is_a_Boeing_plane, is_a_737, is_a_747, and has_two_engines are predicates, p is a constant, and x is a variable. A little thought shows that this deduction is not valid: we also need another premise to state that 737s and 747s are different. The following is adequate for our purposes:

$$\forall x\colon \texttt{is\_a\_737}(x) \supset \neg\texttt{is\_a\_747}(x).$$

Then, applying the logical axioms and rules of inference given earlier (use A4 to instantiate the two universally quantified premises with $[x \leftarrow p]$, then use propositional reasoning), we can prove the conclusion from the three premises. Since predicate calculus is sound, this tells us that the conclusion will be true in any interpretation that valuates the premises to true. Notice, however, that if we substitute is_a_727 for is_a_737, we would still have a valid proof, but a conclusion that is false when the domain $\mathcal{D}$ of interpretation is "airplanes in current service" and the predicates have their intuitive meaning. How can this be? The explanation is that the second premise

$$\forall x : \texttt{is\_a\_Boeing\_plane}(x) \wedge \neg\texttt{is\_a\_747}(x) \supset \texttt{has\_two\_engines}(x)$$

is false in this interpretation. But if that is so, then our original conclusion (about the 737) is also suspect in this interpretation. And indeed it is: although the conclusion happens to be true in this interpretation, and although the line of reasoning is valid (so that the conclusion must be true in any interpretation that validates the premises), one of the premises is false in the intended interpretation. This example illustrates a central issue in formal methods: because we intend to make use of any conclusion we prove, we must have a particular interpretation in mind and we need to pay great attention to validating our premises in that interpretation.

A.4.1 Skolemization and Resolution

Unlike propositional calculus, predicate calculus is only semidecidable. Because it introduces some concepts and terminology that are useful in understanding certain theorem proving strategies, I will sketch one of the ways to demonstrate semidecidability of predicate calculus. Recall that semidecidable means there must be some computer program that is guaranteed to terminate with the answer "yes" when given a theorem; it is not required to terminate when given a nontheorem (but if it does, it must give the answer "no"). Because the predicate calculus is sound and complete, the desired semidecision procedure could either be proof theoretic (i.e., it could look for a proof of the given sentence), or model theoretic (i.e., it could attempt to show that the sentence is universally valid). The approach I will follow is model theoretic.

Suppose we have to establish a sentence of the form $(\forall x : \phi(x))$; to make things concrete, consider $(\forall x : x < x + 1)$. If this sentence is valid, then any particular substitution instance, say $17 < 17{+}1$, must be true. So one way to prove the sentence might be to choose some substitution instance such as this "at random" and check that one instance. The problem, of course, is that a particular, interpreted constant such as 17 might have some special properties, and we might just happen to pick one that satisfied the sentence, whereas another might not (e.g., $(\forall x{:}\, x \times x = x)$ might appear valid if it were enough to exhibit $[x \leftarrow 1]$). But if instead of a particular numeral, we choose a new symbolic constant, say c, that appears nowhere else and could still establish $c < c{+}1$, then we would surely have done enough to establish the original sentence. Reverting to the general case, the idea is to establish $(\forall x : \phi(x))$ by establishing $\phi(c)$ for some new constant c. Notice that $\phi(c)$ has no quantifiers nor variables: it is a *propositional* sentence and therefore, by the decidability of the propositional calculus, we can decide its validity. We seem to have reduced the decision problem for universally quantified formulas of predicate calculus to that of propositional calculus.

Now consider a slightly more complicated sentence: $(\forall x : (\exists y : y = x + 1))$. To establish this, we can start by substituting some arbitrary constant, say c, for the universally quantified variable x to yield the simpler sentence $(\exists y : y = c{+}1)$. Then, to establish this existential sentence it is enough to find some constant term that can be substituted for y in order to make $y = c + 1$ true (obviously, $[y \leftarrow c + 1]$ does the job here). The quantifier-free formula $y = c + 1$ is called a *Skolem form* of the original formula $(\forall x : (\exists y : y = x + 1))$. A substitution instance of the Skolem form, in which all its free variables are replaced by terms involving only constant symbols, is called a *ground instance* of the original formula. The idea behind the process I am developing is that the original first-order sentence must be valid if some ground instance is propositionally valid (i.e., a tautology). Now the construction of the Skolem form is a mechanical process (I will explain it shortly), and propositional validity is decidable—so to test the validity of a first-order formula, all we have to do is generate the Skolem form, and then search for a ground instance that is a tautology.[25] If the original formula is valid, then eventually our search will terminate, but if it is invalid, then our search may go on forever, generating more and more complicated ground instances without ever finding one that is a tautology.

[25]This is not quite accurate: as explained shortly, we may need a disjunction of ground instances to obtain a tautology.

I have just given a crude sketch of the semidecidability of first-order predicate calculus.[26] This idea can be exploited in mechanical "theorem provers" (I use quotes because this is a model-, not proof-theoretic approach)—although it is usual to employ heuristics, or user-supplied hints, to generate ground instances rather than exhaustive search.[27] In order to use this approach, it is necessary to understand Skolemization (as the process of constructing the Skolem form is called) in a little more detail.

My simple example might have made it appear that to Skolemize a first-order sentence, all we have to do is replace the universally quantified variables with (different) arbitrary constants, leave the existentially quantified variables alone, and then remove all the quantifiers. If we apply this naive scheme to the (obviously false) sentence $(\exists x\colon \phi(x)) \supset (\forall y\colon \phi(y))$ we would obtain $\phi(x) \supset \phi(c)$, which has a valid ground instance under the substitution $[x \leftarrow c]$. We have just "proved" a false theorem, so something must have gone wrong.

Our mistake was that we did not take care of the "implicit" negation in the antecedent to an implication. Since $a \supset b$ is the same as $\neg a \vee b$, the original sentence could have been written as $\neg(\exists x\colon \phi(x)) \vee (\forall y\colon \phi(y))$, and this is equivalent to $(\forall x\colon \neg\phi(x)) \vee (\forall y\colon \phi(y))$. Our Skolemization procedure (correctly) transforms this to $\neg\phi(d) \vee \phi(c)$, which is not a tautology. The problem, then, is that existential quantifiers in the antecedent to an implication are "essentially" universal (and vice-versa). We see that Skolemization has to replace the *essentially universally* quantified variables with different arbitrary constants, and to leave the *essentially existential* ones alone. To determine the essential "parity" of a quantifier, simply count the number of negations in whose scope it occurs, noting the implicit negation in the antecedent of an implication (and expanding equivalences of the form $\phi \equiv \psi$ as two implications: $\phi \supset \psi \wedge \psi \supset \phi$); if a quantifier appears within an odd number of negations, then its essential parity is the opposite of its appearance (i.e., an existential is essentially universal, and vice-versa).

Even with this adjustment, our Skolemization process is still flawed. Consider the example we looked at earlier, but with the order of the quantifiers reversed: $(\exists y : (\forall x : y = x + 1))$. This sentence is unsatisfiable (it says there is some y which is equal to $x + 1$ for every x, whereas the original, valid, sentence said that for every x, there is some y that is equal to $x + 1$). Yet our modified, but still naive, Skolemization algorithm produces exactly the same Skolem form, namely $y = c + 1$, as for the original sentence.

[26] My examples used predicate calculus enriched with arithmetic; this is also semidecidable, provided certain restrictions are placed on the fragment of arithmetic employed. This is discussed later in Section A.5.2.

[27] If the search is not exhaustive, the process is no longer a semidecision procedure: if a valid ground instance is found, then the original sentence is certainly valid, but there is no guarantee that the search will find such an instance.

One way to better understand what is going on is to imagine the process of forming the final ground instance as a contest between ourselves, seeking to find substitutions for the essentially existential variables, and a malign opponent who chooses instances for the essentially universal variables [Hod83]. We and our opponent take turns, working from the outside quantifiers towards the innermost; our goal is reach a valid ground sentence, the opponent's goal is prevent us doing so.

If we first consider the valid sentence $(\forall x : (\exists y : y = x + 1))$ we see that the opponent plays first (since the outermost quantifier is essentially universal). The opponent might decide to substitute 17, say, for x, leaving us with $(\exists y : y = 17+1)$. We can now choose to substitute 18 for y, yielding $18 = 17 + 1$, so we win.

Now consider the unsatisfiable form: $(\exists y : (\forall x : y = x + 1))$. Here, we have to play first, so we choose some arbitrary constant, say 9, to leave $(\forall x : 9 = x+1)$. Now our opponent plays, and is careful to choose any number but 8—say 13—thereby winning the contest.

Notice that what happened in the last case was that our opponent did not need to choose a substitution for x until we had made a choice for y; thus, the choice made by the opponent could *depend* on our choice of y—in other words, it could be some *function* $f(y)$ of y. To perform Skolemization correctly, we may only replace essentially universal variables with constants when they are *not* in the scope of some essentially existential quantifier. When they are in the scope of essentially existential quantifiers, the universal variables must be replaced by arbitrary new *function* symbols (called *Skolem functions*) that take the existential variables as arguments. Thus, the correctly Skolemized form of $(\exists y : (\forall x : y = x + 1))$ is $y = f(y) + 1$, which has no valid ground instance. For an example of this kind that is valid, consider $(\exists y : (\forall x : y < x + 1))$, where the variables are constrained to be natural numbers. The Skolem form is $y < f(y) + 1$, which has the valid ground instance $0 < f(0) + 1$ (i.e., $[y \leftarrow 0]$). (The constraint to natural numbers ensures $0 \leq f(y)$ no matter what function is chosen for f.) Because we know nothing about the Skolem function f, it models any strategy our opponent may choose. If we can construct a valid ground instance in the presence of Skolem functions, it means we can always win the contest, no matter what the opponent's strategy.

We now know how to Skolemize first-order sentences correctly, but there is one last problem in this approach to theorem proving. To see this, consider the (valid) sentence

$$(\exists x : \phi(x)) \vee (\exists y : \psi(y)) \supset (\exists z : \phi(z) \vee \psi(z)).$$

The Skolem form of this sentence is

$$(\phi(a) \vee \psi(b)) \supset (\phi(z) \vee \psi(z))$$

where a and b are arbitrary Skolem constants, and we have to find a ground substitution for z that will make the formula a tautology. Unfortunately, it is not hard to see that no such substitution exists.

The problem is that we have to find a single substitution for z without knowing which of $\phi(a)$ or $\psi(b)$ is true (although we do know that at least one of them is true). If we knew which were true, we could find the right substitution ($[z \leftarrow a]$ if $\phi(a)$ is true, otherwise $[z \leftarrow b]$). A little thought should convince us that this is enough: no matter what the true state of affairs, there is always some satisfiable ground instance. Indeed, this is enough, and the result that justifies it is the Herbrand-Skolem-Gödel theorem,[28] which says that a first-order sentence is valid if and only if some finite *disjunction* of ground instances of its Skolem form is tautological. In the case of our example, the disjunction

$$\begin{array}{c} (\phi(a) \vee \psi(b)) \quad \supset \quad (\phi(a) \vee \psi(a)) \\ \vee \\ (\phi(a) \vee \psi(b)) \quad \supset \quad (\phi(b) \vee \psi(b)) \end{array}$$

is valid, and we have succeeded in proving the original theorem.

Almost all mechanical theorem provers employ Skolemization in one form or another, so it is worth having some familiarity with the process. Some systems convert formulas to *prenex* form (in which no quantifiers appear in the scope of a propositional connective) before Skolemizing. This is done by repeatedly using the law $(\exists x : \phi) \equiv \neg(\forall x : \neg\phi)$ and its dual, and by renaming variables if necessary so that expressions such as $(\forall x\colon \phi) \vee \psi$ can be rewritten as $(\forall x\colon \phi \vee \psi)$ (x must not occur free in ψ), and similarly for expressions involving other connectives. In an interactive system, such transformations are disadvantageous, since users are required to examine formulas presented in a very different form than those they typed in. In automatic theorem provers, however, conversion to various special forms can assist the systematic search for a proof. *Resolution* provers, for example, generally eliminate all connectives other than $\wedge$, $\vee$, and $\neg$ (by, for example, transforming $\phi \supset \psi$ into $\neg\phi \vee \psi$), convert the resulting formulas to prenex form, Skolemize them, and then convert to *conjunctive normal form* (CNF) by repeated applications of de Morgan's laws. In CNF, each formula is written as a conjunction of *clauses*, where a clause is a disjunction of *literals*, which are atomic formulas or negations of atomic formulas (the empty clause is identified with *false*).

Propositional formulas in CNF can often be shown to be unsatisfiable by repeated application of the *one-literal rule*: if a clause consists of just a single literal, then that clause can be deleted, and all instances of the negated form of that literal[29] can be deleted from all other clauses: if that deletion results in any clause becoming empty (i.e., *false*) then the original formula was unsatisfiable. For example:

---

[28] This result in model theory should not be confused with Herbrand's Theorem, which is a much deeper result in proof theory.

[29] If the literal is a negation, then its negated form is the unnegated atomic formula (e.g., the negated form of $\neg P$ is simply P).

$(P \vee Q \vee \neg R) \wedge (P \vee \neg Q) \wedge \neg P \wedge R \wedge S$	apply one-literal rule to $\neg P$
$(Q \vee \neg R) \wedge \neg Q \wedge R \wedge S$	apply one-literal rule to $\neg Q$
$\neg R \wedge R \wedge S$	apply one-literal rule to $\neg R$
empty $\wedge\, S$	we are done.

The *resolution* rule for propositional calculus extends the one-literal rule by looking for clauses that contain a *complementary pair* of literals P and $\neg P$. Such a pair of clauses, for example $(P \vee R)$ and $(\neg P \vee Q)$, are then replaced by their *resolvent* $(R \vee Q)$ (thus the one-literal rule is the special case of resolution when either R or Q is empty). Resolution is extended to first-order clauses (i.e., those containing variables) by seeking a single substitution that can be applied to two literals to make them a complementary pair, and then applying the same substitution to the resolvent. The process of constructing a substitution that will cause two literals to become a complementary pair is called *unification*, and the substitution that results is called a *unifier*. Here is an example of first-order resolution:

$(\neg S(y) \vee C(y)) \wedge S(b) \wedge V(a,b) \wedge (\neg C(z) \vee \neg V(a,z))$	$[y \leftarrow b]$ on clauses 1, 2
$C(b) \wedge V(a,b) \wedge (\neg C(z) \vee \neg V(a,z))$	$[z \leftarrow b]$ on clauses 1, 3
$V(a,b) \wedge \neg V(a,b)$	Clauses 1, 2
empty	we are done

Resolution is a complete refutation procedure for first-order logic: if a sentence is unsatisfiable, then resolution will terminate with the empty clause. For theorem proving, we simply conjoin all the premises with the *negated* conclusion, and then use resolution to test whether the result is unsatisfiable: if it is, then the original theorem is true. If the original theorem was untrue, then resolution may not terminate.

The basic resolution and unification algorithms are due to Robinson [Rob65]. A great many extensions and heuristic optimizations to resolution have been developed over the years; some of the more fundamental ones are described in the standard text by Chang and Lee [CL73]; Otter is a modern theorem prover based on highly efficient implementations of several resolution strategies [McC90]. A variety of resolution called SLD-resolution is very effective for Horn clauses (clauses that contain at most one negated literal) and is the technique underlying interpreters for the language Prolog [NS93, Chapter III].

Although resolution theorem provers can be quite effective in some domains, they are of little use in formal methods. Formal methods generally require more than just pure first-order predicate calculus (e.g., they require theories for arithmetic and various datatypes, and possibly set theory or higher-order quantification), and resolution is not at all effective in these contexts.[30] In addition, many of the theorems

[30] If we simply add the axioms for the theories concerned, then the search space becomes so huge that resolution seldom terminates in reasonable time. If we add decision procedures for the theories concerned, then unification needs to be performed modulo these interpreted theories and it is a research topic to make this work effectively [Sti85].

we try to prove in formal methods are untrue when first formulated, and resolution provides little help in such cases (though it can sometimes generate counterexamples). And for true theorems, resolution simply affirms their truth: it does not assist us to develop an argument that can be subjected to human review. Nonetheless, some of the ideas from resolution find application in almost all modern theorem provers. In particular, unification is a fundamental technique for creating substitution instances, and highly efficient (i.e., linear-time) unification algorithms have been developed, as have extensions to the higher-order case.

A.4.2 Sequent Calculus

The alternatives to model theoretic approaches to "theorem proving," such as that sketched in the previous section, are proof theoretic approaches. These include the first-order sequent calculus, which is obtained by extending the propositional sequent calculus of Section A.3.1 with the following rules.

There are four "quantifier rules." In the $\rightarrow \forall$ and $\exists \rightarrow$ rules (i.e., those on the top right and bottom left), a must be a new constant[31] (these rules are the analogs of Skolemization).

$$\frac{\Gamma,\ A[x \leftarrow t] \rightarrow \Delta}{\Gamma,\ (\forall x\colon A) \rightarrow \Delta}\ \forall \rightarrow \qquad\qquad \frac{\Gamma \rightarrow A[x \leftarrow a],\ \Delta}{\Gamma \rightarrow (\forall x\colon A),\ \Delta}\ \rightarrow \forall$$

$$\frac{\Gamma,\ A[x \leftarrow a] \rightarrow \Delta}{\Gamma,\ (\exists x\colon A) \rightarrow \Delta}\ \exists \rightarrow \qquad\qquad \frac{\Gamma \rightarrow A[x \leftarrow t],\ \Delta}{\Gamma \rightarrow (\exists x\colon A),\ \Delta}\ \rightarrow \exists$$

We also need a rule for terminating a branch of the proof tree when we encounter a nonlogical axiom or a previously-proved lemma among the consequents of a sequent:

$$\frac{}{\Gamma \rightarrow \phi,\ \Delta}\ \textbf{Nonlog-ax}$$

where ϕ is a nonlogical axiom or previously-proved lemma.

For convenience in developing proofs, it is useful to provide an additional rule called "cut." This can be seen as a mechanism for introducing a case-split or lemma into the proof of a sequent $\Gamma \rightarrow \Delta$ to yield the subgoals $\Gamma, \phi \rightarrow \Delta$ and $\Gamma \rightarrow \phi, \Delta$. These are equivalent to assuming ϕ along one branch and having to prove it on the other.[32]

[31] Actually, a constant that does not occur in the consequent of the sequent. This constraint is called the *eigenvariable condition*.

[32] Alternatively, applying the rule for negation on the right, this can be seen as equivalent to assuming ϕ along one branch and $\neg\phi$ along the other.

$$\frac{\phi,\ \Gamma \to \Delta \qquad \Gamma \to \phi,\ \Delta}{\Gamma \to \Delta}\ \textbf{Cut.}$$

The Cut Elimination Theorem (also known as Gentzen's Hauptsatz) is one of the most famous results in proof theory: it says that any derivation involving the cut rule can be converted to another (possibly much longer one) that does not use cut. Since cut is the only rule in which a formula (ϕ) appears above the line that does not also appear below it, it is the only rule whose use requires "invention" or "insight"; thus the cut elimination theorem provides the foundation for another demonstration of the semi-decidability of the predicate calculus.

A.5 First-Order Theories

First-order theories are simply those that add some nonlogical axioms (and possibly rules of inference) to the predicate calculus. In this section, I consider four very important classes of first-order theories: equality, arithmetic, simple computer science datatypes (such as lists, trees etc.), and set theory.

All of these are very useful in computer science, and any system intended to support formal methods should normally provide all four.

A.5.1 Equality

The first-order theory of equality (or "identity" as it is sometimes called) simply adds a distinguished two-place predicate (generally called *equals* and written as infix $=$) to the symbols of the predicate calculus. The fundamental idea of equality is that $x = y$ if and only if "anything that may be said of x may be said of y, and vice-versa."[33] This idea of the "identity of indiscernibles" was first stated explicitly by Leibniz. In more technical terms, if some property ϕ holds for x, and $x = y$, then ϕ should also hold when y is substituted for some or all instances of x in ϕ. This can be formalized as follows:

Leibniz' law: $(\forall x, y : x = y \supset \phi \supset \phi[\![x \leftarrow y]\!])$[34]

where I use $\phi[\![x \leftarrow y]\!]$, rather than $\phi[x \leftarrow y]$, to indicate that only some instances of x need be replaced by y.

To get things started, we assert that everything equals itself:

reflexivity: $(\forall x : x = x)$.

[33] Most of this material is drawn from Tarski [Tar76].

[34] The symbol $=$ binds tighter than the propositional connectives.

From these axiom schemes, it is possible to deduce that $=$ satisfies the properties of symmetry and transitivity in addition to reflexivity, and is therefore an equivalence relation.

symmetry: $(\forall x, y : x = y \supset y = x)$,

transitivity: $(\forall x, y, z : x = y \wedge y = z \supset x = z)$.

We can also deduce that $=$ satisfies the property of substitutivity:

substitutivity: for any term or atomic formula ϕ, $x = y \supset \phi = \phi[x \leftarrow y]$

which is what distinguishes true equality from a mere equivalence relation: it says that we can always substitute equals for equals.

Sometimes the quantifier $\exists!$ (pronounced "there exists exactly one") is added to first-order theories with equality. It is defined by

$$(\exists! x : \phi(x)) \equiv (\exists x : \phi(x)) \wedge (\forall x, y : \phi(x) \wedge \phi(y) \supset x = y).$$

Notice that the first conjunct on the right-hand side expresses "at least one," while the second expresses "at most one."

The propositional connective $\equiv$ (equivalence, or "if and only if") satisfies Leibniz' law, and therefore functions as an equality relation on wffs. This means that certain sentences can be proved by applying the rules for equality reasoning to $\equiv$. For example, $(x \equiv y) \supset (x \supset y)$ can be proved from Leibniz' law (let ϕ be x). When this approach can be used, it is fast and simple and generally to be preferred to reasoning from the propositional properties of the connective. However, $\equiv$ is more than an equality relation, so equality reasoning does not capture all its properties and it is sometimes necessary to revert to propositional reasoning on $\equiv$. For example, $(\neg x \vee y) \equiv (x \supset y)$ requires propositional reasoning.

When constructing models for a first-order theory with equality, we usually want the interpretation of "=" to be the identity relation on the domain of the interpretation. Such models are called *normal* and it is possible to show that a first-order system with equality has a model if and only if it has a normal model. Thus, nothing is lost (or gained) by restricting attention to normal models.

Often, we will also want our models to have "no confusion" and "no junk." Suppose we have a theory with just two axioms: $a = b$ and $c = d$ (where a, b, c, and d are constants). Then we could have a model in which the domain of interpretation has but a single element, and all four constants are assigned to that single element. The equation $a = c$ will be true in this model, but cannot be proved from our axioms: the model makes too many things equal (it has confusion). Alternatively, we could have a model with six members m_a, m_b, m_c, m_d, m_x, and m_y. The constants of our theory might be assigned to the first four, leaving the last two as "junk" superfluous to our needs. *Initial* models are, roughly speaking, those that have enough elements so that those assigned to terms that the axioms do not require to be equal can, in fact, be distinct (no confusion), yet with no elements left over (no junk) [GTWW77].

A.5.1.1 Sequent Calculus Rules for Equality

These rules directly encode the axiom schemes of reflexivity and Leibniz' law.

$$\frac{}{\Gamma \to a = a,\ \Delta}\ \textbf{Refl} \qquad\qquad \frac{a = b,\ \Gamma[\![a \leftarrow b]\!] \to \Delta[\![a \leftarrow b]\!]}{a = b,\ \Gamma \to \Delta}\ \textbf{Repl}$$

Observe that the rule **Refl** is an instance of the rule **Nonlog-ax** given in the previous section.

A.5.1.2 Rewriting and Decision Procedures for Equality

Reasoning about equality is so fundamental that most theorem provers provide special treatment for it. For example, chains of equality reasoning such as that required to prove the following theorem arise frequently in formal methods:

$$i = j \wedge k = l \wedge f(i) = g(k) \wedge j = f(j) \wedge m = g(l) \supset f(m) = b(k).$$

Trying to prove formulas such as this by repeated application of Leibniz' law, or the derived axioms and inference rules given earlier, soon becomes hopelessly inefficient as the number or size of the formulas increases. A very efficient method for reasoning with ground equalities of this kind is based on *congruence closure* [DST80, Sho78b].

Equations also commonly arise in the form of definitions, such as that for the absolute value function:

$$|x| = \textbf{if } x < 0 \textbf{ then } -x \textbf{ else } x.$$

One way to prove a theorem such as $|a+b| \leq |a|+|b|$ is to expand the definitions and then perform propositional and arithmetic reasoning. "Expanding a definition" means finding a substitution for the left hand side of the definition that will cause it to match a term in the formula (e.g., $[x \leftarrow a + b]$ will match $|x|$ with $|a + b|$), and then replacing that term by the corresponding substitution instance of the right hand side of the definition concerned—for example,

$$|a+b| = \textbf{if } a+b<0 \textbf{ then } -(a+b) \textbf{ else } a+b.$$

Expanding definitions is a special case of the more general technique of *rewriting*, which can be used with arbitrary equations provided the free variables appearing on the right hand side of each equation are a subset of those appearing on its left. The idea is to find substitutions for the free variables in the left hand side of an equation that will cause it to match some part of the formula being proved; that part is then replaced ("rewritten") by the corresponding substitution instance of the right hand side of the equation concerned. The process of finding substitutions for the free variables in the left hand side by trying to make it equal some subexpression in the formula of interest is called *matching*; it is similar to unification, except that we only seek substitutions for the variables in the equation being matched, and not for the variables in the expression it is being matched against (hence, matching is sometimes described as "one way" unification). Notice that although equations are symmetric (i.e., they mean the same whether written as $a = b$ or as $b = a$), rewriting gives them a left-to-right orientation; when viewed in this way, they are generally called *rewrite rules* rather than equations.

Selection and orientation of equations to be used as rewrite rules, and identification of the target locations where rewriting is to be performed, can be undertaken either by the user or by some automated strategy. One strategy is to rewrite whenever it is possible to do so. Unfortunately, this process may not terminate; a set of rewrite rules is said to have the *finite termination* property if rewriting does always terminate.[35] Often there will be two or more opportunities for rewriting in a given expression; if the final result after rewriting to termination is independent of the choices made at each step, then the set of rewrite rules is said to have the *unique termination* (also known as Church-Rosser) property. If a theory can be specified by a set of rewrite rules with the finite and unique termination properties, then rewriting can serve as a decision procedure for the theory concerned: to decide whether two terms are equal, simply rewrite them to termination; if the results are syntactically identical, then the original terms were equal (i.e., rewriting to termination yields a normal form). This procedure is sound, and for ground formulas it is complete. We must be careful, however, if we wish to deduce disequality. Suppose,

[35] Rules that express commutativity (e.g., $x+y = y+x$) can be applied repeatedly (e.g., $a+b \rightarrow b+a \rightarrow a+b \ldots$) and so render a set of rewrite rules nonterminating. This difficulty can be overcome by imposing restrictions that lead to a true normal form (e.g., only apply the rewrite if the substitution instance for x is less than that for y in some suitable ordering). Similar techniques can be used for operators that are associative (and for the combined associative-commutative case) [Sti81]. These techniques are usually embedded in the matching rather than the rewriting mechanism, and referred to as *AC-matching*.

for example, we used the rewrite rules $a \rightarrow b$ and $c \rightarrow d$ to rewrite a to b and c to d and then observed that b and d are in normal form, but not syntactically identical: may we then deduce $a \neq c$? Plainly it would not be sound to do so in the standard semantics, since we could have a model with only a single element (so that all terms are equal). However, deducing disequalities in this way is sound (and complete) for the *initial* model. Consequently, systems that use rewriting to normal form as their main (or only) means of deduction generally use the initial model semantics (OBJ [Gog89] does this), whereas systems that use rewriting as merely one method among several generally use the classical semantics and do not infer disequalities from unequal normal forms (disequality can then only be inferred from axioms that mention it explicitly).

Given an arbitrary set of equations, there are some quite effective heuristic procedures for testing for the finite and unique termination properties [DJ90]; one, known as Knuth-Bendix completion, can often extend a set of rewrite rules that does not have unique termination into one that does. Beyond ordinary rewriting is *conditional* rewriting, which is used for axioms having the form $a \supset b = c$. Various control strategies are possible: the simplest will only rewrite b to c if a can be proved, others will do the rewrite and carry a along as an additional proof obligation.

Term rewriting is so effective that it provides the main means of deduction in some systems, for example Affirm [Mus80], Larch [GH93], and RRL [KZ88]. The very powerful Boyer-Moore prover [BM79, BM88] integrates a number of techniques, but rewriting is one of the most central. Techniques similar to rewriting are used in resolution provers under the name "paramodulation" and its variants. In general, congruence closure and term rewriting are fundamental to productive theorem proving, and theorem provers lacking these mechanisms should find little application in formal methods.

A.5.2 Arithmetic

Numbers are fundamental to mathematics. Peano's arithmetic [Pea67] (much of which should really be attributed to Dedekind) is a formal system that characterizes the natural numbers (i.e., the nonnegative integers $0, 1, 2, \ldots$)[36] by adding nonlogical axioms to the predicate calculus with equality. The first four axioms introduce the constant 0, the successor function *succ* and the predicate $\mathcal{N}$, which we can read "is a number":

- $\mathcal{N}(0)$ (0 is a number),

[36] Sometimes (and, indeed, in Peano's original formulation) the natural numbers are considered to start at 1; nowadays it is more common to refer to the numbers starting from 1 as the positive integers.

- $\mathcal{N}(x) \supset \mathcal{N}(succ(x))$ (the successor of a number is a number),
- $\mathcal{N}(x) \supset succ(x) \neq 0$ (0 is not the successor of any number),
- $\mathcal{N}(x) \wedge \mathcal{N}(y) \wedge succ(x) = succ(y) \supset x = y$ (numbers with identical successors are identical).

The fifth axiom is the scheme of *mathematical induction*:

- $(\phi(0) \wedge (\forall x : \mathcal{N}(x) \wedge \phi(x) \supset \phi(succ(x)))) \supset (\forall z{:}\mathcal{N}(z) \supset \phi(z))$.

  This says that to establish that some property ϕ holds for any natural number, it is sufficient to prove that it holds for 0 and, given that it holds for some arbitrary natural number x, to prove that it also holds for $succ(x)$.

The first four axioms imply that there are infinitely many numbers; the fifth axiom ensures there are not too many.

The numerals are introduced by regarding 1 as $succ(0)$, 2 as $succ(1)$, and so on. The remaining axioms introduce addition (written as infix $+$): and multiplication (written as infix $\times$). I drop the tedious predicate $\mathcal{N}$ and simply assume that the variables range over numbers (cf. many-sorted logic in Section A.9):

- $x + 0 = x$,
- $x + succ(y) = succ(x + y)$,
- $x \times 0 = 0$,
- $x \times succ(y) = x \times y + x$.

Although Peano's system is considered sound, it is incomplete: there are valid statements in arithmetic that cannot be proved using Peano's axioms. It might seem that this could be remedied by adding more powerful axioms, but this is not so. Gödel's incompleteness theorems, probably the most famous theorems in the whole of logic, show that all reasonably powerful formal systems are necessarily incomplete. The first incompleteness theorem says that any consistent formal system that includes arithmetic must be incomplete;[37] the second says that the consistency of such a system cannot be proved within the system (i.e., the formula that asserts consistency is an example of a true but unprovable statement).[38] Profound though it is, Gödel's first incompleteness theorem is not a limitation on the decidability of

[37] The first incompleteness theorem has been formally verified [Sha94]—almost certainly the hardest mechanically checked formal verification ever undertaken.

[38] Gentzen and Gödel demonstrated the consistency of Peano Arithmetic using metamathematical arguments that cannot be formalized within Peano Arithmetic.

formulas that arise in formal methods any more than the existence of algorithmically unsolvable problems (e.g., the halting problem for Turing machines) is a limitation on the practical utility of computers. The true but unprovable statements concern sweeping properties of logic itself, not the specific kinds of properties we are concerned about in formal methods.[39] Gödel's incompleteness theorems are best seen as affirmations of the remarkably expressive power of arithmetic, rather than limitations on the everyday applicability of logic. The practical limitation on our ability to decide whether or not a certain statement is valid is our ability to find a proof, not whether a proof exists.

There are other arithmetic theories besides Peano's. For example, there are axiomatizations (mostly due to Dedekind [Ded63]) of the rational and the real numbers [Fef89, Sup72]. Axiomatizations of the reals suffer from limitations as surprising as those imposed by the incompleteness theorems. The Löwenheim-Skolem Theorem says that if a first-order theory has a model of infinite cardinality, then it has models of all infinite cardinalities.[40] In particular, this means that any axiomatization of the real numbers has a model that is only countably infinite (i.e., has only as many elements as there are natural numbers). Since Cantor showed (by his "diagonal" argument), that the real numbers are not countably infinite (i.e., they cannot be put into 1-1 correspondence with the natural numbers), this means that no axiomatization of the real numbers can capture the properties of the reals *uniquely*—there will always be *nonstandard* models that satisfy the axioms, but are different from the reals. This discovery is not as disappointing as it may seem: it led to the invention of nonstandard analysis [Rob74], which provides a consistent interpretation to infinitesimals and allows analysis to be built up without recourse to the usual notions of limits and convergence, thereby providing a rigorous reconstruction of early treatments of the calculus [Dau88, Lak78]. Neither do these limitations diminish the practical utility of formal systems of arithmetic in computer science. In fact, nonstandard analysis provides a basis for formal verification of the accuracy of certain floating point calculations [Pra92].

[39] A few unprovable formulas of a genuinely mathematical (as opposed to logical) character are now known: "Since 1931, the year Gödel's Incompleteness Theorems were published, mathematicians have been looking for a strictly mathematical example of an incompleteness in first-order Peano arithmetic, one which is mathematically simple and interesting and does not require the numerical encoding of notions from logic. The first such examples were found in 1977" [PH78]. The example in the cited paper concerns an extension to the Finite Ramsey Theorem.

[40] Infinite cardinalities are explained in Section A.8. There are actually two parts to the Löwenheim-Skolem Theorem as I have stated it: the "downward" half, which allows smaller models to be obtained from bigger ones, is the original Löwenheim-Skolem Theorem; the "upward" half, which allows larger models to be obtained from smaller ones, is really a consequence of the Compactness Theorem. The Compactness Theorem says that if every finite subset of a set of sentences has a model, then the set of sentences has a model.

Since we can formalize Cantor's argument for the uncountability of the reals, there must be some countable model that validates this theorem. A simpler observation of the same kind is "Skolem's Paradox": the set of subsets of the natural numbers must be uncountable (Cantor's Theorem establishes that the powerset of a set has greater cardinality than the original set [Dun91, Chapter 12]), but the formalization of this result is satisfied in some countable model. These are really not the paradoxes they seem: the sets concerned will be uncountable *in the model*, but countable in the "real universe" [EFT84]. That is, the model will be sufficiently impoverished that we cannot construct an injection from the interpretation of the set to that of the naturals, but in the "real universe" we will have more functions available to us and will be able to construct the injection. (Boolos and Jeffrey provide a good discussion of this topic [BJ89, pp. 152–155].)

Formal systems for integer and rational arithmetic similar to one first investigated by Presburger in 1929 are very useful in computer science. Essentially, these are linear arithmetics with addition, subtraction, multiplication, equality, and a "less than" predicate $<$. (By simple constructions, the predicates $>$, $\leq$, and $\geq$ can be added as well.) By linear is meant the restriction of multiplications to literal constants—so that $3 \times x$ is admitted, but $c \times y$ (where c is a symbolic constant) and $x \times y$ (where x is a variable) are not. A valuable property of these arithmetics is that they are decidable, and therefore conducive to efficient theorem proving [Rab78]. Since arithmetic is ubiquitous, support tools for formal methods that lack effective automation of arithmetic reasoning are extremely tedious and unproductive to use. Classical Presburger arithmetic is a first-order theory (i.e., it permits quantification), but it allows only simple constants and not function symbols. Because function symbols are also ubiquitous (they can arise, through the process of Skolemization, even in formulas that do not include them originally), variations on Presburger arithmetic that make different tradeoffs to maintain decidability can be more useful in practice. In particular, tools for formal methods often use Presburger arithmetic restricted to the propositional (i.e., ground) case only, but with extensions (such as equality with uninterpreted function symbols) [NO79, Sho77, Sho79, Sho84] that are undecidable in the quantified theory. These arithmetics are adequately expressive for most problems in computer science, and formal methods tools incorporating efficient implementations of their decision procedures can be very effective in practical applications.

A.5.3 Simple Datatypes

Peano's axioms serve as a prototype for axiom systems specifying other datatypes commonly used in computer science, such as lists, trees, and so on. Usually, these datatypes have some *constructors*, which are constants or functions that generate values of the kind considered (e.g., 0 and *succ* in Peano arithmetic), and some *accessors*, which are functions that reverse the process—breaking a value of the kind under consideration into the components that generated it (there are no accessors in Peano arithmetic as I defined it; the *predecessor* function would be an accessor if it were added to the theory). Datatypes require some axioms to specify the relationship among the constructors and accessors, others to specify that values constructed from equal components are equal, and another that specifies an induction scheme.

As an example, here is a theory of *lists*, with constructors Λ (a constant, representing the empty list) and *cons* (a function that builds a new list from a term and an existing list), and accessors *car* and *cdr*.[41] $\mathcal{L}$ is a predicate that recognizes lists; $\mathcal{L}(l)$ is true exactly when l is a list. The first three axioms state that Λ and *cons* construct lists, and that Λ is different from any list constructed using *cons*.

- $\mathcal{L}(\Lambda)$,
- $\mathcal{L}(l) \supset \mathcal{L}(cons(x, l))$,
- $\mathcal{L}(l) \supset cons(x, l) \neq \Lambda$.

The next two axioms describe the relationship between the accessors *car* and *cdr* and the constructor *cons*:

- $\mathcal{L}(l) \supset car(cons(x, l)) = x$,
- $\mathcal{L}(l) \supset cdr(cons(x, l)) = l$,

  Note $car(\Lambda)$ and $cdr(\Lambda)$ are left unspecified; they are usually taken as errors. More sophisticated logics allow them to be explicitly disallowed.

Next is an *extensionality* axiom (one that says values are equal if their components are).

- $\mathcal{L}(l_1) \wedge \mathcal{L}(l_2) \wedge l_1 \neq \Lambda \wedge l_2 \neq \Lambda \wedge car(l_1) = car(l_2) \wedge cdr(l_1) = cdr(l_2) \supset l_1 = l_2$.

The following is an example of what is sometimes called an *eta* rule (one that says a value is equal to that constructed from its components); it is a lemma that can be proved from extensionality.

- $\mathcal{L}(l) \wedge l \neq \Lambda \supset l = cons(car(l), cdr(l))$.

Next, we have a structural induction scheme: to prove a property ϕ of a general list l, it is enough to prove it for Λ and to prove that if it is true for a list l', then it will also be true for the list $cons(x, l')$.

[41] Sometimes the names *hd* (head) and *tl* (tail) are used instead of *car* and *cdr*, respectively.

- $(\phi(\Lambda) \wedge (\forall x, l'\colon \mathcal{L}(l') \wedge \phi(l') \supset \phi(cons(x, l')))) \supset (\forall l\colon \mathcal{L}(l) \supset \phi(l))$.

One consequence of the induction scheme is that all lists are either Λ or a *cons*:

- $\mathcal{L}(l) \supset l = \Lambda \vee \exists x, l' : l = cons(x, l')$.

Finally, we have a *length* function that can be used in termination arguments for recursively-defined functions on lists:

- $length(\Lambda) = 0 \wedge length(cons(x, l)) = length(l) + 1$.

(More generally, we sanction functions defined in a similar manner to that used for *length*.)

Many datatypes commonly used in computer science (such as trees, stacks, etc.) can be specified by axioms similar to those shown above for lists. Because they are so regular, some computer systems supporting formal specification languages can generate suitable axioms automatically for a certain class of data structures, given only a very compact description of the datatype concerned. For example, PVS [ORS92] generates axioms equivalent to all those shown above (they are slightly different because PVS is a typed logic) from the following specification.

```
list[t:TYPE] : DATATYPE
 BEGIN
  null: null?
  cons (car: t, cdr: list): cons?
 END list
```

The "shell" mechanism of the Boyer-Moore prover [BM88] provides similar capability.

A.5.4 Set Theory

During the nineteenth century, several mathematicians attempted to provide a defensible basis for the manipulations performed in analysis (canceling by dx and the like). Cauchy, for example, gave precise definitions to the notions of limit and convergence that had troubled mathematicians since the invention of calculus. Dedekind and Peano's abstract formulations of the real and natural numbers suggested that it might be possible to construct all of mathematics on a few broad, basic principles. Central to these constructions were the ideas of set theory (mainly developed by Cantor) and logic.[42]

[42] Sources for this material include Hatcher [Hat82], Levy [Lev79], Fraenkel [FBHL84], and Shoenfield [Sho78a].

Frege was the first to develop the notion of logic as a formal system in the modern sense. His system was similar to a modern higher-order system,[43] with a nonlogical component that defined a form of set theory, now known as *naive set theory*. The nonlogical part of Frege's system included a two-place predicate written as infix $\in$ and pronounced "is a member of." The intended interpretation is that $x \in y$ expresses the idea that x is a member of the set y. The important point is that as well as being composed of its members, a set can also be regarded as single entity, and can itself be a member of other sets. Furthermore, it is essential to the constructions of Dedekind and Cantor that sets can have infinitely many members.

It seems reasonable to suppose that two sets are equal if and only if they have the same members. This is called the principle of *extensionality*, and it was an axiom of Frege's system. A set can be specified *extensionally* by simply listing all its members: for example

$$y = \{red, blue, green\}.$$

Another way to specify a set is by stating a property that its members must satisfy: for example,

$$y = \{x \mid \phi(x)\}$$

is the usual notation for saying that the set y consists of exactly those members x satisfying the property ϕ. This is called specifying a set *intensionally*. The idea that every property determines a set (i.e., that we are allowed to specify sets intensionally) is the principle of set *comprehension* (also called *abstraction*). The principle of comprehension, which was another axiom of Frege's system, can be expressed as

$$(\exists y : (\forall x : x \in y \equiv \phi(x))).$$

It says that for any property (wff) ϕ of one free variable, there is a set y consisting of exactly those x satisfying ϕ.

Frege showed that he was able to construct the building blocks of mathematics (such as the natural numbers) within his system, and gave convincing arguments that it could serve as a foundation for the whole of mathematics. Unfortunately, this plan was destroyed by Russell in 1902 [Rus67], who pointed out that Frege's system is inconsistent, and therefore unsound (any sentence can be proved in an inconsistent system). *Russell's Paradox*, as it is called,[44] is simply the principle of comprehension with $x \notin x$ for $\phi(x)$. Intuitively, this can be interpreted as follows: if every predicate determines a set, then consider the set y determined by the predicate

[43] Higher-order systems are described in Section A.10.

[44] It should really not be called a paradox; it is a plain contradiction. Other contradictions were known in naive set theory prior to Russell's discovery. These, too, are generally called paradoxes, although technical literature often uses the term *antinomies*. These other antinomies (for example, those of Cantor and Burali-Forti) involve ideas from set theory (in particular, infinite cardinal and ordinal numbers), whereas Russell's goes to very heart of logic itself.

$x \notin x$. That is, y is the set of all sets that are not members of themselves. Now, is y a member of itself or not? If it is, then it satisfies its defining predicate, so that $y \notin y$—that is, it is *not* a member of itself. Conversely, if y is not a member of itself, it does not satisfy the defining predicate, so that $y \in y$—that is, it *is* a member of itself. Either way we obtain a contradiction.

Frege acknowledged that Russell's Paradox destroyed his system's foundation [Fre67] but it was left to others to reconstruct those foundations on a secure footing. The basic problem is in the unrestricted principle of comprehension; fixing the foundations requires placing some control on the way this principle is employed. There are two main approaches by which this can be done. *Axiomatic set theory* is one approach, *type theory* is the other. Here I sketch the axiomatic approach; type theory is described in Section A.10.

The idea behind axiomatic set theory is to allow new sets to be constructed only from existing sets—that way, we avoid things getting out of hand and leading to the antinomies. The best known axiomatic set theory is called Zermelo-Fraenkel (or ZF), after its inventors. In the most austere presentations of ZF, everything is a set (i.e., has zero or more members); sometimes "urelements" (also called "individuals") are provided as well—these can be members of sets but cannot themselves have members. Since everything can be encoded in set theory without urelements, and since it makes some of the statements simpler, I will consider only sets. Also note that ZF uses a very limited predicate calculus, in which $\in$ and $=$ are the only predicate symbols, and there are no function symbols. Functions and additional predicates (not to mention numbers, and the whole of mathematics) are later constructed *within* ZF.

There are eight axioms in ZF; they can all be stated in several different forms (see [FBHL84] for an extended discussion and [Hal84] for an examination of the underlying intuitions), and I will merely describe them, rather than give their formal statements, here.

Extensionality: says that two sets are equal if they have the same members. In symbols

$$(\forall x\colon x \in a \equiv x \in b) \equiv a = b.$$

We can introduce the notion of subset by a similar construction:

$$(\forall x\colon x \in a \supset x \in b) \equiv a \subseteq b,$$

but notice that this is merely a definition (i.e., a metalinguistic abbreviation), not an axiom.

Pair: says that if we have two sets a and b, then we can form a new set whose members are just a and b. This set is called the *pair-set* of a and b. By taking

$a = b$, this also allows us to form the set $\{a, a\}$, which by extensionality is the same as the singleton set $\{a\}$. (This axiom can be dispensed with if certain technical adjustments are made to provide "function-classes" [Lev79]).

Separation: says that given a set a, we can form a new set b consisting of just those members of a satisfying a property ϕ. This is similar to the unsound axiom of comprehension, except that members of the set defined by the property are required to come from some existing set a. This construction is usually written

$$b = \{x \in a \mid \phi(x)\}.$$

Union: says that if we have a set a (of sets), then we can form a new set consisting of the members of all its members. This set is sometimes called the *sum-set* and written $\bigcup a$.

Iterated application of the axioms of pairing, separation, and union allow us to construct many familiar sets. For example, we can define the *union* $a \cup b$ of two sets a and b to be the sum-set of their pair-set. Then we can construct the set $\{a, b, c, d\}$ as the union of two pairs: $\{a, b\} \cup \{c, d\}$. Intersection and set difference can be defined using separation:

$$a \cap b = \{x \in a \mid x \in b\},$$

$$a \setminus b = \{x \in a \mid x \notin b\},$$

and the properties of associativity and distributivity can be proved from these definitions.

Power set: says that the powerset of a set is a set (i.e., we can talk of the set of all subsets of a given set). The powerset of a set a is usually denoted $\mathcal{P}(a)$ or 2^a.

Infinity: says that there is an infinite set. The axioms introduced so far tell us how to combine existing sets to yield new ones, but they do give us any sets with which to start the process. If we suppose that we had a set to start with—call it X, say, then we could form the emptyset by separation: $\emptyset = \{x \in X \mid x \neq x\}$. We could then form an infinite collection of sets by a recurrence such as

$$\begin{aligned} n_0 &= \emptyset \\ n_{i+1} &= n_i \cup \{n_i\} \end{aligned}$$

(so that $n_0 = \emptyset, n_1 = \{\emptyset\}, n_2 = \{\emptyset, \{\emptyset\}\}, n_3 = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$ and so on[45]). But although we then have an infinite *collection* of sets, we have nothing that

[45] Note the difference between the emptyset and the set having the emptyset as its only member.

allows us to call that collection a *set*, and no other way to be sure that there is an infinite set.

There are many ways to state the axiom of infinity, but one way is to say that the collection of sets n_i defined as above is a set (it is usual also to state that the emptyset is a set, in order to avoid the need to "seed" the process with some arbitrary set X). The set of n_i created in this way is called ω, and it plays a fundamental role in the development of numbers (the number 2, say, is *defined* to be the set n_2 and so on), but how do we know it is infinite? One definition of an infinite set (due to Dedekind) is one whose members can be put into one-to-one correspondence with some strict subset of itself. Since the members of ω can be put into one-to-one correspondence with the strict subset $\omega \setminus n_0$ (just associate n_i with n_{i+1}) we see that ω is infinite. The particular set ω might seem an arbitrary choice, so more general statements of the axiom of infinity sanction the *kind* of recurrence we used to create the members n_i of ω, without singling out that particular construction.

Replacement: says that if f is a function and x is a set, then the collection of all $f(y)$ for $y \in x$ is a set. I have expressed this axiom in terms of a function f, but functions are not primitive in ZF (I will define them shortly), so formal statements of the axiom can be rather complicated.

To understand why it is needed, consider the infinite collection of sets defined by:

$$\begin{aligned} V_0 &= \emptyset \\ V_{i+1} &= \mathcal{P}(V_i) \end{aligned}$$

(so that $V_0 = \emptyset, V_1 = \{\emptyset\}, V_2 = \{\emptyset, \{\emptyset\}\}, V_3 = \{\emptyset, \{\emptyset\}, \{\{\emptyset\}\}, \{\emptyset, \{\emptyset\}\}\}$ and so on). We would like to say that this collection is a set, and the axiom of replacement allows us to do so by exhibiting the function f such that $f(n_i) = V_i$. Since the n_i form a set (by the axiom of infinity), so do the V_i.

Most of the axioms of ZF were first stated by Zermelo, but the axiom of replacement is usually credited to Fraenkel (although Skolem and von Neumann have stronger claims). The importance of this axiom is that it is what allows the construction of the "higher" infinities (e.g., by allowing the collection $\{\omega, \mathcal{P}(\omega), \mathcal{P}(\mathcal{P}(\omega)) \ldots\}$ to be a set; we then take its sum-set, and start taking powersets again, and so on).

Regularity (also called **foundation**): says that every nonempty set (of sets) has a member that has no members in common with the original set. This is a technical axiom, intended to eliminate unintuitive possibilities, such as sets which are members of themselves.

Among its consequences are that every set is *well-founded*: that is, if a is any set and $a_0 \in a$, and $a_1 \in a_0$ ...and $a_{i+1} \in a_i$ and ..., then this sequence eventually terminates.[46] Well-foundedness allows us to prove things about sets by induction over the $\in$ relation. Non well-founded sets have been studied recently, and may have some application in computer science [Acz88].

The eight axioms given above are generally augmented by one more:

Choice: says that for any set of sets, there is a *choice function* that selects a member from each of the member sets. (By the axiom of replacement, we can then form a set containing just the members so chosen.) This axiom was added to support the constructions used in a number of important theorems (e.g., Zermelo's well-ordering of the reals).

ZF plus the axiom of choice is generally denoted ZFC and is considered an adequate foundation for the whole of mathematics.

The axiom of choice is rather different from the other axioms because of its essentially nonconstructive character: it asserts that a choice function exists, but doesn't tell us how to construct one. For this reason, the axiom is sometimes considered a little suspect, and there have been many attempts to find more constructive replacements for it. It turns out that weaker axioms are unable to support some parts of mathematics, and alternative axioms of adequate power (e.g., "every set can be well-ordered" or the result known as Zorn's Lemma) are all equivalent to the axiom of choice [RR63].

Gödel showed in 1939 that the axiom of choice was consistent with the other axioms (i.e., if you couldn't get a contradiction from the other axioms, adding the axiom of choice wouldn't enable you to get one). In 1963, Cohen showed that the negation of the axiom of choice is also consistent with the other axioms—so that this axiom is truly independent of the others.

ZFC minus the axiom of replacement is called Zermelo set theory, and is strictly weaker than ZFC, since it excludes certain transfinite sets. There are other set theories, of which the best known is that of von Neumann, Bernays and Gödel (known as NBG or VNB set theory). This theory has a notion of (proper) "classes" as well as sets (proper classes can have sets as members, but cannot themselves be the members of sets or proper classes). NBG is sometimes preferred to ZFC because it has a finite axiomatization.[47]

---

[46] A relation $\succ$ is well-founded if there are no "infinite descending chains"—i.e., no infinite sequences $a_0 \succ a_1 \succ \ldots \succ a_i \succ \ldots$ A well-founded set is one such that the $\ni$ relation ($a \ni b \equiv b \in a$) is well-founded. This and other properties of relations are described in more detail in Section A.7.

[47] It is also a *conservative extension* of ZF. Conservative extensions are described in the next section.

Some of these distinction can be important when mechanizing formal methods. Although ZF is used by most tools for formal methods based on set theory, Mizar, for example, uses a set theory due to Tarski and Grothendieck [Rud92].

The operator known as Hilbert's ε symbol is related to the axiom of choice. When ϕ is some wff with a single free variable x, then $\varepsilon x : \phi$ denotes "the x such that ϕ"—that is some value that satisfies ϕ, if any such exists, otherwise some arbitrary value. The symbol can be specified by the axiom scheme

$$(\exists x\colon \phi(x)) \supset \phi(\varepsilon x\colon \phi)$$

and can be used to define a choice function (for a set a, let ϕ be $x \in a$). If we denote ZF plus the ε symbol as ZF_ε, then it might look as though $\mathrm{ZF}_\varepsilon =$ ZFC. This is not so, however, for ZF_ε merely adds the ε operator, it does not strengthen the axioms of separation and replacement to allow wffs involving ε (see [FBHL84, page 73, footnote 1] and [Lei69, section 4.4]); thus, although we could use ε to select representatives from a set of sets, we cannot say that this collection of representatives is a set. Hilbert's Second ε Theorem states that any theorem of ZF_ε not involving ε in its statement is also a theorem of ZF (i.e., ZF_ε is a conservative extension of ZF and so the axiom of choice is not a theorem of ZF_ε). If the axioms of separation and replacement are extended to allow wffs involving ε, then we do obtain the axiom of choice.[48]

I have already sketched how familiar set operators such as $\cup$ and $\cap$ can be defined within ZF, next I briefly describe how relations and functions are constructed. This process is essentially similar to "programming" in a very restricted language.

First, the *ordered* pair (a, b) is represented by the set $\{a, \{a, b\}\}$ (this "coding trick" is due to Kuratowski and Wiener). Then we can define the Cartesian product $x \times y$ of two sets x and y by

$$x \times y = \{(a, b) \mid a \in x \wedge b \in y\}.$$

This construction can be justified (i.e., we can establish the right-hand side is a set) by observing that

$$(a \in x \wedge b \in y) \supset (a, b) \in \mathcal{P}(\mathcal{P}(x \cup y))$$

and then using the separation axiom. A *predicate* P on a set x is a subset of x, and I generally write $P(a)$ for $a \in P$. A *relation* R between x and y is defined to be a subset of $x \times y$ and I usually write aRb for $(a, b) \in R$ (if $x = y$, we say a relation *on* x).

[48] In systems (such as PVS) that are based on higher-order logic (see section A.10), it is a simple exercise in specification and theorem proving to state and prove a type-restricted form of the axiom of choice from the axiom for ε.

In set theory, quantified expressions of the form $(\forall x : x \in S \supset \phi(x))$ are usually abbreviated to $(\forall x \in S : \phi(x))$. Similarly, $(\exists x \in S : \phi(x))$ is an abbreviation for $(\exists x{:}\, x \in S \wedge \phi(x))$. Using these notations, the *domain* of R is the set $\{a \in x \mid \exists b \in y : aRb\}$ (i.e., the set of all first components of pairs in R), and its *range* is the set $\{b \in y \mid \exists a \in x : aRb\}$ (i.e., the set of all second components of pairs in R). (Relations on three or more sets can be defined by iterating the construction). The *domain restriction* of R to a set s is $\{(x, y) \in R \mid x \in s\}$ and is denoted $s \lhd R$; The *range restriction* of R to a set s is $\{(x, y) \in R \mid y \in s\}$ and is denoted $R \rhd s$. The *image* of a set s under the relation R is the range of $s \lhd R$; the *inverse image* is the domain of $R \rhd s$. The *inverse* relation R^{-1} of R is given by $R^{-1} = \{(b, a) \in y \times x \mid (a, b) \in R\}$.

A relation f is a *function* from x to y if $(\forall a \in x{:}\, (\forall b, c \in y : a\, f\, b \wedge a\, f\, c \supset b = c))$ (i.e., if at most one member of the range relates to each member of the domain); f is *total* if its domain is the whole of x (otherwise it is *partial*); it is *surjective* if its range is the whole of y. A function is *injective* if $(\forall c \in y{:}\, (\forall a, b \in x : a\, f\, c \wedge b\, f\, c \supset a = b))$ (i.e., if at most one member of the domain relates to each member of the range); a function is *bijective* if it is both injective and surjective. When f is a total function and $a \in x$, the unique b such that $a\, f\, b$ is denoted $f(a)$. It is a matter of considerable debate how $f(a)$ should be treated when f is partial and $a \in x$ is not in its domain; some of the options are discussed in Section A.11.2.

Although ZF set theory provides the underpinnings for some well-known specification notations such as Z [Spi88], there are rather few theorem proving systems for classical set theory. Ontic [McA89], and the theorem prover (called "Never") of the Eves system [CKM$^+$91], are the most substantial. Also, the generic theorem prover Isabelle [Pau88] has an instantiation for ZF set theory. Decision procedures for fragments of set theory have been extensively studied by Cantone and others (see, e.g., [Can91]). Over a period of 20 years, the Mizar project has formalized and mechanically checked an extensive body of mathematics from a set-theoretic foundation [Rud92]. The Mizar proof-checker seems to provide very little automation, however.

A.6 Definitions and Conservative Extension

In Section A.3 I explained how new propositional connectives could be introduced either as abbreviations (a metalogical approach), or by extending the formal system with additional axioms. In this section I examine how new predicate and function symbols can be introduced into a formal system.[49]

It is clear that we can introduce new predicate or relation symbols as abbreviations (they are then generally called *defined symbols*) and that we do not add

[49] Sources for this section include Gordon [Gor88] and Shoenfield [Sho67].

anything new (other than notational convenience) to the formal system by so doing (since we can always simply expand the abbreviations). Thus, if we already have the relation $<$ available, we can introduce $\leq$ as a defined symbol using

$$x \leq y \stackrel{\text{def}}{=} x < y \vee x = y.$$

If we want $x \leq y$ to be an expression in the object language, then we must expand that language by adding $\leq$ as a new predicate symbol, and then supplying

$$x \leq y \equiv x < y \vee x = y$$

as a new axiom.

Now there are many sentences involving $\leq$ that we might have chosen to add as its "defining axiom," and some of these might be dangerous. For example,

$$(x \leq y \vee x = y) \wedge (x \leq y \supset (x < y \vee x = y))$$

enables us to deduce a contradiction (from $3 \neq 2$, deduce $3 \leq 2$ from the first conjunct, and hence the contradiction $3 < 2 \vee 3 = 2$ from the second). What we need are some simple rules that stop us ruining our formal system by introducing inconsistencies under the guise of defining a new symbol. The requirement that a defining axiom should leave the formal system consistent is a little too strong, since the system might have already been inconsistent; what we can require is that the defining axiom should add no *new* inconsistencies or, equivalently, that if the formal system was consistent before, then it will remain so after the addition of our defining axiom.

A definition is presumably intended to introduce some new concept to the formal system, not to add new properties to existing concepts. Thus we should require that the addition of a definition to a theory creates a *conservative extension* of the existing theory. The enlarged theory is an *extension* of the original if all theorems of the original remain so in the enlarged theory; the extension is *conservative* if every theorem of the enlarged theory that is expressed in the language of the original theory is also a theorem of the original theory. Whether or not an extension is conservative can depend on how the additions are made. For example, if we have a theory $\mathcal{T}$ to which we add two new constants a and b to form a theory $\mathcal{T}'$, then $\mathcal{T}'$ is indeed a conservative extension of $\mathcal{T}$. But if we then add the axiom $a = b$ to create the theory $\mathcal{T}''$, then $\mathcal{T}''$ is not a conservative extension of $\mathcal{T}'$, since $a = b$ is a theorem of $\mathcal{T}''$ that is expressed in the language of $\mathcal{T}'$, but it is not a theorem of $\mathcal{T}'$. However, $\mathcal{T}''$ *is* a conservative extension of $\mathcal{T}$. Distinctions such as this are significant when extensions are made incrementally, since many support systems for formal methods require that each step is conservative; such systems would allow $\mathcal{T}$ to be extended to $\mathcal{T}''$ in one step, but not in two steps via $\mathcal{T}'$.[50]

Our original search for ways of extending theories without introducing inconsistencies can now be recast as one for rules of definition that guarantee conservative extension. In the case of a predicate symbol P, the rule that the defining axiom should be of the form

$$P(x_1, \ldots, x_n) \equiv \phi,$$

where P does not appear in the wff ϕ and no variables but $x_1, \ldots, x_n$ are free in ϕ does the job. Our original definition for $\leq$ has this form, and is obviously satisfactory.

For a function symbol f, a suitable defining axiom[51] is one of the form

$$y = f(x_1, \ldots, x_n) \equiv \phi,$$

where ϕ is a term not containing f and in which no variables but $x_1, \ldots, x_n$ and y are free, *provided* we can prove the *existence condition*

$$\exists y : \phi$$

and the *uniqueness condition*

$$\phi \wedge \phi[y \leftarrow z] \supset y = z.$$

For example, to introduce the square root function $\sqrt{x}$, the defining axiom would be

$$y = \sqrt{x} \equiv y \times y = x$$

[50]There are more powerful ways of building up consistent theories than by simply making incremental conservative extensions. For example, Robinson's Consistency Theorem says that if two consistent first order systems are in "complete agreement" on their common language (i.e., for each formula ϕ in their common language, at most one of ϕ or $\neg\phi$ is provable in both systems), then the union of the two systems is consistent [BJ89, chapter 24].

[51]It is reasonable to ask whether every concept (i.e., constant, predicate, or function) can be introduced by an explicit defining axiom: might there not be some that can only be defined implicitly by a complex web of axioms? First of all, we can say that an axiom is *independent* of the others in a theory if there is a model for the others that falsifies the one under consideration. (E.g., non-Euclidean geometries are interpretations of Euclid's system that make the Fifth Postulate false but all the others true, thereby demonstrating that the Fifth postulate is independent of the others.) Padoa extended this idea to a method for showing that a concept q is independent of others $p_0, \ldots, p_n$ under axioms $A_0, \ldots, A_m$: if there are two models that both give the same interpretations to $p_0, \ldots, p_n$ and both valuate $A_0, \ldots, A_m$ to true, but give different interpretations to q, then q is independent of the other concepts. Beth's *Definability Theorem* shows that Padoa's technique is complete: if a concept q is, in fact, independent of the others in a theory, then Padoa's technique will succeed in demonstrating that fact. From this it follows that any concept that is implicitly definable in first-order logic is also explicitly definable (see [BJ89, chapter 24] or [Kle67, § 57]).

(so that ϕ is $y \times y = x$), the existence condition would be

$$\exists y \colon y \times y = x,$$

and the uniqueness requirement would be

$$y \times y = x \wedge z \times z = x \supset y = z$$

(which might be hard to prove on the reals!).

An important special case is the one where y does not appear in ϕ; in this case, the defining axiom is equivalent to the equation

$$f(x_1, \ldots, x_n) = \phi$$

and the existence and uniqueness conditions are always provable. This convenient result can be extended to recursive definitions (i.e., those in which f appears in ϕ) provided they have certain simple forms.

The simplest such form is that of *primitive recursion*:

$$\begin{aligned} f(0, x_1, \ldots, x_n) &= g(x_1, \ldots, x_n) \\ f(i+1, x_1, \ldots, x_n) &= h(f(i, x_1, \ldots, x_n), i, x_1, \ldots, x_n) \end{aligned}$$

where g and h are already defined functions that need not take all of the arguments $x_1, \ldots, x_n$ (nor i in the case of h). This can be also be written in the form

$$f(i, x_1, \ldots, x_n) = \textbf{if } i = 0 \quad \begin{array}{l} \textbf{then } g(x_1, \ldots, x_n) \\ \textbf{else } h(f(i-1, x_1, \ldots, x_n), i-1, x_1, \ldots, x_n). \end{array}$$

(Clearly, the induction variable i need not be the first argument.) As an example, observe that Peano's axioms for multiplication

$$\begin{aligned} x \times 0 &= 0 \\ x \times (i+1) &= x \times i + x \end{aligned}$$

are primitive recursive ($n = 1$, $g(x) = 0$, $h(z, i, x) = z + x$) and can also be written in the alternative form

$$x \times i = \textbf{if } i = 0 \textbf{ then } 0 \textbf{ else } x \times (i-1) + x.$$

When we enlarge our formal system with constants, functions, and predicates that define some new "datatype," we often wish to define additional functions by recursion on the structure of the datatype. For example, if we have introduced lists, with Λ, *cons*, *car*, and *cdr*, we might want to define a function *sum* that adds up the value of all the nodes. We might define this by

$$\begin{array}{rcl} sum(\Lambda) & = & 0 \\ sum(cons(x,l)) & = & x + sum(l) \end{array}$$

or, equivalently, by

$$sum(l) = \textbf{if } l = \Lambda \textbf{ then } 0 \textbf{ else } car(l) + sum(cdr(l)).$$

Plainly, we could prove a metalogical theorem that this construction provides conservative extension just as primitive recursion does, and we could even prove more general results for all datatypes defined by certain structured sets of axioms. However, another way to justify these definitions is by providing a *measure* function from the recursion variable (here l) to the natural numbers and proving that the value of this function strictly decreases across the recursion. In this case, the obvious measure function is the *length* of the lists concerned, and the theorem we would have to prove is

$$l \neq \Lambda \supset length(cdr(l)) < length(l),$$

which will be provable in any well constructed theory of lists.

There are several tricky details that need to be taken care of in specification languages that provide a "definitional principle" for recursions of this sort. For example, taking the identity function as the measure, the definition

$$silly(n) = \textbf{if } n = 0 \textbf{ then } 0 \textbf{ else } silly(n-2) \times n$$

might appear conservative, despite the fact that it "steps over" the "termination condition" $n = 0$ when applied to an argument that is an odd number. Obviously, the definitional principle must eliminate this possibility if it is to be sound, but methods for doing this are best examined in the context of the particular language concerned.[52]

Not all functions are primitive recursive. The standard example is a function due to Ackermann, which in its modern form is given by

$$\begin{array}{rl} ack(m,n) = & \textbf{if } m = 0 \textbf{ then } n+1 \\ & \quad \textbf{elsif } n = 0 \\ & \quad\quad \textbf{then } ack(m\text{-}1,\ 1) \\ & \quad\quad \textbf{else } ack(m\text{-}1,\ ack(m,\ n\text{-}1)). \end{array}$$

[52] In PVS [ORS92], for example, the definition of *silly* would be considered type-incorrect because the recursive argument $n - 2$ cannot be shown to be a natural number (unlike $n - 1$, which could be shown to be a natural number if n is, in the context $n \neq 0$).

This function cannot be defined by (first-order) primitive recursion.[53] However, a definition such as this can be shown to be conservative by demonstrating that the "size" of its pair of arguments (notice that *ack* is recursive in both its arguments) decreases across the recursions according to a *lexicographic* ordering on pairs:[54]

$$(m_1, n_1) < (m_2, n_2) \equiv m_1 < m_2 \vee (m_1 = m_2 \wedge n_1 < n_2).$$

Here, we need

$$\begin{aligned} (m-1, 1) &< (m, 0), \\ (m-1, ack(m, n-1))^{55} &< (m, n), \text{ and} \\ (m, n-1) &< (m, n), \end{aligned}$$

which are all true in the lexicographic ordering.

We might try to justify use of lexicographic ordering to ensure conservative extension using a naive extension to the measure function approach. That is, we could look for a function *size* that would map the pair of arguments to *ack* into a single number that is strictly decreasing across the recursions:

$$\begin{aligned} size(m-1, 1) &< size(m, 0), \\ size(m-1, ack(m, n-1)) &< size(m, n) \text{ and} \\ size(m, n-1) &< size(m, n). \end{aligned}$$

A plausible function is

$$size(m, n) = \xi \times m + n,$$

provided ξ is big enough to ensure

$$\xi \times (m-1) + ack(m, n-1) < \xi \times m + n$$

(from the second case)—that is

$$\xi > ack(m, n-1) - n.$$

Now $ack(m, n-1)$ grows much faster than n, so the right-hand side is unbounded—and this suggests that ξ needs to be infinite!

In fact, this is not so implausible as it might seem, and it serves to motivate introduction of the transfinite ordinals, which are numbers with the properties we require.

[53] Though it can be defined by a higher-order primitive recursion: that is, a definition restricted to the form of primitive recursion, but with functions allowed as arguments [Gor88, pp. 96, 97].

[54] This ordering is called lexicographic because it is the way words are ordered in a dictionary—first on the initial letter, then on the second, and so on.

[55] Since *ack* is the function whose termination we are trying to prove, it should not appear in its own termination argument. A better form of this obligation is $\forall f\colon (m-1, f(m, n-1)) < (m, n)$. The quantification over all *functions* f (of the appropriate signature) is a *higher-order* construction. Since I have not yet introduced higher-order logic (see section A.10), I will continue with the rather suspect construction that uses *ack* itself.

A.7 Ordinal Numbers

Natural numbers can be used in two ways. If we have the members of a set somehow arranged in order, then we can count off its members, and can speak of the "second" member, and the 365th, and so on. Numbers used in this way are called *ordinals*. Alternatively, we can simply ask how many members there are in the set; numbers used in this way are called *cardinals*. For finite sets, the natural numbers serve as both ordinals and cardinals; with infinite sets, however, things get a little more complicated.

The extension of ordinal and cardinal numbers to infinite sets was the work of Cantor.[56] First, though, it is necessary to distinguish different kinds of infinity. Aristotle distinguished the *potential* from the *actual* or *completed* infinite; the potential infinite typically arises when some variable can take on values without limit. Any particular value is finite, but the range of possibilities is unbounded. Prior to Cantor, it was generally assumed that mathematics could only be concerned with the potential infinite; the actual infinite was considered unfathomable. Cantor broke through this restriction of thought and divided the actual infinite into the increasable infinite, or *transfinite*, and the *absolute* infinite. For Cantor, the transfinite

> "is in itself constant, and larger than any finite, but nevertheless unrestricted, increasable, and in this respect thus unbounded. Such an infinite is in its way just as capable of being grasped by our restricted understanding as is the finite in its way." [Hal84, page 14]

Ordinal numbers are used to count the members of a set arranged in some order, so first we need some terminology for ordering relations. A binary relation $<$ on a set a is said to be

reflexive: if $x < x$,

irreflexive: if $\neg(x < x)$,

symmetric: if $x < y \supset y < x$,

asymmetric: if $x < y \supset \neg(y < x)$,

antisymmetric: if $x < y \wedge y < x \supset x = y$,

transitive: if $x < y \wedge y < z \supset x < z$,

[56] Sources for this section include Cantor [Can55], Hallett [Hal84], Hatcher [Hat82], Johnstone [Joh87] and Phillips [Phi92]. A less technical but very readable introduction to Cantor's work is provided by Dunham [Dun91, chapters 11 and 12].

connected: if $x < y \vee y < x$,

trichotomous: if $x < y \vee y < x \vee x = y$, and

well-founded: if $\forall b \subseteq a : b \neq \emptyset \supset \exists x \in b : \neg \exists y \in b : y < x$ (i.e., every nonempty subset of a has a $<$-minimal member or, equivalently, there are no infinite $<$-descending chains).

Furthermore $<$ is said to be a

preorder: if it is reflexive and transitive,

partial order: if it is reflexive, transitive, and antisymmetric,

total order: if it is reflexive, transitive, antisymmetric, and trichotomous,

strict order: if it is irreflexive, transitive, and antisymmetric (actually, the third is implied by the first two),

strict total (or **linear**) **order**: if it is irreflexive, transitive, antisymmetric, and trichotomous (again, the third is implied by the first two),

well-order: if it is irreflexive, transitive, antisymmetric, trichotomous, and well-founded (actually, the last condition implies the first, the last two imply the second, and the first two imply the third—so that all we really need is well-founded and trichotomous). Equivalently, we can say that a well-ordering is a well-founded linear order.

The idea of the ordinals is that they should be sets with a canonical well-order. Then, since every set has a well-ordering in ZFC (the existence of a well-ordering is equivalent to the axiom of choice), the members of any set can be placed in order alongside those of an ordinal, whose members then act as numbers for all the positions in the line. (The unique ordinal isomorphic to a given set a, well-ordered by $<$, is called the *order-type* of $(a, <)$.)

In ZFC, the only primitive relations are $\in$ and $=$, and of these only $\in$ is a candidate for forming a well-ordering. Then, in order to achieve trichotomy, we will need to construct sets whose members are such that of any distinct pair of members, one is a member of the other (this is *connectedness* on $\in$). It turns out that sets with these properties can be found among the *transitive* sets, where a set a is called transitive if

$$y \in x \wedge x \in a \supset y \in a.$$

(Transitive sets a have the properties $\bigcup a \subseteq a$ and $a \subseteq \mathcal{P}(a)$—i.e., every element of a is also a subset of a.) We then define the *ordinals* as the transitive sets that are well-ordered by $\in$ (or, equivalently, the sets that are both transitive and connected).

$\emptyset$ is an ordinal, and if x is an ordinal, then so is $x \cup \{x\}$ (which I will denote x'). Consequently, the sets $n_0, n_1, \ldots n_i, \ldots$, which were defined in just this way in the previous section but one, are ordinals (they are called the *von Neumann natural numbers*). An ordinal y such that $y = x'$ for some ordinal x is called a *successor*; all other ordinals (except $\emptyset$) are called *limits*. Now, we know (from the axiom of infinity) that the set ω of all the von Neumann natural numbers exists; it can be seen to be transitive and connected, and is therefore an ordinal. It can be shown not to be a successor, and is therefore a limit ordinal (in fact the smallest one). It is also our first infinite number. We can create larger infinite ordinals by iterating the process used to create ω. That is, we can form the successors $\omega', \omega'', \omega''', \ldots$ and then collect these (together with $0, 1, 2, \ldots, \omega$) up into a set, which we will call $\omega \times 2$ and which is another ordinal. In this way we can form $\omega \times 3, \omega \times 4, \ldots$ To get further, we need to define the operations of addition, multiplication, and exponentiation on ordinals. These are each defined by recursion, with separate cases according to whether the second argument is 0, a successor, or a limit.

$$\begin{array}{lcl} \alpha + 0 & = & \alpha \\ \alpha + \beta' & = & (\alpha + \beta)' \\ \alpha + \beta & = & \bigcup\{\alpha + \gamma \mid \gamma < \beta\} \text{ where } \beta \text{ is a limit} \end{array}$$

(The less-than relation $<$ on the ordinals is just set-membership $\in$.) If α is the order-type of a set a, and β the order-type of a set b, then $\alpha + \beta$ is the order-type of the disjoint union of a and b.

$$\begin{array}{lcl} \alpha \times 0 & = & 0 \\ \alpha \times \beta' & = & \alpha \times \beta + \alpha \\ \alpha \times \beta & = & \bigcup\{\alpha \times \gamma \mid \gamma < \beta\} \text{ where } \beta \text{ is a limit} \end{array}$$

If α is the order-type of a set a, and β the order-type of a set b, then $\alpha \times \beta$ is the order-type of the Cartesian product $a \times b$ under lexicographic ordering.

$$\begin{array}{lcl} \alpha^0 & = & 1 \\ \alpha^{\beta'} & = & \alpha^\beta \times \alpha \\ \alpha^\beta & = & \bigcup\{\alpha^\gamma \mid 0 < \gamma < \beta\} \text{ where } \beta \text{ is a limit} \end{array}$$

If α is the order-type of a set a, and β the order-type of a set b, then β^α is the order-type of the function space $a \rightarrow b$.

Note that the first two cases in each set of equations are the same as the corresponding Peano axioms (with the $'$ operator in place of *succ*), so that the successor ordinals (and, in particular the von Neumann natural numbers) behave just like the natural numbers. The case of the limit ordinals is a little different, however, for addition and multiplication are not commutative in these cases (e.g., $1 + \omega = \omega$ whereas $\omega + 1 = \omega'$, and $2 \times \omega = \omega$ whereas $\omega \times 2 = \omega + \omega$).

In this way we can form ordinals up to ϵ_0, which is the limit of the sequence $\omega, \omega^\omega, \ldots, \omega^{\omega^{\omega^{\cdot^{\cdot^{\cdot}}}}}$ and has the property $\omega^{\epsilon_0} = \epsilon_0$. (The ordinals continue beyond this, but for formal methods ϵ_0 is usually adequate.)

Each nonzero ordinal α below ϵ_0 has a unique representation (called Cantor normal form) with the structure

$$\alpha = \omega^{\alpha_0} \times a_0 + \omega^{\alpha_1} \times a_1 + \cdots \omega^{\alpha_n} \times a_n,$$

where $\alpha \geq \alpha_0 \geq \alpha_1 \geq \cdots \geq \alpha_n$ are ordinals, and $a_0, a_1, \ldots a_n$ are nonzero natural numbers.

Constructive representations of the Cantor normal form are quite convenient in formal methods for establishing "termination" arguments for recursions on tree-like data structures, and are used, for example, in the Boyer-Moore prover and in PVS. (The ordering relation $<$ can be defined by primitive recursion on the ordinals below ϵ_0, so its own well-definedness is easily established in most formal methods). Returning to the example of Ackermann's function that motivated this introduction of the ordinals, we can now see that our treatment of its termination becomes satisfactory if we set $\xi = \omega$.

The definitions given above for ordinal addition, multiplication, and exponentiation are examples of definition by *transfinite recursion*. There is a corresponding induction scheme called *transfinite induction*, which can be described as follows:

$$\text{If} \left[\begin{array}{l} \phi(0) \wedge \\ \forall \alpha \in \mathcal{O} : \phi(\alpha) \supset \phi(\alpha') \wedge \\ \forall \beta \in \mathcal{O}_L : (\forall \gamma < \beta : \phi(\gamma)) \supset \phi(\beta) \end{array}\right] \text{then } \forall \alpha \in \mathcal{O} : \phi(\alpha).$$

Where $\alpha \in \mathcal{O}$ means that α is an ordinal, and $\alpha \in \mathcal{O}_L$ that it is a limit ordinal.

Another way to write transfinite induction is

$$(\forall \alpha \in \mathcal{O} : (\forall \beta \in \mathcal{O} : \beta < \alpha \supset \phi(\beta)) \supset \phi(\alpha)) \supset (\forall \gamma \in \mathcal{O} : \phi(\gamma)).$$

Written in this form, it can be seen that transfinite induction owes more to the fact that $<$ is a well-ordering, than to the fact that the values concerned are ordinals. This gives rise to *well-ordered* induction, which is simply the second form of transfinite induction generalized to the case where $\mathcal{S}$ is a set well-ordered by $<$

$$(\forall \alpha \in \mathcal{S} : (\forall \beta \in \mathcal{S} : \beta < \alpha \supset \phi(\beta)) \supset \phi(\alpha)) \supset (\forall \gamma \in \mathcal{S} : \phi(\gamma)).$$

In fact, this induction scheme is sound when $<$ is merely a well-founded relation, in which case it is known as *well-founded induction* (it is also called *Noetherian* *induction*, after Emmy Noether). Well-founded induction is the most general induction scheme: transfinite induction, and all the other induction schemes that we have seen, can be derived from it. It is a great convenience if tools for formal methods are able to support these very general induction schemes.

A.8 Cardinal Numbers

The transfinite ordinals are needed to establish the soundness of certain constructions in formal methods. Transfinite cardinal numbers, on the other hand, find little application in formal methods, so I will just briefly mention them.[57]

The *cardinality* (or cardinal number) of a set a, often written $|a|$, is the "number" of members it contains. This concept of "number" is straightforward for finite sets, but needs care when we consider infinite sets. The key idea (due to Cantor) is to start by defining two sets to have the same "size" (the technical term is *equipollent*) if the members of one can be put into bijective (i.e., 1-1 and onto) correspondence with those of the other. A set a is smaller than a set b if there is an injection (i.e., 1-1 function) from a to b, but not vice-versa.[58]

The cardinal number of a set could be defined as some canonical member chosen from the collection of sets that have the same size. The trouble is that this collection is too big to be a set, so we cannot define it within set theory. However the ordinals having the same size as a given set do constitute a set; and since any set of ordinals is well-ordered, we can choose the least member of that set to be the canonical representative that is the cardinal number of the original set. It is easy to see that the transfinite cardinals are chosen from the limit ordinals. The smallest transfinite cardinal is ω; when considered as a cardinal, it is usually written $\aleph_0$ and pronounced "aleph-null." Sets of cardinality $\aleph_0$ (i.e., those whose members can be put into bijective correspondence with the natural numbers) are called *countably* or *denumerably* infinite. (Recall Dedekind's definition that a set is infinite if it has the same size as some strict subset of itself.) All the ordinals mentioned in the previous section, including ϵ_0 are countable. The smallest uncountable ordinal is denoted Ω; as a cardinal it is denoted $\aleph_1$.

Cantor's theorem demonstrates that the size of any set is strictly less than that of its powerset,[59] so we know $|\omega| < |\mathcal{P}(\omega)|$. We have defined $|\omega| = \aleph_0$, and cardinal arithmetic is defined so that $|\mathcal{P}(\omega)| = 2^{\aleph_0}$. Thus $\aleph_0 < \aleph_1 \leq 2^{\aleph_0}$. The

[57] This section draws on Halmos [Hal60].

[58] The Schröder Bernstein theorem states that if there is an injection from a to b, and vice-versa, then the two sets have the same size. Its proof is a standard test piece in mechanized theorem proving. (It is the transfinite cases that are interesting.)

[59] This is another standard small problem for mechanized theorem proving. Again, it is the transfinite cases that are interesting.

continuum hypothesis conjectures that the second of these relations is not strict: i.e., $\aleph_1 = 2^{\aleph_0}$. Now the set of real numbers has the same size as $\mathcal{P}(\omega)$, so another (in fact, the original) way to state the continuum hypothesis is that the cardinality of the reals is the least uncountable cardinal. (The set of real numbers is also known as the continuum, hence the name of the hypothesis.) The continuum hypothesis is known (by results of Gödel and Cohen) to be independent of the axioms of set theory.

A.9 Many-Sorted Logic

Classical first-order theories treat all individuals as belonging to some single universe. It is sometimes convenient to distinguish different *sorts* of individuals within the universe. For example, we may want some variables to represent natural numbers, others reals, and so on. Many-sorted logic allows first-order systems to be extended in this way. Mathematicians generally use subscripts to indicate the sort to which a variable belongs, so that x_σ, for example, indicates a variable x of some sort called σ. Quantification in many-sorted logic extends only over the sort of the variable concerned, so that

$$(\forall x_\sigma : \phi(x))$$

means that the formula ϕ with free variable x is true whenever x is replaced by any value of sort σ (for simplicity, I generally drop the sort subscript once the variable has been introduced). Functions and predicates also indicate the sorts of values over which they operate. For example, the function *abs* whose value is the natural number that is the absolute value of its integer argument will be described as having *signature* $\mathcal{Z} \to \mathcal{N}$, where $\mathcal{Z}$ is the sort of integers, and $\mathcal{N}$ that of natural numbers. As with individuals, the signature of a function can be indicated in formulas as a subscript, for example

$$x_{\mathcal{Z}} \neq 0 \supset abs_{\mathcal{Z}\to\mathcal{N}}(x_{\mathcal{Z}}) > 0.$$

The model theory of predicate calculus is adjusted in the many-sorted case so that each sort is interpreted by its own specific "carrier set" and various technical adjustments are made (for example, sorts should generally not be empty) so that the resulting system is sound and complete. Many-sorted systems are very natural for computer science, since computer programs usually need to distinguish the different kinds of objects represented and manipulated.

A.10 Typed Systems and Higher-Order Logic

While Zermelo and Fraenkel developed axiomatic set theory, Russell [WR27] explored a different approach to the construction of a consistent set theory.[60] Considered from one perspective, Russell's paradox demonstrates that Frege's axiom of comprehension is dangerously unrestricted. This is, essentially, the perspective of axiomatic set theory: the ZF axioms break the unrestricted connection between predicates and sets given by the axiom of comprehension and substitute more constrained ways of constructing sets. But there must be more to it than this, for it is possible to reproduce Russell's paradox without ever mentioning sets: simply consider the predicate that characterizes those predicates that do not exemplify themselves.[61] Predicates such as this are *higher order*: their construction involves predicates that apply to predicates, and quantification over predicates. For example, if we abbreviate the predicate that characterizes those predicates that do not exemplify themselves by $PNET$ (for Predicates Not Exemplifying Themselves), then the definition for this predicate is $PNET(P) \equiv \neg P(P)$. Russell's paradox is obtained by substituting $PNET$ for the free predicate variable P in this definition to obtain $PNET(PNET) \equiv \neg PNET(PNET)$.

Frege's system was higher-order, and we can now see that axiomatic set theory does much more than replace the axiom of comprehension with the axioms of ZF: it also eliminates all functions and all predicates but $\in$ and $=$ from the logic and restricts quantification to the first-order case (i.e., variables can range only over sets).[62] ZF then reconstructs functions and predicates *within* set theory (as sets of pairs) and in this way develops an adequate foundation for mathematics. This approach is that associated with the *formalist* school of Hilbert, and is essentially mathematical and pragmatic; it is mathematical because it is based on essentially mathematical intuitions. No one would claim that the axiom of replacement, say, is an elementary truth, fundamental to all rational thought; rather, it is a mathematical construction (and a pragmatically motivated one at that).

Unlike the formalists, Frege and Russell were *logicists*: they wanted logic to comprise *exactly* the fundamental truths of rational thought, and then wanted to show that mathematics followed from such a logic: they didn't want to axiomatize mathematics, they wanted to *derive* it. What Russell sought to do, therefore, was

[60] Sources for this material include Andrews [And86], Hatcher [Hat82], Benacerraf and Putnam [BP83], van Bentham and Doets [vBD83], and Hazen [Haz83].

[61] A predicate exemplifies itself if it has the property that it characterizes: for example, the predicate that characterizes the property of being a predicate is clearly a predicate, and does exemplify itself; whereas the predicate that characterizes the property of being a man is not a man, and so does not exemplify itself.

[62] The idea that set theory should be based on first-order logic was due to Skolem. A very readable account of the emergence of first-order logic is given by Moore [Moo88].

to retain all the generalities of Frege's system, with arbitrary function and predicate symbols, and higher-order quantification (because these are necessary to capture informal logical discourse, and to derive mathematics without additional axioms), but to find some minimal and plausibly motivated restriction that would keep the antinomies at bay.

Russell and Poincaré, in the course of an exchange of letters, decided that the source of the antinomies was what Russell called "vicious circle" constructions, and what Poincaré called *impredicative* definitions [Gol88]. An impredicative definition is one that has a particular kind of circularity; it is not the circularity of recursion, but one where a thing is defined by means of a quantification whose range includes the thing being defined. Thus, a set b is defined impredicatively if it is given by forms such as

$$b = \{x \mid \forall y \in a\colon P(x, y)\},$$

where P is a predicate and b may be an element of a.

In work culminating in the Principia Mathematica [WR27] (written with A. N. Whitehead), Russell developed his *ramified theory of types* that augments higher-order logic with rules that exclude impredicative definitions.[63] There are two components to the ramified theory: *types* and *orders*. The second of these causes certain technical difficulties, and Russell introduced an "axiom of reducibility" to get around them. Chwistek and Ramsey thought that this axiom vitiated the purpose of orders, and that one might as well have started out with just the notion of types and never bothered with orders. However, this simpler theory is not predicative and is vulnerable to some paradoxes that are avoided by the ramified theory.

Accordingly, Ramsey (building on an observation of Peano) divided the paradoxes into those he called *logical* (such as Russell's, Cantor's and Burali-Forti's), and those he called *semantic* or *epistemological* (such as the Liar, Richard's, Grelling's, and Berry's[64]); he then argued that the epistemological paradoxes are the concern of philosophy or linguistics, not logic, since they concern the interpretation of concepts like "nameable," not basic problems in the machinery of deduction. Ramsey argued that the requirement on a logic is that it should exclude the logical paradoxes. He

[63] Limitation to predicative definitions is a powerful restriction: it eliminates construction of the Dedekind cut, and with it a good part of analysis, and also Cantor's theorem and the general theory of cardinal numbers and the higher infinities. Furthermore, from the Platonist perspective adopted by most mathematicians (namely, that mathematical objects really exist), there is nothing objectionable in impredicative definitions: the "vicious circle" disappears if we conceive of the offending definitions as selecting some member from a pre-existing collection. Consequently, few mathematicians espouse the rigors of a predicative type theory. (See Hazen [Haz83] for a readable account of predicative logics.)

[64] Berry's is the easiest of these to describe: consider "the least natural number not nameable by an English phrase of less than 200 letters." The material in quotation marks is an English phrase of less than 200 letters that names this number.

noted that types are adequate, on their own, for this; orders and the complexities of the ramified theory are needed only to exclude impredicative definitions and the epistemological paradoxes. Ramsey [Ram90] then reconstructed Russell's system (principally by eliminating orders) so that "its blemishes may be avoided but its excellences retained" to yield what is now called the *simple theory of types* (its modern formulation is due to Church).[65]

Roughly speaking, simple type theory avoids the logical paradoxes of naive set theory by "stratifying" sets according to their "type" or "level." Individual elements can be considered to belong to level 0, sets of those elements to level 1, sets of sets of elements to level 2, and so on. The comprehension axiom

$$(\exists y : (\forall x : x \in y \equiv \phi(x)))$$

is then restricted to the case where x is of a lower level than y. I say "roughly speaking" because simple type theory is a logic of *predicates* rather than sets: instead of saying "x is a member of the set y" (i.e., $x \in y$), in type theory it is more common to say "x has property y" (i.e., $y(x)$). In terms of predicates, simple type theory requires that each predicate has a higher type than the objects to which it applies (and therefore a predicate cannot apply to itself). The "definition" of the predicate *PNET* given earlier is inadmissible because its right-hand side is not well-typed.

I just said that simple type theory is a logic of predicates, but it is usually built on a more primitive foundation that takes (total) functions as the basic elements, with predicates as a special case. Within this framework, functions are allowed to take other functions as arguments and may return functions as values. The *type* of a function indicates the "level" at which it operates in a more sophisticated way than the simple numbering scheme for levels suggested earlier. Thus, if ι is the type of individuals (which can be considered as functions of zero arguments), the type of functions from individuals to individuals can be denoted $\iota \rightarrow \iota$, and a function on those functions will have type $(\iota \rightarrow \iota) \rightarrow (\iota \rightarrow \iota)$.[66]

It is generally convenient to use a many-sorted foundation, and to subsume sorts within the type structure, so that a function type $(\mathcal{Z} \rightarrow \mathcal{N}) \rightarrow \mathcal{N}$ indicates

---

[65] The simple theory of types is not the same as the ramified theory plus the axiom of reducibility: the simple theory is *extensional*, whereas the other is not. Extensional means that coextensive properties are considered identical (i.e., $(\forall x\colon \phi(x) = \psi(x)) \supset \phi = \psi$). Examples of nonextensional properties are "propositional attitudes": a person may *want* or *wonder about* one proposition but not another even though they are, in fact coextensive. The classic example is:

Scott is the author of Waverley,
King George wonders whether Scott is the author of Waverley,
―――――――――――――――――――――――――――――――
King George wonders whether Scott is Scott,

whose absurdity is taken as evidence that "wonders whether" in nonextensional. Unlike the simple theory, the ramified theory of types (even with the axiom of reducibility) can give an account of nonextensional properties [Haz83, section 5].

[66] Types allow us to distinguish $(\iota \rightarrow \iota) \rightarrow (\iota \rightarrow \iota)$ from $(\iota \rightarrow \iota) \rightarrow \iota$, which cannot be done by simply numbering "levels."

a function whose values are of type[67] natural number and whose arguments are functions from integers to natural numbers. Within this framework, we distinguish a type $\mathcal{B}$ (for "Boolean," though mathematicians generally write it o), and then view predicates as functions returning type $\mathcal{B}$. Thus, the predicate $<$ on the natural numbers can be considered a function of type $\mathcal{N} \times \mathcal{N} \rightarrow \mathcal{B}$. Next, since the principle of comprehension holds that every predicate determines a set, we dispense with the separate notion of set and simply identify sets with predicates: $x \in y$ is identified with $y(x)$.[68]

In this scheme, dropping some of the more tedious subscripts, the comprehension axiom becomes

$$(\exists y_{\sigma \rightarrow \mathcal{B}} : (\forall x_\sigma : y(x) \equiv A_{\mathcal{B}}))$$

where $A_{\mathcal{B}}$ is a formula in which y does not occur free. The typing restrictions attached to x and y prevent the kind of self-reference that leads to the paradoxes of naive set theory. The same construction can be used to assert the existence of general functions as well as predicates:

$$(\exists y_{\sigma \rightarrow \tau} : (\forall x_\sigma : y(x) = A_\tau)). \tag{A.2}$$

The function $y_{\sigma \rightarrow \tau}$ whose existence is thereby asserted can be denoted

$$(\lambda x_\sigma : A_\tau)_{\sigma \rightarrow \tau}.$$

The λ here is called the *abstraction* operator; $(\lambda x_\sigma : A_\tau)_{\sigma \rightarrow \tau}$ is the name of the function whose value on argument x_σ is A_τ, where x_σ presumably occurs free in A_τ. For example,

$$(\lambda x_{\mathcal{Z}} : (\textbf{if } x < 0 \textbf{ then } -x \textbf{ else } x)_{\mathcal{N}})_{\mathcal{Z} \rightarrow \mathcal{N}}$$

is the "absolute value" function on the integers. λ is a variable-binding operator, just like the quantifiers.[69]

Using this notation, the comprehension axiom (A.2) becomes

$$(\forall x_\sigma : (\lambda x_\sigma : A_\tau)_{\sigma \rightarrow \tau}(x_\sigma) = A_\tau)$$

[67] Actually this is a sort, but the distinction between sorts and types is generally dropped in type theory.

[68] Note that the restrictions of sorts and types mean that sets are homogenous in simple type theory: it is not possible to form sets such as $\{x, \{x\}\}$, since x and $\{x\}$ are of essentially different types.

[69] In fact it is possible to *define* the quantifiers in terms of λ by means of the construction

$$(\forall x : \phi) \stackrel{\text{def}}{=} (\lambda x : \phi) = (\lambda x : \mathit{true}).$$

and since the innermost x_σ is bound by the λ, whereas the others are bound by the $\forall$, we can instantiate the $\forall$-bound instances with a term t_σ to obtain

$$(\lambda x_\sigma : A_\tau)_{\sigma\to\tau}(t_\sigma) = A_\tau[x_\sigma \leftarrow t_\sigma],$$

which is the rule of *β-conversion.*

Church first developed a system for defining and manipulating functions defined by λ-abstraction. His system is called the λ-calculus, and it provides a foundation for the semantics of functional programming languages. In addition to β-conversion, there are two other "conversions" (sometimes also called "reductions") in the λ-calculus.

alpha-conversion: this is the renaming of bound variables—that is,

$$(\lambda x_\sigma : A_\tau(x))_{\sigma\to\tau} = (\lambda y_\sigma : A_\tau(y))_{\sigma\to\tau}.$$

eta reduction: this says that a function equals its own abstraction—that is,

$$(\lambda x_\sigma : f_{\sigma\to\tau}(x))_{\sigma\to\tau} = f_{\sigma\to\tau}.$$

In axiomatic set theory, individuals, sets, sets of sets, and so on all belong to the same universe, and quantification extends over the entire universe. In type theory, the universe is stratified by the type hierarchy, so that quantification applies to each type separately. This means that type theory allows (in fact, requires) quantification over functions and predicates. For this reason, it is also known as *higher-order logic*: propositional calculus allows no quantification and can be regarded as "zero-order" logic, predicate calculus allows quantification over individuals and is also known as "first-order" logic, "second-order" logic allows quantification over functions and predicates of individuals, "third-order" allows quantification over functions and predicates of functions, and so on up to ω-order logic, which allows quantification over arbitrary types. In this scheme, type theory or higher-order logic corresponds to ω-order logic.

In higher-order logic, functions can take other functions as arguments and return functions as values (functions that do so are sometimes called *functionals*, and the term *function* is then reserved for those functions that operate on, and return, simple values). Functions of several arguments can be treated as functionals of a single argument in higher-order logic by a process known as "Currying" (named after the logician H. B. Curry, although it was actually first described by Schönfinkel [Sch67]). To see how this works, suppose we have a function f that takes two arguments and returns the result of adding the first to the square of the second:

$$f \stackrel{\text{def}}{=} (\lambda x_{\mathcal{N}}, y_{\mathcal{N}} : x + y^2)_{\mathcal{N}\times\mathcal{N}\to\mathcal{N}}$$

so that we have $f(2,3) = 11$. We could instead define a functional f' that takes a single argument and returns a *function* that takes another single argument and returns the square of that value, plus the argument to the first functional:

$$f' \stackrel{\text{def}}{=} (\lambda x_{\mathcal{N}} : (\lambda y_{\mathcal{N}} : x + y^2)_{\mathcal{N}\to\mathcal{N}})_{\mathcal{N}\to(\mathcal{N}\to\mathcal{N})}.$$

In this case we have $f'(2) = (\lambda y : 2 + y^2)_{\mathcal{N}\to\mathcal{N}}$, so that $f'(2)(3) = 11$. In general, a function application $g(x, y, \ldots, z)$ is equivalent to a sequence of Curried applications $g'(x)(y)\cdots(z)$. Currying makes it easier to define the logic since it is only necessary to consider functions and predicates of one argument (I am implicitly doing this in the presentation here). Some formulations and mechanizations of higher-order logic (e.g., HOL) *require* all functions to be Curried, whereas others merely allow it. The mandatory use of Curried functions makes for unfamiliar constructions and is a difficult hurdle for many users; however, the freedom to use functionals is very valuable when used appropriately.

Type theory or higher-order logic has the same axioms and rules of inference as first-order logic with equality (generalized to allow quantification over functions and predicates), together with the type-restricted form of the comprehension axiom given earlier, and an axiom of *extensionality*:

Extensionality: $(\forall x_\sigma : f_{\sigma\to\tau}(x) = g_{\sigma\to\tau}(x)) \supset f = g.$

This simply says that two functions (or predicates) are equal if they take the same value as each other for every value of their arguments.

There are two main classes of models for type theory. The details are very technical but a rough idea can be conveyed as follows. Since we can quantify over functions in type theory, we have to ask what we mean when we speak of *all* functions from D to D, say. Under the *standard* interpretation, we really do mean all functions on the domain that interprets D. A formula is valid if it valuates to true in all such interpretations. On the other hand, we might want to interpret *all* as ranging over just some particular class of functions (e.g., the continuous functions). In this latter case, the interpretation is relative to what is variously called a *frame* [Mon76] or a *type structure* [And86] that identifies the particular class of functions concerned. Under the *general models* interpretation of type theory (due to Henkin), a formula is valid if it valuates to true in *all* frames or type structures. Since the standard interpretation is included as one frame among those of the general models interpretation, it is clear that *fewer* formulas are valid under the general models interpretation than under the standard one. The standard interpretation is the more natural one, but the axioms given above are not complete in this case (though they are sound and consistent). The advantage of the general models interpretation is that the axiomatization is complete for this case (it cuts out the valid but unprovable formulas), and several other metamathematical properties can also be established. Furthermore, under the general models interpretation, there is a way to translate or interpret higher-order logic formulas into (sorted) first-order logic.

The logicist's hope was that with type theory as a "self evident" foundation, it would be possible to derive mathematics in a consistent way without the need to introduce additional axioms such as those of Peano, or Zermelo-Fraenkel:

> "What were formerly taken, tacitly or explicitly, as axioms are either unnecessary or demonstrable" [WR27, preface].

Natural numbers can be defined using an idea due to Frege: 3, for example, is the predicate that designates the property of being a triple. I will not give the formal definitions here, since they are tricky, but the idea is that 0 is the property of being the empty set, and the property of being the successor $succ(n)$ to some natural number n is that of being a set that can be formed by adding an element to a set that has the property n.[70] The problem with this approach is that there is nothing that requires the individuals to be infinite in number, and so we cannot guarantee that any large numbers exist. This difficulty is overcome by adding an *axiom of infinity* to the other axioms of type theory. There are several ways to state this axiom, but one way is to assert that there exists some relation $<$ on the individuals that is irreflexive, transitive, and unbounded (i.e., $(\forall x : (\exists y \colon x < y))$). With this foundation, it is possible to derive Peano's axioms as theorems.

As noted earlier, type theory without the axiom of infinity is consistent and (under the general models interpretation) complete. Since the addition of the axiom of infinity allows us to develop arithmetic, it is not surprising, in light of Gödel's results, that the full system (with the axiom of infinity) is incomplete. In terms of power, this form of simple type theory is equivalent to Zermelo set theory (i.e., ZF without the axiom of replacement). For computer science, the consequent loss of the higher infinities is unimportant.

Type theory is notationally complex[71] and has some properties that mathematicians find inconvenient. For example, the empty set is not a simple notion: for

[70] Since sets and properties are equivalent in type theory, another way to say this is than n is the set of all n-element sets. This way of looking at things led to much criticism: surely the set of all triples, conceived as a totality, is "an extravagant affair" [Car58, page 141]. This indicates the significant difference between set theory and type theory. In set theory, sets are conceived as totalities, and to avoid the paradoxes we have to make sure they do not get "too big." In type theory, sets are conceived as determined by their properties, and to avoid the paradoxes we must obey the restrictions of typing.

[71] Especially using the mathematicians' notation; they Curry all functions, drop all punctuation, reverse the order of the type-symbols, and write function application without parentheses—see Andrews [And86], for example.

each type, the empty set of elements of that type is distinct from the empty set of a different element type. This duplication seems perfectly acceptable on linguistic grounds (e.g., is having no money the same as having no worries?); rather less so is the realization that the construction of the natural numbers undertaken above was performed relative to a type (I glossed over this), so that every type has its own version of the natural numbers, and its own arithmetic.[72]

Type theory also lacks some of the interesting metamathematical properties of first-order logic, such as the Compactness, Löwenheim-Skolem, and Interpolation Theorems.[73] This is no accident, for some of the metamathematical properties of first-order logic are unique: Lindstrom's Characterization Theorem says that first-order logic is the "strongest" logic with both a compactness theorem and a (downward) Löwenheim-Skolem Theorem. Since these two theorems are the principal tools for deriving metamathematical results in first-order logic, logicians are reluctant to forsake them, and many therefore regard first-order logic as the only "true logic." Barwise [Bar85] observes that this attitude confuses the subject matter of logic with one of its tools. He likens first-order logic, a tool constructed to help investigate the general conception "logic," to the telescope, a tool constructed to help study the universe. Logicians who assert that logic *is* first-order logic are like those who would claim that astronomy is the study of telescopes.

Although first-order logic has many interesting properties, there are several useful and important notions that cannot be defined within it; for example, none of *infinite* set, *countable* set, *open* set, *continuous* function, *well-ordering*, and *random* variable are first-order definable. These can all be defined in type theory, however. Other concepts, such as induction or Hilbert's ε operator, which require set-theoretic constructions or metalogical assistance from schemes to specify in first-order logic, can also be stated simply and directly in type theory.

For formal methods in computer science, the expressive power of type theory may be more important than the metamathematical richness of first-order logic. Moreover, the complex notation of type theory can be considerably streamlined when used as a specification language for formal methods: computer scientists are used to typed systems in programming languages, and their practice of "declaring" variables and constants before use supplies type information implicitly, without the sub- or superscripts of the mathematicians notation; "overloading" (allowing a single symbol such as + or "emptyset" to denote several similar functions of

[72] Fraenkel, Bar-Hillel, and Levy call the "reduplication" of notions such as numbers and the empty set "repugnant" [FBHL84, page 160].

[73] The first two of these were described earlier; the last, often called Craig's Interpolation Lemma (or Theorem), says that if $\phi(A, B) \supset \psi(B, C)$, where $\phi(A, B)$ denotes a formula built up from names in collections A and B, then there is a formula $\gamma(B)$ (i.e., one built up from such names as are common to both ϕ and ψ) such that $\phi(A, B) \supset \gamma(B) \supset \psi(B, C)$. The Interpolation Theorem is used in the proof of Beth's and Robinson's Theorems.

different types that are distinguished by context) also contributes to simple, familiar notation.[74] The duplication of numbers and arithmetics can be overcome by simply adopting Peano's axioms, so that the (canonical) natural numbers are available as a base sort.[75] The incompleteness of type theory under the standard interpretation renders it unattractive to metamathematicians, but this is of little concern in formal methods, since we will need arithmetic and other incomplete theories in any case.

The great advantages of type theory as a foundation for formal specification languages are the very strong and mechanized "typechecking" that can be provided (this makes for very effective early error-detection in specifications), and the expressive convenience of higher-order constructions and quantification. The fact that everything is built on total functions also allows for rather effective mechanical theorem proving in type theory (recall that in set theory, total functions are a special case).

Formal methods and theorem proving systems based on the simple theory of types include HOL [GM93], PVS [ORS92], and TPS [AINP88].

A.11 Additional Topics

In this section I briefly touch on a number of topics. The aim here is simply to give readers exposure to certain terms they may run across in the literature or in discussions on formal methods. My treatment will be very brief and rather superficial, but should at least point to the relevant literature and help the reader identify the field to which a particular concept or piece of terminology belongs.

A.11.1 Modal Logics

Aristotle and, later, medieval logicians conceived of many *modes* of truth, such as *necessary* truths, and those things that *ought* to be true, or that we *know* to be true, or *believe* to be so.

It is possible to formalize these notions by adding the *modal operators* (also called *modalities*) $\Box$ and $\Diamond$ to the propositional calculus (thereby yielding a propositional modal logic).[76] For example, $\Box A$ can read as "it is necessary that A," in which case $\Diamond A$ is read as "it is possible that A."

---

[74] Russell employed a similar approach, which he called "typical ambiguity": type symbols were usually omitted, and the user was expected to mentally supply them in a manner that provided a well-formed construction.

[75] This would be repugnant to Russell, since he wanted to minimize use of axioms; but for formal methods we are concerned with utility, not philosophical purity.

[76] Mints [Min92] is a good introduction to modal logic.

All modal logics define $\Diamond$ in terms of $\Box$ by the identity

$$\Diamond A \stackrel{\text{def}}{=} \neg\Box\neg A.$$

They also include all the theorems of propositional calculus (generalized to include those with modal operators, such as $\Box A \supset \Box A$), modus ponens, and the inference rule of *necessitation*:

$$\frac{A}{\Box A}.$$

What axiom schemes are added depends on the readings ascribed to the modal operators. There are several axiom schemes with standardized names:

K: $\Box(A \supset B) \supset (\Box A \supset \Box B)$
D: $\Box A \supset \Diamond A$
T: $\Box A \supset A$
4: $\Box A \supset \Box\Box A$
B: $A \supset \Box\Diamond A$
5: $\Diamond A \supset \Box\Diamond A$.

Modal logics are given names such as KT4, which identifies the logic that contains just the axiom schemes K, T, and 4. Some standard modal logics are KD, KT4 (also known as S4), KT5 (also known as S5), and KD45. S5 is historically important as the logic of necessity, but more recently it has become widely used in computer science as a logic of knowledge ($\Box A$ is then read as "it is known that A"). Using this modal logic to reason about "who knows what" has proved a powerful way to analyze distributed systems [HF89]. S4 gives rise to many temporal logics ($\Box A$ is then read as "always A" and $\Diamond A$ as "sometimes" or "eventually" A), which can be used to reason about the properties of concurrent and distributed systems. The modal logic KD45 can be used as a logic of belief ($\Box A$ is then read as "it is believed that A") and has found applications in reasoning about cryptographic key distribution protocols [Ran88]. KD is a "deontic" logic: one for reasoning about obligations.

Leibniz was the first to conceive a recognizably modern semantics for modal logic. He imagined that there could be many worlds: necessary truths would be those that are true in all worlds, possible truths would be those that are true in only some of them. Models for modal logics are based on this idea of a "possible world" semantics (also called "Kripke" semantics after Saul Kripke, who developed the formal treatment while still in his teens). The details are a little too lengthy to go into here, but a key component is an "accessibility" relation between possible worlds; the various standard modal logics can then be characterized by the algebraic properties of the accessibility relation in their models. For example, S5 is characterized by an accessibility relation that is an equivalence relation. These topics are developed in standard texts on modal logics, such as those by Hughes and Cresswell [HC68, HC84], and Chellas [Che80].

As noted, *temporal* logics are modal logics (generally specializations of S4) in which the operators $\Box$ and $\Diamond$ are read as "always" and "eventually," respectively. Pnueli [Pnu77] was the first to recognize that these logics could be used to reason about distributed computations and, in particular, about liveness properties (that is, about things that must happen eventually). There are two families of temporal logics: linear time and branching time [Lam83]. Both families have their adherents, and both have led to effective specification techniques. It is usually necessary to embellish the basic logics of either family with additional operators in order to achieve a comfortable degree of expressiveness; examples include "next state," "until," and backwards-time operators. Interval logics are temporal logics specialized for reasoning over intervals of activity. The Temporal Logic of Actions (TLA) [Lam91] is a temporal logic in which the modal operators are generalized from states to pairs of states (actions); it achieves considerable expressiveness with very little mechanism.

As with predicate calculus, a valid formula in modal logic is one that valuates to true in all interpretations. Many modal logics have what is called the "finite model property," which renders them decidable. The models of temporal logic are essentially finite-state machines; conversely a finite-state machine is a potential model for a temporal logic formula. This observation gives rise to "model checking": the goal is to check whether a finite-state machine describing a system implementation satisfies a desired property specified as a temporal logic formula. This process is equivalent to testing whether the specified machine is a model for the specified formula. Because temporal logic can be quite expressive, and because model-checking is decidable, this technique offers a completely automatic means for verifying certain properties of certain systems [CES86]. Very clever model-checking algorithms allow finite-state machines with large numbers of states to be checked in reasonable time [BCM+90]. Model checking is an example of a state-exploration method (recall Section 2.2.3); it is not a replacement for conventional theorem proving in support of verification (it is applicable to only certain properties and implementations), but it can be a very valuable adjunct.

Despite their name, temporal logics do not provide ways to reason about "time" in a direct or quantitative (i.e., "real-time") sense: they provide a tool for reasoning about the order (i.e., temporal sequencing) of events, and about eventuality properties. Several extensions have been proposed to both temporal [AH89, Koy90] and classical [JM86, Ost90] logics for reasoning about real-time properties, but these are best regarded as promising research, rather than techniques ready for immediate exploitation.

All the modal logics I have considered so far are propositional. It is possible to combine modal operators with quantification to yield first-order modal logics but the details prove rather tricky, as exemplified by the following formula.

$$\Diamond \exists x\colon \phi(x) \supset \exists x\colon \Diamond \phi(x).$$

This is called the Barcan formula (after Ruth Barcan), and it is provable if S5 is combined with first-order logic. The difficulty posed by this formula is that, interpreting $\diamond$ as "possibly," it seems to require that we admit the existence of (or, as logicians and philosophers say, "include in our ontology") merely possible objects. For example, if it is possible that a cow jumped over the moon, then the Barcan formula tells us that there is a specific cow that has possibly performed this act. This ontological commitment is considered objectionable (or, at least, controversial) and so there have been many attempts to combine modal and first-order logics in ways that exclude the Barcan formula (see [Gar84, HC68]).

A.11.2 Logics for Partial Functions

Partial functions are those whose values are not defined for all arguments in their domain: for example, division is not defined when the divisor is zero. The question then is what meaning to ascribe to expressions such as

$$\frac{x}{y} \times y = x \tag{A.3}$$

when it is possible that y could be zero. I briefly mention five main approaches. Cheng and Jones [CJ90] consider a few others but do not, in my opinion, do adequate justice to the third and fifth of the alternatives below. Farmer [Far90] gives a very readable discussion at the beginning of an otherwise very technical paper.

Russell's Theory of Descriptions. In set theory, functions are sets of pairs, and the meaning of $f(x)$ is "the y such that $(x, y) \in f$." The problem is then that of ascribing a meaning to expressions of the form "the such and such" when there is no object with the property "such and such." ("The present king of France" is the standard example.) Expressions of the form "the such and such" are called *definite descriptions* and are intended to single out some unique individual satisfying the description "such and such." How to interpret a definite description when it does not identify a unique individual (either because there is none, or more than one, having the given property) was one to which Russell devoted a great deal of effort in the early 1900s. He used the notation $\iota x: \phi(x)$ for "the x such that $\phi(x)$," where ι is the definite description operator.[77] Russell's solution to the problem of how to deal with $\iota x: \phi(x)$ when this did not identify a unique individual was a contextual one: he gave no meaning to $\iota x: \phi(x)$ considered in isolation, but set the expression in which it occurred to false. Thus, "the present king of France" has no meaning on its own, but "the present king of France is bald" is false in Russell's treatment.

[77] Russell actually used an upside-down ι, but that is a typographical trick that is not so easy to perform in LaTeX as it is with metal type.

Using Russell's operator, the value of $f(x)$ is given by $\iota y{:}\,(x, y) \in f$; this is well-defined if $\exists! y{:}\,(x, y) \in f$, otherwise any expression in which it appears is false. The expression (A.3) is therefore false in this treatment when $y = 0$.

One problem with Russell's approach is that it raises the question of scope: how large an enclosing expression should be set to false when a definite description does not denote a unique individual? Discussion of this question occupies much of Russell's papers on the subject (one written with Whitehead is the most accessible [WR67]). Another problem is that identity is no longer reflexive: $f(x) = f(x)$ is false if x is not in the domain of f. For these reasons, Russell's approach is seldom employed today, but his theory of descriptions has influenced much of the work that followed.

All functions are total. In this scheme, every function has some value assigned for each member of its domain—so that $\frac{x}{0}$, for example, does have a value. We can choose whether or not to supply axioms that enable any properties of such "artificial" values to be deduced. If, for example, we choose not to specify any properties of the value $\frac{x}{0}$, then we will not be able to prove (A.3) as a general theorem, though we will be able to prove the weaker form

$$y \neq 0 \supset \frac{x}{y} \times y = x. \tag{A.4}$$

This approach is quite natural in type theory, where (total) functions are primitive. In set theory, it is equivalent to defining the value of $f(x)$ to be $\varepsilon y{:}\,(x, y) \in f$, where ε is Hilbert's choice operator (recall subsection A.5.4 on page 266).

The arguments against this approach are that it does not correspond to informal mathematical practice (rather than a way to treat partial functions, it is really a way of doing without them), and that we have to be very careful to avoid inadvertently defining (and hence having to implement) properties of the artificial values or, worse, introducing inconsistencies. For example, if we took (A.3) as an axiom, then the case $y = 0$ gives $\frac{x}{0} \times 0 = x$. But another rule of arithmetic is $z \times 0 = 0$, so that the substitution $z \leftarrow \frac{x}{0}$ gives $\frac{x}{0} \times 0 = 0$, which contradicts the earlier result (when x is nonzero).

Functions are total, on a precisely specified domain. This scheme is the same as the previous one, except that it is performed in the context of a logic with a very rich type system, including what are called *predicate* subtypes and *dependent* types, that allow the domains of functions to be specified very precisely.

As its name suggests, a predicate subtype is one associated with a predicate. In the case of division, for example, we can associate the subtype $\mathcal{Q}'$ of the rationals $\mathcal{Q}$ with the predicate $(\lambda q : q \neq 0)$ and can then type the division operation as a total function $\mathcal{Q} \times \mathcal{Q}' \to \mathcal{Q}$. An expression like $\frac{x}{0}$ then has no meaning since it does not satisfy the rules of typing. The expression (A.3) is perfectly valid (and can be taken as an axiom), provided y is of type $\mathcal{Q}'$. More interestingly, the theorem (A.4) is valid, even when y is of type $\mathcal{Q}$ (and could, therefore, be zero), because the value of the expression is independent of the type-incorrect application $\frac{x}{0}$: if $y = 0$, then (A.4) is true because the antecedent to the implication is false (and the value of $\frac{x}{0}$ is irrelevant); otherwise y is of type $\mathcal{Q}'$ and (A.3) applies.

Dependent types are most easily understood by means of an example. Consider the function

$$f(x, y)_{\mathcal{R} \times \mathcal{R} \to \mathcal{R}} \stackrel{\text{def}}{=} \sqrt{x - y}.$$

This function is undefined (on the reals) if its second argument is larger than its first. But if we change the domain of the function from $\mathcal{R} \times \mathcal{R}$ to

$$x\colon \mathcal{R} \times \{y\colon \mathcal{R} \mid x \geq y\} \tag{A.5}$$

(i.e., to pairs of real numbers in which the first is not less than the second), then the partiality disappears: the function is total on this more precisely specified domain. Types such as A.5 are called *dependent* because the type of the second component depends on the *value* of the first. Dependent types, and several other innovations in the treatment of types, had their origins in the pioneering Automath project of de Bruijn [dB80].

The advantages of this approach over the previous one are that it obviates any need to construct artificial values or axioms for expressions such as $\frac{x}{0}$ or *pop*(*empty*), and it disallows (and typechecking will flag) unsafe expressions such as (A.3) when y is of type $\mathcal{Q}$, thereby forcing us, early on, to recognize the need for the guarded form (A.4). Its disadvantage is that typechecking becomes undecidable: theorem proving may be needed to decide the acceptability of formulas. Pragmatically, however, this approach appears quite effective (especially when its mechanization includes a powerful theorem prover), and seems attractively close to informal mathematical practice. Approaches of this kind are used in the Nuprl [CAB+86], PVS [ORS92], and Veritas [HDL89] systems.

Multiple-Valued Logics. If we admit truly partial functions into the logic, then several further choices must be faced. One choice, typified by the "free" logics discussed in the next item, allows terms that have no denotation, but does not allow "undefined" as a value. The alternative introduces a special "undefined" value, often denoted $\perp$ that can be manipulated like an ordinary value.

Once "undefined" is admitted as a value, we have to further decide whether undefinedness is allowed to propagate from the term to the (logical) expression level: for example, does an expression like $f(x) < g(y)$ always yield an ordinary truth value, even if $f(x)$ could be undefined, or could the expression itself be undefined? Attempts to restrict "undefined" to the term level (Russell's treatment attempts to do this) have not been very successful, so it is generally necessary to admit "undefined" into the logic as an additional truth value. The Logic of Partial Functions (LPF) [BCJ84] is of this kind; it provides the logical foundation for the VDM specification notation. A disadvantage of this approach is that all the logical connectives must be extended to the case of undefined expressions and, more importantly, certain properties of traditional logic must be adjusted to retain soundness. In the case of LPF this leads, for example, to the loss of the law of the excluded middle and the deduction theorem.

The main argument against LPF is that it is awkward, nonstandard, and unfamiliar. A consideration of some alternative partial logics, and the identification of one that is free of many of the disadvantages of LPF, has appeared recently [Owe93].

Free Logics. The original motivation for free logics was to avoid certain "paradoxes" that arise when constants are assumed always to refer to objects in the domain of quantification [Ben85]. For example, starting from the identity $K = K$, where K is some constant, we can obtain the theorem $(\exists x : x = K)$ by existential generalization. If K abbreviates "The King of France," then we have just proved that there is a King of France! Free logics avoid this anomaly by introducing a predicate E called the existence predicate ($E(a)$ is interpreted as "a exists") and restricting quantifiers to range only over existing objects. The quantifier rules are modified so that we can deduce $(\exists x : x = K)$ from $K = K$ only if we have already established $E(K)$—which is tantamount to what we are trying to prove.

Logics to deal with partial functions have been proposed along these lines [TvD88, volume I, chapter 2, section 2]. The challenge is to construct a logic that is as faithful to informal mathematical practice as possible. Scott's formulation [Sco79], for example, has the disadvantage that $(\forall x : \phi(x))$ is no longer equivalent to $\phi(x)$. Beeson introduced an alternative system that overcomes most objections [Bee86] (more accessible references are [Bee85, Chapter 6, Section 1] and [Bee88, Section 5]).

Whereas Scott's is a logic of partial *existence* (it allows models with objects that do not satisfy the existence predicate), Beeson's is a logic of partial terms (LPT) that is concerned with the *definedness* of names and terms. LPT introduces the atomic formula $a\!\downarrow$, where a is a term, with the interpretation "a is defined." The quantifier axioms become

A4′. $(\forall x : \phi(x)) \wedge t\!\downarrow \supset \phi[x \leftarrow t]$,

A5′. $\phi[x \leftarrow t] \wedge t\!\downarrow \supset (\exists x : \phi(x))$.

In addition, we have $c\!\downarrow$ and $x\!\downarrow$ for each constant symbol c and variable x. Atomic formulas evaluate to true only if their arguments are defined:

$$\phi(s, t, \ldots, u) \supset s\!\downarrow \wedge\, t\!\downarrow \wedge \cdots \wedge u\!\downarrow, \tag{A.6}$$

where $s, t, \ldots, u$ are metavariables representing terms. Two notions of equality are used: $=$ is strict equality (false if either of its arguments is undefined), and $\simeq$ is "Kleene equality," which is true if both its arguments are undefined:

$$s \simeq t \stackrel{\text{def}}{=} (s\!\downarrow \vee\, t\!\downarrow) \supset s = t.$$

The equality axioms are

$$x = x \wedge (x = y \supset y = x),$$

$$s \simeq t \wedge \phi(s) \supset \phi(t), \text{ and}$$

$$s = t \supset s\!\downarrow \wedge\, t\!\downarrow .$$

(Here x and y are variables). The last of these is a special case of (A.6), as is

$$f(s, t, \ldots, u)\!\downarrow \supset s\!\downarrow \wedge\, t\!\downarrow \wedge \cdots \wedge u\!\downarrow$$

(i.e., functions are defined only if their arguments are).

In the case of the example (A.3), we will presumably have $\frac{x}{y}\!\downarrow \equiv y \neq 0$, so that $(\frac{x}{y} \times y)\!\downarrow \equiv y \neq 0$. Thus, the left hand side of (A.3) is defined (and equal to x) provided $x \neq 0$, and the right hand side is always defined (i.e., $x\!\downarrow$). An equation is false if either of its arguments is undefined, so (A.3) is always defined, and is true if $y \neq 0$, and false otherwise. The form (A.4) is unconditionally true.

PX [HN88] is a computational logic based on these ideas, while a (higher-order) logic of this kind is mechanized in the IMPS system [Far90, FGT93], and variants have been proposed for other specification languages [MR91]. Gumb [Gum89, Chapter 5] uses a free logic to express facts about execution-time errors in programs, and Parnas presents a "logic for software engineering" [Par93] that uses similar ideas.

All these approaches to partial functions, except the second, generally require the user to prove subsidiary theorems in order to discharge definedness obligations. The third approach requires these subsidiary proofs to be performed during typechecking, the first, fourth and fifth require them during proof construction. Doing the subsidiary proofs at typecheck time avoids some duplication of effort (the other approaches can require the same subsidiary results to be proved separately in proofs of different theorems), and allows some faults to be detected earlier. In addition to their treatment of partial functions, the rich type systems of the third approach allow terms to be typed more precisely, and thereby increase the information present in a specification [HD92].

A.11.3 Constructive Systems

The earliest conceptions of mathematical existence were constructive: to Euclid, proving that through every point there exists a line parallel to another given line meant that there was a *procedure* for constructing that line (with ruler and compass). There were some early proofs by contradiction (for example, of the irrationality of $\sqrt{2}$), but these dealt with *non*existence or impossibility, rather than existence. More troubling are proofs of *existence* by contradiction. For example, there exist two irrational numbers x and y such that x^y is rational: if ${\sqrt{2}}^{\sqrt{2}}$ is rational, we are done; otherwise take this number as x and let y be $\sqrt{2}$. This is surely a little unsatisfactory, as we do not know which of the two cases is true.

During the 19th and early 20th centuries, the advent and rapid application of set theory led to many nonconstructive proofs of this kind.[78] One of the most controversial was Zermelo's nonconstructive 1905 proof of the well ordering of the reals. Several mathematicians were dissatisfied with such proofs, but it was Brouwer who launched a full scale assault on the admissibility of nonconstructive methods:

> "the only possible foundation of mathematics must be sought...under the obligation carefully to watch which constructions intuition allows and which not...any other attempt at such a foundation is doomed to failure" [from Brouwer's PhD Thesis, quoted in [Bee85, page 430]].

Brouwer's position became known as *intuitionism*, and although he opposed several nonconstructive techniques and concepts (for example, the completed infinite) it was his criticism of the "law" of the excluded middle [Bro67] (i.e., $\phi \vee \neg\phi$, as used about x^y above) that became the *sine qua non* of intuitionistic mathematics.

[78] The Historical Appendix to Beeson's book [Bee85] is a good, brief source for those who would like to read more of the history of these topics. Beeson's book and the two-volume work of Troelstra and van Dalen [TvD88] are standard texts on constructive mathematics.

Although Brouwer was not particularly interested in creating a formal system for intuitionism (after all, he rejected the axiomatic method as a foundation), Kolmogorov, Heyting and others developed intuitionistic propositional and predicate calculi that do without the law of the excluded middle; in this context, "ordinary" logical systems that retain the excluded middle are called *classical* logics.

Both Brouwer and the opposing camp (the formalists, led by Hilbert) thought that strict adherence to intuitionistic principles would force the abandonment of large tracts of mathematics (by logicians at least; "practical" mathematics would proceed undisturbed by such arcane disputes). The intuitionists seemed to accept this, but the formalists saw it as a challenge to the relevance and utility of foundational studies (for example, Ramsey writes of "the Bolshevik menace of Brouwer and Weyl" [Ram90, page 229]) and were inspired to shore up their foundations: Hilbert declared

> "Wherever there is any hope of salvage, we will carefully investigate fruitful definitions and deductive methods. We will nurse them, strengthen them, and make them useful. No one shall drive us out of the paradise which Cantor has created for us" [Hil83].

However, Gödel later showed (via what is called the double-negation interpretation) that intuitionistic and classical arithmetic have the same strength [NS93, Section V.6],[79] and Bishop provided a practical demonstration that much of analysis could be developed in an entirely constructive (though not specifically intuitionistic) manner [BB85]. Thus, restriction to constructive methods does *not* seem to limit to an unacceptable degree the amount of mathematics that actually can be done.

What does all this have to do with computers? Well, anything that can be done by a computer is necessarily constructive; thus, if we seek a logical foundation for computing (as opposed to general mathematics) it seems natural to seek it within a constructive framework. Intuitionistic logic is a particularly strong candidate because of what is known as the "Curry-Howard isomorphism," which establishes a direct correspondence between intuitionistic proofs and functional programs:[80] an intuitionistic existence proof can be converted into a program that constructs the thing concerned. An idea of the flavor of this approach can be obtained as follows. In intuitionism, a proof that A implies B means a procedure which will take any (intuitionistic) proof of A and convert it into a proof of B; thus, if we have a program that constructs A, the proof of $A \supset B$ gives us a program that constructs B.

[79] Gödel also demonstrated a connection between modal and intuitionistic logics [KK62, page 680]. ([KK62] is a standard, and accessible, text on the history and development of logic.)

[80] This really goes back to Kleene, who showed that any formula that could be defined in intuitionistic (Heyting) arithmetic was Turing computable; thus, Heyting arithmetic could serve as a high-level programming language.

As with classical systems, intuitionistic logics that are adequate for actually specifying and reasoning about interesting computations require more than just the intuitionistic fragment of predicate calculus: we need arithmetic, sets, and/or types. Constructive type theories, especially those built on ideas first widely canvassed by Martin-Löf, dominate this enterprise (often under the slogan "propositions as types"). Martin-Löf's theories are *not* theories in ordinary predicate calculus, so they cannot be described by simply listing their axioms: the entire apparatus of the system has to be developed, and I do not have space to attempt this here. Interested readers can find a good introduction in [Tho91].

Constructive type theories are the foundation for a number of interesting systems for formal specifications and proofs, of which the oldest and best known is Nuprl [CAB+86] (which is based on, and extends, Martin-Löf's approach), and one of the most recent is COQ [DFH+91] (which mechanizes the "Calculus of Constructions," which itself is derived from a system variously known as the "second-order" or "polymorphic" λ-calculus and as "System-F.") Arguments in favor of these approaches are that they bring programming and proof into intimate correspondence. Arguments against them are that their foundation is unfamiliar to most users, and that the need to constantly discharge existence obligations is tedious. However, very similar proof obligations arise in classical logics with rich type systems when trying to establish admissibility of definitional forms or consistency of axiomatizations. Hence, it may be that the actual practice of formal methods in constructive and classical settings is more similar than might at first appear.

A.11.4 Programming Logics

Functional programs can be specified quite directly in any logic that supports recursive definitions; reasoning about such programs can then be performed directly within the logic concerned. Imperative programs, on the other hand, operate on an implicit "state" that has no counterpart in ordinary logic. For example, the Fortran assignment statement `x = x+1` must be interpreted as saying that the value of `x` in the "new state" is to be equal to `1` plus the value of `x` in the "old state." *Programming logics* are logics that provide ways to specify and reason about imperative programs and their effects on a program state.

Hoare [Hoa69] introduced the notation $\{P\}S\{Q\}$ (often called a *Hoare sentence*) for specifying the behavior of a program fragment S. In this construction, P and Q are expressions involving the "variables" of the program S and the intuitive interpretation is that if execution of S starts in a state in which the expression P is true, and if execution terminates, then it will do so in a state in which Q is true. A typical programming language logic will include the following axiom for the *skip* (do nothing) statement

$$\{P\}\ \mathbf{skip}\ \{P\}$$

and the following for the assignment statement

$$\{P[v \leftarrow e]\}\, \texttt{v := e}\, \{P\},$$

where `v` is a "program variable" and `e` is an arithmetic expression, together with axioms or rules of inference for each of the other constructs in the programming language concerned. Standard rules include the following ones for sequential composition, and for the *conditional* and *while* statements, respectively:

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

$$\frac{\{P \wedge B\}S_1\{Q\} \wedge \{P \wedge \neg B\}S_2\{Q\}}{\{P\}\,\textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{endif}\,\{Q\}},$$

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

In addition, there will be some rules concerning the interaction between the programming logic and the classical logic that underlies it, such as the following rule of *precondition strengthening*:

$$\frac{(P' \supset P) \wedge \{P\}S\{Q\}}{\{P'\}S\{Q\}}.$$

As an example, let us prove that the statement

$$\textbf{if}\ \texttt{x<0}\ \textbf{then}\ \texttt{x := -x}\ \textbf{else}\ \ \textbf{skip}\ \ \textbf{endif}$$

causes the final value of x to be the absolute value of its initial value. We can specify this requirement by

$$\{x = X\}\,\textbf{if}\ \texttt{x<0}\ \textbf{then}\ \texttt{x := -x}\ \textbf{else}\ \ \textbf{skip}\ \ \textbf{endif}\,\{x = |X|\},$$

where $X$is a logical variable (implicitly universally quantified over the whole Hoare sentence). By the rule for conditional statements, we need to prove

$$\{x = X \wedge x < 0\}\,\texttt{x := -x}\,\{x = |X|\}$$

and

$$\{x = X \wedge \neg x < 0\}\ \textbf{skip}\ \{x = |X|\}.$$

By precondition strengthening, the second of these follows from

$$(x = X \wedge \neg x < 0 \supset x = |X|) \wedge \{x = |X|\}\ \textbf{skip}\ \{x = |X|\},$$

whose first conjuct follows by predicate calculus, arithmetic, and the definition of the absolute value function, and whose second conjuct is an instance of the *skip* rule given earlier. The other case of the conditional follows by precondition strengthening from

$$(x = X \wedge x < 0 \supset -x = |X|) \wedge \{-x = |X|\}\, \texttt{x := -x}\, \{x = |X|\},$$

whose first conjuct follows by predicate calculus, arithmetic, and the definition of the absolute value function as before, and whose second conjuct is an instance of the *assignment* rule given earlier.

A programming logic can be cast into the more familiar form of a classical logic by making explicit the program state that is implicit in the axioms and rules of the programming logic. In this interpretation, the precondition P and postcondition Q of a Hoare sentence $\{P\}S\{Q\}$ are regarded as classical predicates over the "before" and "after" states s and t, respectively, and the program fragment S is regarded as a shorthand for its denotation or "meaning" $[\![S]\!]$, which is a relation on states: $[\![S]\!](s,t)$ is true if execution of the program fragment S, when started in state s, can terminate in state t. Then the interpretation of the Hoare sentence $\{P\}S\{Q\}$ is

$$\forall s,t : P(s) \wedge [\![S]\!](s,t) \supset Q(t).$$

In this interpretation, the rule of precondition strengthening becomes

$$\frac{(\forall s : P'(s) \supset P(s)) \wedge (\forall s,t : P(s) \wedge [\![S]\!](s,t) \supset Q(t))}{(\forall s,t : P'(s) \wedge [\![S]\!](s,t) \supset Q(t))}$$

which is classically valid.

Similarly, the rule for sequential composition becomes:

$$\frac{(\forall s,t : P(s) \wedge [\![S_1]\!](s,t) \supset Q(t)) \wedge (\forall s,t : Q(s) \wedge [\![S_2]\!](s,t) \supset R(t))}{(\forall s,t : P(s) \wedge [\![S_1;S_2]\!](s,t) \supset R(t))}.$$

I have not gone into enough detail, nor developed enough notation, to give the interpretation of the axiom for assignment in the general case (see Gordon [Gor89]), but the general idea can be conveyed by considering the particular example

$$\{-x = |X|\}\, \texttt{x := -x}\, \{x = |X|\}$$

encountered earlier. The interpretation of an integer program "variable" such as x is a function from state to integer (so that $x(s)$ is its value in state s), and the interpretation of this Hoare sentence is then

$$(\forall s,t,X : -x(s) = X \wedge [\![\texttt{x := -x}]\!](s,t) \supset x(t) = |X|).$$

The End

That concludes this rapid introduction to those aspects of mathematical logic that I consider most relevant to formal methods. Experts will notice that I have simplified or ignored some details—but not, I hope, to the point of inaccuracy. I recommend those who wish to read further to consult initially the books noted near the beginning of the section of this appendix that deals with the topic of interest to them.

Bibliography

[Acz88] Peter Aczel. *Non-Well-Founded Sets*, volume 14 of *CSLI Lecture Notes*. Center for the Study of Language and Information, Stanford, CA, 1988.

[AH89] R. Alur and T. A. Henzinger. A really temporal logic. In *30th IEEE Symposium on Foundations of Computer Science*, pages 164–169, 1989.

[AINP88] Peter B. Andrews, Sunil Issar, Daniel Nesmith, and Frank Pfenning. The TPS theorem proving system. In Lusk and Overbeek [LO88], pages 760–761.

[AK88] William Aspray and Philip Kitcher, editors. *History and Philosophy of Modern Mathematics*, volume XI of *Minnesota Studies in the Philosophy of Science*. University of Minnesota Press, Minneapolis, MN, 1988. Outgrowth of a conference held in 1985.

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.

[And86] Peter B. Andrews. *An Introduction to Logic and Type Theory: To Truth through Proof*. Academic Press, New York, NY, 1986.

[Bar78a] Jon Barwise, editor. *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, Holland, 1978.

[Bar78b] Jon Barwise. An introduction to first-order logic. In *Handbook of Mathematical Logic* [Bar78a], chapter A1, pages 5–46.

[Bar85] J. Barwise. Model-theoretic logics: Background and aims. In J. Barwise and S. Feferman, editors, *Model-Theoretic Logics*, Perspectives in Mathematical Logic, chapter 1, pages 3–23. Springer-Verlag, New York, NY, 1985.

[BB85] E. Bishop and D. Bridges. *Constructive Analysis*. Springer-Verlag, Berlin, Germany, 1985.

[BCJ84] H. Barringer, J. H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[BCM+90] J. R. Burch, E. M. Clarke, K. L McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, PA, June 1990. IEEE Computer Society.

[Bee85] Michael J. Beeson. *Foundations of Constructive Mathematics*. Ergebnisse der Mathematik und ihrer Grenzgebiete; 3. Folge · Band 6. Springer-Verlag, 1985.

[Bee86] Michael J. Beeson. Proving programs and programming proofs. In *International Congress on Logic, Methodology and Philosophy of Science VII*, pages 51–82, Amsterdam, 1986. North-Holland. Proceedings of a meeting held at Salzburg, Austria, in July, 1983.

[Bee88] Michael J. Beeson. Towards a computation system based on set theory. *Theoretical Computer Science*, 60:297–340, 1988.

[Ben85] Ermanno Bencivenga. Free logics. In Gabbay and Guenthner [GG85], chapter III.6, pages 373–426.

[BJ89] George S. Boolos and Richard C. Jeffrey. *Computability and Logic*. Cambridge University Press, Cambridge, UK, third edition, 1989.

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, NY, 1979.

[BM88] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, New York, NY, 1988.

[Bou68] N. Bourbaki. *Elements of Mathematics: Theory of Sets*. Addison-Wesley, Reading, MA, 1968.

[BP83] Paul Benacerraf and Hilary Putnam, editors. *Philosophy of Mathematics: Selected Readings*. Cambridge University Press, Cambridge, England, second edition, 1983.

[Bro67] Luitzen Egbertus Jan Brouwer. On the significance of the principle of excluded middle in mathematics, especially in function theory. In van Heijenoort [vH67], pages 334–345. First published 1923.

[Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[BS89] G. Birtwistle and P. A. Subrahmanyam, editors. *Current Trends in Hardware Verification and Theorem Proving.* Springer-Verlag, New York, NY, 1989.

[CAB+86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Can55] Georg Cantor. *Contributions to the Founding of the Theory of Transfinite Numbers.* Dover Publications, Inc., 1955. Translated and with an 80 page introduction by Philip E. B. Jourdain. Originally published as *Beiträge zur Begründung der Transfiniten Mengenlehre*, 1895 and 1897.

[Can91] D. Cantone. Decision procedures for elementary sublanguages of set theory X: Multilevel syllogistic extended by the singleton and powerset operators. *Journal of Automated Reasoning*, 7(2):193–230, June 1991.

[Car58] Rudolf Carnap. *Introduction to Symbolic Logic and Its Applications.* Dover Publications, Inc., New York, NY, 1958. English translation of *Einführung in die symbolische Logik*, 1954.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[Che80] Brian F. Chellas. *Modal Logic: An Introduction.* Cambridge University Press, Cambridge, UK, 1980.

[Chu56] Alonzo Church. *Introduction to Mathematical Logic*, volume 1 of *Princeton Mathematical Series.* Princeton University Press, Princeton, NJ, 1956. Volume 2 never appeared.

[CJ90] J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In Carroll Morgan and J. C. P. Woodcock, editors, *Proceedings of the Third Refinement Workshop*, pages 51–69. Springer-Verlag Workshops in Computing, 1990.

[CKM+91] Dan Craigen, Sentot Kromodimoeljo, Irwin Meisels, Bill Pase, and Mark Saaltink. EVES: An overview. In Prehn and Toetenel [PT91], pages 389–405.

[CL73] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving.* Computer Science and Applied mathematics. Academic Press, New York, NY, 1973.

[Cop67] Irving M. Copi. *Symbolic Logic.* The Macmillan Company, third edition, 1967.

[Cur77] Haskell B. Curry. *Foundations of Mathematical Logic.* Dover Publications, Inc., New York, NY, 1977.

[Dau88] Joseph W. Dauben. Abraham Robinson and nonstandard analysis: History, philosophy and foundations of mathematics. In Aspray and Kitcher [AK88], pages 177–200.

[Dav89] Ruth E. Davis. *Truth, Deduction, and Computation.* Principles of Computer Science Series. Computer Science Press, an imprint of W. H. Freeman and Company, New York, 1989.

[dB80] N. G. de Bruijn. A survey of the project Automath. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 589–606. Academic Press, 1980.

[Ded63] Richard Dedekind. *Essays on the Theory of Numbers.* Dover Publications, Inc., New York, NY, 1963. Reprint of the 1901 Translation by Wooster Woodruff Beman; the original essay was written 1887.

[DeL70] Howard DeLong. *A Profile of Mathematical Logic.* Addison-Wesley series in mathematics. Addison-Wesley, Reading, MA, 1970.

[DFH+91] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Christine Paulin-Mohring, and Benjamin Werner. The COQ proof assistant user's guide: Version 5.6. Rapports Techniques 134, INRIA, Rocquencourt, France, December 1991.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 6, pages 243–320. Elsevier and MIT press, Amsterdam, The Netherlands, and Cambridge, MA, 1990.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics.* Texts and Monographs in Computer Science. Springer-Verlag, New York, NY, 1990.

[DST80] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpressions problem. *Journal of the ACM*, 27(4):758–771, October 1980.

[Dun85] J. Michael Dunn. Relevance logic and entailment. In Gabbay and Guenthner [GG85], chapter III.3, pages 117–224.

[Dun91] William Dunham. *Journey Through Genius: The Great Theorems of Mathematics.* Penguin Books, New York, NY, 1991.

[EFT84] H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic.* Undergraduate Texts in Mathematics. Springer-Verlag, New York, NY, 1984.

[End72] H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, New York, NY, 1972.

[Far90] William M. Farmer. A partial functions version of Church's simple theory of types. *Journal of Symbolic Logic*, 55(3):1269–1291, September 1990.

[FBHL84] A. A. Fraenkel, Y. Bar-Hillel, and A. Levy. *Foundations of Set Theory*, volume 67 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, Amsterdam, The Netherlands, second printing, second edition, 1984.

[Fef89] Solomon Feferman. *The Number Systems: Foundations of Algebra and Analysis.* Chelsea Publishing Company, New York, NY, second edition, 1989. First edition published 1964.

[FGT93] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. *Journal of Automated Reasoning*, 11(2):213–248, October 1993.

[Fre67] G. Frege. Letter to Russell. In van Heijenoort [vH67], pages 126–128. Written 1902.

[Gal86] Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving.* Harper and Row, New York, 1986.

[Gar84] James W. Garson. Quantification in modal logic. In Dov M. Gabbay and Franz Guenthner, editors, *Handbook of Philosophical Logic–Volume II: Extensions of Classical Logic*, volume 165 of *Synthese Library*, chapter II.5, pages 249–307. D. Reidel Publishing Company, Dordrecht, Holland, 1984.

[GG83] Dov M. Gabbay and Franz Guenthner, editors. *Handbook of Philosophical Logic–Volume I: Elements of Classical Logic*, volume 164 of *Synthese Library*. D. Reidel Publishing Company, Dordrecht, Holland, 1983.

[GG85] Dov M. Gabbay and Franz Guenthner, editors. *Handbook of Philosophical Logic–Volume III: Alternatives to Classical Logic*, volume 166 of *Synthese Library*. D. Reidel Publishing Company, Dordrecht, Holland, 1985.

[GH93] John V. Guttag and James J. Horning with S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, UK, 1993.

[Gog89] J. A. Goguen. OBJ as a theorem prover with application to hardware verification. In Birtwistle and Subrahmanyam [BS89], pages 218–267.

[Gol88] Warren Goldfarb. Poincaré against the Logicists. In Aspray and Kitcher [AK88], pages 61–81.

[Gor88] Michael J. C. Gordon. *Programming Language Theory and its Implementation*. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, 1988.

[Gor89] Michael J. C. Gordon. Mechanizing programming logics in higher-order logic. In Birtwistle and Subrahmanyam [BS89], pages 387–439.

[GTWW77] Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24(1):68–95, January 1977.

[Gum89] Raymond D. Gumb. *Programming Logics: An Introduction to Verification and Semantics*. John Wiley and Sons, New York, NY, 1989.

[Hal60] Paul R. Halmos. *Naive Set Theory.* The University Series in Undergraduate Mathematics. Van Nostrand Reinhold Company, New York, NY, 1960.

[Hal84] Michael Hallett. *Cantorian Set Theory and Limitation of Size.* Number 10 in Oxford Logic Guides. Oxford University Press, Oxford, England, 1984.

[Hat82] William S. Hatcher. *The Logical Foundations of Mathematics.* Pergamon Press, Oxford, UK, 1982.

[Haz83] Allen Hazen. Predicative logics. In Gabbay and Guenthner [GG83], chapter I.5, pages 331–407.

[HC68] G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logic.* Methuen, London, UK, 1968.

[HC84] G. E. Hughes and M. J. Cresswell. *A Companion to Modal Logic.* Methuen, London, UK, 1984.

[HD92] F. K. Hanna and N. Daeche. Dependent types and formal synthesis. In C. A. R. Hoare and M. J. C. Gordon, editors, *Mechanized Reasoning and Hardware Design*, pages 121–135, Hemel Hempstead, UK, 1992. Prentice Hall International Series in Computer Science.

[HDL89] F. Keith Hanna, Neil Daeche, and Mark Longley. Specification and verification using dependent types. *IEEE Transactions on Software Engineering*, 16(9):949–964, September 1989.

[HF89] Joseph Y. Halpern and Ronald Fagin. Modeling knowledge and action in distributed systems. *Distributed Computing*, 3:159–177, 1989.

[Hil83] David Hilbert. On the infinite. In Benacerraf and Putnam [BP83], pages 183–201. First published 1926.

[HN88] Susumu Hayashi and Hiroshi Nakano. *PX: A Computational Logic.* Foundations of Computing. MIT Press, Cambridge, MA, 1988.

[Hoa69] C. A. R. Hoare. An axiomatic basis of computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[Hod77] Wilfred Hodges. *Logic.* Penguin, London, UK, 1977.

[Hod83] Wilfred Hodges. Elementary predicate logic. In Gabbay and Guenthner [GG83], chapter I.1, pages 1–131.

[JM86] Farnam Jahanian and Aloysius Ka-Lau Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–904, September 1986.

[Joh87] P. T. Johnstone. *Notes on Logic and Set Theory.* Cambridge Mathematical Textbooks. Cambridge University Press, Cambridge, UK, 1987.

[KK62] William Kneale and Martha Kneale. *The Development of Logic.* Clarendon Press, Oxford, England, 1962. Paperback edition, 1984.

[Kle52] Stephen C. Kleene. *Introduction to Metamathematics.* North-Holland, Amsterdam, The Netherlands, 1952.

[Kle67] Stephen C. Kleene. *Mathematical Logic.* John Wiley and Sons, New York, NY, 1967.

[Koy90] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, November 1990.

[KZ88] D. Kapur and H. Zhang. RRL: A rewrite rule laboratory. In Lusk and Overbeek [LO88], pages 768–769.

[Lak78] Imre Lakatos. Cauchy and the continuum: the significance of non-standard analysis for the history of philosophy of mathematics. In John Worrall and Gregory Currie, editors, *Mathematics, Science and Epistemology: Philosophical Papers of Imre Lakatos, Volume 2*, chapter 3, pages 43–60. Cambridge University Press, Cambridge, UK, 1978. (Cambridge Paperback Library edition, 1990).

[Lam83] L. Lamport. Sometime is sometimes not never. In *10th ACM Symposium on Principles of Programming Languages*, pages 174–185, Austin, TX, January 1983.

[Lam91] Leslie Lamport. The temporal logic of actions. Technical Report 79, DEC Systems Research Center, Palo Alto, CA, December 1991. To Appear in ACM Transactions on Programming Languages and Systems.

[Lei69] A. C. Leisenring. *Mathematical Logic and Hilbert's ε-Symbol.* Gordon and Breach Science Publishers, New York, NY, 1969.

[Lem87] E. J. Lemmon. *Beginning Logic.* Chapman and Hall, London, UK, second edition, 1987.

[Lev79] Azriel Levy. *Basic Set Theory.* Perspectives in Mathematical Logic. Springer-Verlag, Berlin, Germany, 1979.

[LO88] E. Lusk and R. Overbeek, editors. *9th International Conference on Automated Deduction (CADE)*, volume 310 of *Lecture Notes in Computer Science*, Argonne, IL, May 1988. Springer-Verlag.

[McA89] David A. McAllester. *ONTIC: A Knowledge Representation System for Mathematics*. MIT Press, Cambridge, MA, 1989.

[McC90] W. W. McCune. OTTER 2.0 users guide. Technical Report ANL-90/9, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, March 1990.

[Men64] Elliott Mendelson. *Introduction to Mathematical Logic*. The University Series in Undergraduate Mathematics. D. Van Nostrand Company, New York, NY, 1964.

[Min92] Grigori Mints. *A Short Introduction to Modal Logic*, volume 30 of *CSLI Lecture Notes*. Center for the Study of Language and Information, Stanford, CA, 1992.

[Mon76] J. Donald Monk. *Mathematical Logic*. Graduate Texts in Mathematics. Springer-Verlag, New York, NY, 1976.

[Moo88] Gregory H. Moore. The emergence of first-order logic. In Aspray and Kitcher [AK88], pages 95–135.

[MR91] C. A. Middelburg and G. R. Renardel de Lavalette. LPF and MPL_ω—a logical comparison of VDM SL and COLD-K. In Prehn and Toetenel [PT91], pages 279–308.

[Mus80] D. R. Musser. Abstract data type specification in the Affirm system. *IEEE Transactions on Software Engineering*, 6(1):24–32, January 1980.

[NO79] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[NS93] Anil Nerode and Richard A. Shore. *Logic for Applications*. Texts and Monographs in Computer Science. Springer-Verlag, New York, NY, 1993.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, pages 748–752, Saratoga, NY, June 1992. Volume 607 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag.

[Ost90] Jonathan S. Ostroff. A logic for real-time discrete event processes. *IEEE Control Systems Magazine*, 10(4):95–102, June 1990.

[Owe93] Olaf Owe. Partial logics reconsidered: A conservative approach. *Formal Aspects of Computing*, 5(3):208–223, 1993.

[Par93] David Lorge Parnas. Predicate logic for software engineering. *IEEE Transactions on Software Engineering*, 19(9):856–862, September 1993.

[Pau88] Lawrence C. Paulson. Isabelle: The next seven hundred theorem provers. In Lusk and Overbeek [LO88], pages 772–773.

[Pea67] Giuseppe Peano. The principles of arithmetic, presented by a new method. In van Heijenoort [vH67], pages 83–97. First published 1889.

[PH78] Jeff Paris and Leo Harrington. A mathematical incompleteness in Peano arithmetic. In Barwise [Bar78a], chapter D8, pages 1133–1142.

[Phi92] I. C. C. Phillips. Recursion theory. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science; Volume 1 Background: Mathematical Structures*, pages 79–187. Oxford Science Publications, Oxford, UK, 1992.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th Symposium on Foundations of Computer Science*, pages 46–57, Providence, RI, November 1977. ACM.

[Pra92] Sanjiva Prasad. Verification of numerical programs using Penelope/Ariel. In *COMPASS '92 (Proceedings of the Seventh Annual Conference on Computer Assurance)*, pages 11–24, Gaithersburg, MD, June 1992. IEEE Washington Section.

[PT91] S. Prehn and W. J. Toetenel, editors. *VDM '91: Formal Software Development Methods*, volume 551 of *Lecture Notes in Computer Science*, Noordwijkerhout, The Netherlands, October 1991. Springer-Verlag. Volume 1: Conference Contributions.

[Rab78] Michael O. Rabin. Decidable theories. In Barwise [Bar78a], chapter C8, pages 595–629.

[Ram90] F. P. Ramsey. The foundations of mathematics. In D. H. Mellor, editor, *Philosophical Papers of F. P. Ramsey*, chapter 8, pages 164–224. Cambridge University Press, Cambridge, UK, 1990. Originally published in *Proceedings of the London Mathematical Society*, 25, pp. 338–384, 1925.

[Ran88] P. Venkat Rangan. An axiomatic basis for trust in distributed systems. In *Proceedings of the Symposium on Security and Privacy*, pages 204–211, Oakland, CA, April 1988. IEEE Computer Society.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Rob74] A. Robinson. *Nonstandard Analysis*. North-Holland, Amsterdam, The Netherlands, revised edition, 1974.

[RR63] H. Rubin and J. Rubin. *Equivalents of the Axiom of Choice*. North-Holland, 1963.

[Rud92] Piotr Rudnicki. *An Overview of the MIZAR Project*. Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, June 1992.

[Rus67] Bertrand Russell. Letter to Frege. In van Heijenoort [vH67], pages 124–125. Written 1902.

[Sch67] Moses Schönfinkel. On the building blocks of mathematical logic. In van Heijenoort [vH67], pages 355–366. First published 1924.

[Sco79] D. S. Scott. Identity and existence in intuitionistic logic. In M. P. Fourman, C. J. Mulvey, and D. S. Scott, editors, *Applications of Sheaves*, pages 660–696. Volume 753 of *Lecture Notes in Mathematics*, Springer-Verlag, 1979.

[Sha94] N. Shankar. *Metamathematics, Machines, and Gödel's Proof*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, 1994.

[Sho67] Joseph R. Shoenfield. *Mathematical Logic*. Addison-Wesley, Reading, MA, 1967.

[Sho77] Robert E. Shostak. On the SUP-INF method for proving Presburger formulas. *Journal of the ACM*, 24(4):529–543, October 1977.

[Sho78a] J. R. Shoenfield. Axioms of set theory. In Barwise [Bar78a], chapter B1, pages 321–344.

[Sho78b] Robert E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7):583–585, July 1978.

[Sho79] Robert E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, April 1979.

[Sho84] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[Spi88] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge Tracts in Theoretical Computer Science 3. Cambridge University Press, Cambridge, UK, 1988.

[Sti81] Mark Stickel. A unification algorithm for associative-commutative functions. *Journal of the ACM*, 28(3):423–434, July 1981.

[Sti85] Mark E. Stickel. Automated deduction by theory resolution. *Journal of Automated Reasoning*, 1(4):333–355, December 1985.

[Sun83] Göran Sundholm. Systems of deduction. In Gabbay and Guenthner [GG83], chapter I.2, pages 133–188.

[Sup72] Patrick Suppes. *Axiomatic Set Theory*. Dover Publications, Inc., New York, NY, 1972.

[Tar76] Alfred Tarski. *Introduction to Logic and to the Methodology of Deductive Sciences*. Oxford University Press, New York, NY, third edition, 1976. First published 1941.

[Tho91] Simon Thompson. *Type Theory and Functional Programming*. International Computer Science Series. Addison-Wesley, Wokingham, England, 1991.

[TvD88] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics: An Introduction*, volume 121 and volume 123 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, Holland, 1988. In two volumes.

[US93] Thomás E. Uribe and Mark E. Stickel. *Ordered Binary Decision Diagrams and the Davis-Putnam Procedure*, 1993. Submitted for publication.

[vBD83] Johan van Bentham and Kees Doets. Higher-order logic. In Gabbay and Guenthner [GG83], chapter I.4, pages 275–329.

[vH67] Jean van Heijenoort, editor. *From Frege to Gödel*. Harvard University Press, Cambridge, MA, 1967.

[WR27] A. N. Whitehead and B. Russell. *Principia Mathematica*. Cambridge University Press, Cambridge, revised edition, 1925–1927. Three volumes. The first edition was published 1910–1913.

[WR67] Alfred North Whitehead and Bertrand Russell. Incomplete symbols: Descriptions. In van Heijenoort [vH67], pages 216–223. First published 1910; Russell's original account "On Denoting" appeared in 1905.