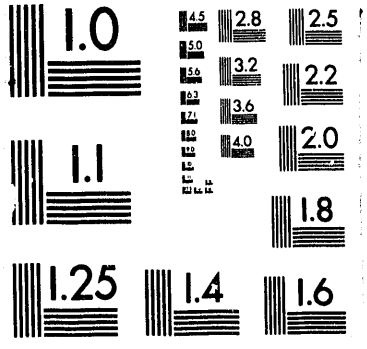

1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
10.0
11.2
12.5

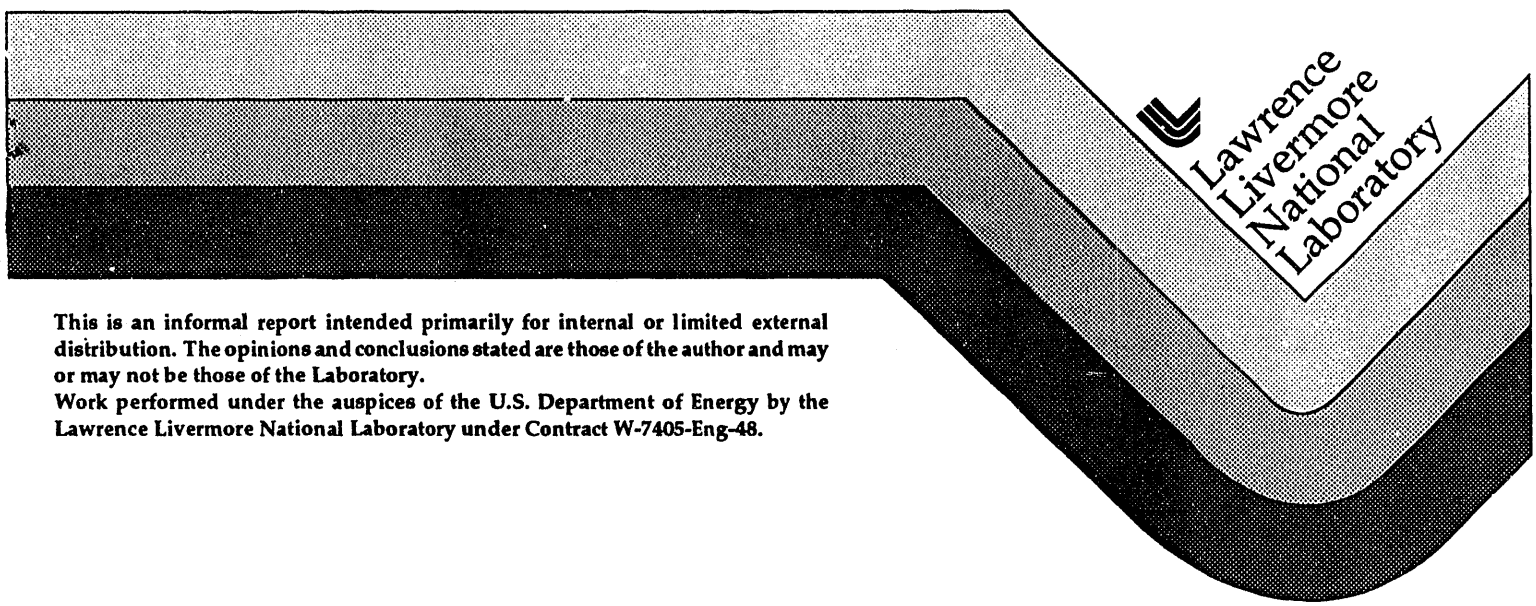
1 of 1

UCRL-ID-109416

Formal Methods in the Development of Safety Critical Software Systems

Lloyd G. Williams
Software Engineering Research
Boulder, CO

April 1992



Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

SERM-014-91

November 1991
(Revised April 1992)

Formal Methods in the Development of Safety Critical Software Systems

Version 3.0

Lloyd G. Williams
Software Engineering Research
Boulder, CO

Abstract

As the use of computers in critical control systems such as aircraft controls, medical instruments, defense systems, missile controls, and nuclear power plants has increased, concern for the safety of those systems has also grown. Much of this concern has focused on the software component of those computer-based systems. This is primarily due to historical experience with software systems that often exhibit larger numbers of errors than their hardware counterparts and the fact that the consequences of a software error may endanger human life, property, or the environment.

A number of different techniques have been used to address the issue of software safety. Some are standard software engineering techniques aimed at reducing the number of faults in a software protect, such as reviews and walkthroughs. Others, including fault tree analysis, are based on identifying and reducing hazards. This report examines the role of one such technique, formal methods, in the development of software for safety critical systems. The use of formal methods to increase the safety of software systems is based on their role in reducing the possibility of software errors that could lead to hazards.

Table of Contents

Executive Summary

Introduction

As the use of computers in critical control systems such as aircraft controls, medical instruments, defense systems, missile controls, and nuclear power plants has increased, concern for the safety of those systems has also grown. Much of this concern has focused on the software component of those computer-based systems. This is primarily due to historical experience with software systems that often exhibit larger numbers of errors than their hardware counterparts and the fact that the consequences of a software error may endanger human life, property, or the environment.

A number of different techniques have been used to address the issue of software safety. Some are standard software engineering techniques aimed at reducing the number of faults in a software protect, such as reviews and walkthroughs. Others, including fault tree analysis, are based on identifying and reducing hazards. This report examines the role of one such technique, formal methods, in the development of software for safety critical systems. The use of formal methods to increase the safety of software systems is based on their role in reducing the possibility of software errors that could lead to hazards.

The use of formal methods in the development of software systems is controversial. Proponents claim that the use of formal methods can eliminate errors from the software development process, and produce programs that are provably correct. Opponents claim that they are difficult to learn and that their use increases development costs unacceptably. This report discusses the potential of formal methods for reducing failures in safety critical software systems.

Formal Methods

Formal methods are approaches, based on the use of mathematical techniques and notations, for describing and analyzing properties of software systems. That is, descriptions of the system are written using notations which are based on mathematical expressions rather than a natural language, such as English, or other informal notation. These mathematical techniques and notations are typically drawn from areas of discrete mathematics, such as logic, set theory, or graph theory.

Formal methods can be viewed as the "applied mathematics" of software engineering. They are analogous to the applied mathematics used in the physical sciences and engineering, such as calculus or statistics, but different in that, as noted above, they usually involve discrete mathematics. As with applied mathematics in other engineering disciplines, formal methods may be used to construct models of a proposed system to analyze and predict its properties before it is actually constructed. The mathematics associated with formal methods are not any more difficult than those used in other engineering disciplines. Thus, most practicing software engineers should be able to understand the mathematics necessary to understand and write formal specifications. Performing the proofs associated with formal verification of those specifications is, however, likely to require more advanced training.

Formal methods offer two principal benefits in software development. First, the use of a precise mathematical notation eliminates the imprecision and ambiguity that is inherent in natural language descriptions. Formal methods also make it possible to analyze or prove, via rigorous mathematical

techniques, certain properties of software descriptions, e.g., specifications and designs. A developer may, for example, prove that a specification is complete and internally consistent or that the products of a given phase of the software process are consistent with those of a previous phase (e.g., that a design is consistent with its specification).

Formal Methods in the Software Process

Formal methods may be used in many different ways throughout the software process. Traditional approaches have used formal methods in a retrospective or after-the-fact fashion. In those cases, the software was developed using a standard, non-formal approach. After the software had been developed, a separate team prepared a formal specification and applied formal validation/verification techniques. The validation and verification techniques were used to prove certain properties about the specification, or proof techniques were applied to demonstrate that the code is consistent with the specification.

Alternatively, formal specification and verification may be performed in parallel with standard development methods, often also using two different development teams. Both of these approaches involve significant overhead, both in terms of additional development effort and communication between the teams. A more reasonable approach is to integrate formal methods into the development process. Rather than consider formal methods to be separate from, or parallel to, the "real" development process, they are a fundamental part of it.

Limitations of Formal Methods

Formal methods have the potential to help in reducing the number of errors in a software product. They do, however, have some limitations. These limitations are both practical and theoretical.

Limitations of formal methods become apparent at both ends of the development process. At the "front end," it is necessary to translate the customer's informally stated requirements into a formal specification. The correctness of this translation cannot be formally verified. Thus, it is not possible to be certain, in a formal sense, that the formal specification is an accurate representation of the user's actual requirements.

Other problems arise due to the fact that formal models describe only some aspects of a system's behavior. While behavioral characteristics of sequential programs are straightforward to model, concurrent systems present greater difficulty.

At the "back end" of the process, it is difficult to ensure that the actual behavior of the running system will match that described in a formal model. The correspondence is limited by factors such as the programming language, compiler, operating system, and hardware. A complete proof would require that the correctness of the compiler, operating system, and hardware all be formally verified as well. Because of this, it is not possible to assume that a program will function correctly simply because correspondence between the high-level source code and the specification has been proven.

Practical Applications of Formal Methods

The use of formal methods in software development is maturing and experience with their use is growing. We can divide the use of formal methods on real-world projects into two categories. Those that use formal methods in conjunction with proofs of correctness and those that use formal methods primarily as a means of stating system specifications in a concise and unambiguous fashion. The former approach has been pursued primarily by researchers and, at times, developers of commercial and government-contracted systems in the United States. These formal proofs of correctness have been applied principally to secure systems. The latter approach has been used primarily in Europe and has been applied to a wider variety of systems.

Standards governing system and software development have also begun to address the use of formal methods. While these have, until recently, focused primarily on security systems, newer standards are concerned with safety-critical systems.

Conclusions

Based on the preceding discussion, it is possible to draw several conclusions about the role of formal methods in the development of safety-critical systems.

- Formal methods are maturing. There is as growing body of experience in their use for developing commercial systems as well as security and safety-critical systems. As noted by Ralston and Gerhart, "formal methods are increasingly looked to as means that can be demonstrated to be the 'best possible practices' from the standpoint of legal or regulatory responsibility."

  Aspects of these methods, such as proofs of correctness, may, however, not be sufficiently mature for inclusion in standards for development and review of safety-critical software. Formal methods are also least mature in the areas of concurrency and timing, both of which are important for safety-critical control systems. These areas require additional research.

- Formal methods cannot, by themselves, guarantee that software meets its requirements or that it is error-free. Thus, other techniques, such as testing, will continue to be an important component of safety-critical systems development. Formal verification and testing can, however, be used in a complimentary fashion. As noted by Parnas, "verification can reveal problems that might never be found in testing, but testing can reveal errors in the assumptions made during the verification." Formal methods can also assist in establishing properties of the software which cannot be addressed by testing.

  Statistical testing techniques offer a means of quantitatively estimating the reliability of a tested software system. These techniques may, however, not be applicable to safety critical systems. Additional research is needed to determine their applicability.

- The use of formal methods can increase the time and cost required to develop software systems. This penalty can be reduced by integrating the formal methods into the development process and by having a set of standards and tools which support the use of those methods.

- The level of mathematical sophistication required to construct and understand formal specifications is not excessively difficult. Experience on several projects indicates that this level of mathematics is well within the capabilities of practicing engineers. More advanced skills are, however, required for those performing proofs of correctness.

- Formal methods can play an important role in the development of safety critical systems. They cannot, however, guarantee their safety.

- It is likely that no single formal approach will be adequate for describing all of the properties of safety critical systems. Combinations of methods may, however, provide the necessary capabilities. An example of such a combination is the use of Z to model the sequential aspects of a system and CSP to model concurrency.

  It is likely, however, that problems will arise in combining formal methods in this way. It may, for example, not be possible to combine the proof systems associated with the different methods, making complete verification impossible. This is an area in which further research is required.

1. Introduction

As the use of computers in critical control systems such as aircraft controls, medical instruments, defense systems, missile controls, and nuclear power plants has increased, concern for the safety of those systems has also grown. Much of this concern has focused on the software component of those computer-based systems. This is primarily due to historical experience with software systems that often exhibit larger numbers of errors than their hardware counterparts and the fact that the consequences of a software error may endanger human life, property, or the environment. Software components of safety critical systems have, in fact, already been blamed for a number of different accidents, some resulting in death. In one well-known case, a software error in the Therac-25 linear accelerator, a computer controlled medical instrument designed to deliver controlled doses of either electron beam or X-ray radiation for treatment of cancer, resulted in several patients receiving overdoses of radiation. At least two deaths and one serious injury have been attributed to this failure [Neum87].

Software safety is a complex issue. As Leveson [Leve91] points out, "computers are not inherently unsafe and software cannot directly cause accidents." Software can only be unsafe insofar as it operates as part of a system which is dangerous. A software failure in such a system may create a hazard. For example a software error[1] in a process control system may lead to a pressure relief valve being closed when it should be open. This may, in turn, lead to an accident if the pressure exceeds the capacity of the vessel.

Safety is, therefore, defined in terms of *hazards* and *risks*. A hazard is a condition, or set of conditions, that can produce an accident under the right circumstances. A pressure relief valve being closed when it should be open is an example of a hazard. The level of risk associated with the hazard depends on the probability that the hazard will occur, the probability of an accident taking place if the hazard does occur, and the potential consequences of the accident. The safety of software-based systems can, therefore, be increased if software failures that produce hazards are reduced or eliminated and the level of risk associated with system hazards is reduced.

A number of different techniques have been used to address the issue of software safety. Some are standard software engineering techniques aimed at reducing the number of faults that are introduced in the software development process, such as reviews and walkthroughs. Others, including fault tree analysis [Leve91], are based on identifying and reducing hazards. This report examines the role of one such technique, formal methods, in the development of software for safety critical systems. The use of formal methods to increase the safety of software systems is based on their role in reducing the possibility of software errors that could lead to hazards.

The use of formal methods in the development of software systems is controversial. Proponents claim that the use of formal methods can eliminate errors from the software development process, and produce programs that are provably correct. Opponents claim that they are difficult to learn and that their use increases development costs unacceptably. This report discusses the potential of formal methods for reducing failures in safety critical software systems. We begin with an overview of formal methods, their notations, and tools to support their use. This is followed by a discussion of the role of formal methods in the software process. Next, limitations of formal methods are described, followed by a presentation of some practical examples of their use. Semi-formal/informal methods and their relationship to formal methods are also discussed. The final

[1] Software errors may occur as a result of errors or omissions in specification, design, or coding.

section of the report offers a number of conclusions regarding the role of formal methods in construction of safety-critical systems.

2. Formal Methods

Formal methods are approaches, based on the use of mathematical techniques and notations, for describing and analyzing properties of software systems [Wing90]. That is, descriptions of the system are written using notations which are based on mathematical expressions rather than a natural language, such as English, or other informal notation. These mathematical techniques and notations are typically drawn from areas of discrete mathematics, such as logic, set theory, or graph theory.

Formal methods can be viewed as the "applied mathematics" of software engineering [Rals91]. They are analogous to the applied mathematics used in the physical sciences and engineering, such as calculus or statistics, but different in that, as noted above, they usually involve discrete mathematics. As with applied mathematics in other engineering disciplines, formal methods may be used to construct models of a proposed system to analyze and predict its properties before it is actually constructed. The mathematics associated with formal methods are not any more difficult than those used in other engineering disciplines. Thus, most practicing software engineers should be able to understand the mathematics necessary to understand and write formal specifications.[2] Performing the proofs associated with formal verification of those specifications is, however, likely to require more advanced training.

Formal methods are not necessarily methods in the sense that a "method" prescribes a particular sequence of actions or tasks to be performed during the software development process. They consist of a formal notation, a set of rules that govern use and manipulation of the notation (e.g., for constructing system models and deriving or proving properties of the models), and sometimes, but not always, a description of the steps to be performed when using the notation and rules in software development.

Formal methods offer two principal benefits in software development. First, the use of a precise mathematical notation eliminates the imprecision and ambiguity that is inherent in natural language descriptions. This enhanced level of precision means that specifications and designs are less likely to be misinterpreted and any errors or ambiguities that do exist will be discovered earlier. Errors that are discovered early are both easier and less costly to fix than those that are discovered in later phases of the development process [Boeh81].

Formal methods also make it possible to analyze or prove, via rigorous mathematical techniques, certain properties of software descriptions, e.g., specifications and designs. A developer may, for example, prove that a specification is complete and internally consistent or that the products of a given phase of the software process are consistent with those of a previous phase (e.g., that a design is consistent with its specification). Formal methods may also be used to answer questions

---

2 While understanding formal methods should be within the capabilities of most practicing software engineers, these topics are not typically included in undergraduate computer science curricula. Most graduate programs include courses on theory which cover the required mathematical concepts but do not typically address formal methods per se. Additional training is, therefore, likely to be necessary. Experience in introducing formal methods at Rolls Royce & Associates [Hill88] indicates that training in the underlying mathematical concepts must, in fact, accompany training in a particular formal method.

about whether a specification (as well as its design and implementation) meet certain (formally specified) requirements. Safety considerations may, for example, require that if some portion of the system is in state A, another cannot simultaneously be in state B. This is a common situation in control systems. For example, during early testing of the F-16, the test pilot told the computer to raise the landing gear while the plane was still on the runway. The resulting damage to the aircraft could have been averted if the developers had verified that when the aircraft was in the state "weight-on-wheels" it was not possible to also be in the state "landing gear up."

Two principal concerns that arise in systems development are validation and verification. Validation addresses whether the system that is produced actually fulfills the user's[3] needs. Verification, on the other hand, attempts to establish whether the products of a particular phase of the software development process meet the requirements established during the previous phase. Informally, we can express the difference by asking the questions [Boeh84]:

- *Validation*: "Are we building the right system?"
- *Verification*: "Are we building the system right?"

Validation is necessarily an informal process since the user's intentions are informal. Formal methods can, however, help with validation by removing ambiguities from specifications and improving the quality of feedback between developers and users. The type of interaction which is possible is illustrated by a widely-circulated anecdote involving an early application of Z (see below for a description of Z) [Dows91]. The program being specified was to assist in making hotel bookings for academic meetings and conferences. Use of inference techniques on a Z specification of the program revealed that it was possible to book unmarried people of opposite sex into a shared room. The developers were then able to discuss this unforeseen problem with the users and take appropriate action. Formal methods more directly address verification by making it possible to analyze or prove various properties of software products such as specifications, designs, and code.

Formal methods can help in assessing specifications for:

- *Completeness*: A specification is complete if it contains provisions for each requirement and each of those provisions is fully developed.
- *Consistency*: A specification is consistent if its provisions do not internally conflict with one another or externally conflict with specifications for other systems or entities.
- *Testability*: A specification is testable to the extent that it is possible to determine whether it is satisfied by the software developed from it. To be testable, a specification must be unambiguous and, to the extent possible, quantitative. Formal methods help in constructing such specifications.

Formal methods may be may be used in more traditional verification techniques such as reviews and testing. The precision provided by a formal description makes it easier to recognize problems and errors in reviews. Formal methods can also help in establishing properties of the software that

[3] In this context, the term "user" refers to the individual, group, or organization responsible for defining the requirements for the system or software. This individual, group, or organization may not actually be the end-user of the system or software.

cannot be determined through testing. Adherence to a requirement such as "The system shall never lose an input event" cannot, for example, be established through testing or simulation. The use of formal techniques can, however, establish whether the requirement is met [Hall90].

2.1 Methods and Notations

A number of different formal methods have been developed using different approaches (e.g., state-based versus axiomatic) and/or addressing different aspects of software development (e.g., specification versus testing). The choice of a method determines what can be described (e.g., sequential programs or parallel systems) as well as more superficial features such as syntax and modeling style.

There are several ways in which formal methods may be classified. One frequently-made distinction is between model-oriented and property-oriented methods [Wing90]. Model-oriented methods are used to construct a model of a system's behavior. State-transition diagrams, for example, are used to model the behavior of a system as a set of states and transitions between them. Property-oriented methods are used to describe software in terms of a set of properties, or constraints, that must be satisfied. Axiomatic techniques, for example, are used to specify the properties of an abstract data type in the form of a set of axioms that must be satisfied by operations on that type.

Several well-known formal methods are described below as illustrations. Additional descriptions of these methods, as well as examples of their use, may be found in the tutorial by Wing [Wing90].

- *VDM*: The Vienna Development Method (VDM) [Jone86] is a model-oriented specification and design method. It is used for specifying the behavior of abstract data types and sequential programs. The behavior is specified in terms of pre- and postconditions on pairs of states. A precondition is a predicate that must be true in order for the result of an operation to be defined. A postcondition is a predicate that is true after the operation has completed. VDM includes both a method, stepwise refinement, and a notation, META-IV.

  With VDM, a formal specification is constructed based on an informal statement of requirements. This specification is then refined or decomposed in a series of steps that add implementation details and a proof is constructed to demonstrate that each refined description satisfies the one from the previous step. The process of refinement and proof is continued until the META-IV description is at the level of a concrete source program in the target implementation language. At that point, the formal description can be translated into the target language and correspondence between the specification and the implementation can be proven.

- *Z*: The Z (pronounced "zed") notation [Spiv89], another model-oriented approach, is based on set theory and logic (first-order predicate calculus). It is also used for specifying the behavior of abstract data types and sequential programs. A Z specification describes a state space for the system and a set of operations that may be performed on that state space. The state space corresponds to the variables that determine the program's state. The operations are defined as relations on pairs of states from the state space which are conceptually similar to VDM's pre- and postconditions.

Z allows a complex specification to be divided into smaller, more manageable, parts using *schemas*. A schema groups variable declarations with a list of predicates that constrain possible values of those variables. These parts can then be combined to produce the overall description of the system.

An example of a Z specification is given in the Appendix.

- *EHDM*: EHDM [Rush91] is another model-oriented approach to formal specification. It combines first-order predicate calculus with elements of type theory (higher-order logic), lambda calculus, and relational (Hoare) calculus. An EHDM specification consists of a set of declarations for user-defined objects (types, constants, and variables) and operations (functions) together with a set of axioms that constrain the declarations.

  Specifications in EHDM are structured into modules which are similar to modules in programming languges. Modules may be combined "horizontally" to produce a complex specification from a set of simpler ones, much in the same way that program modules are combined to produce large programs. Modules may also be combined "vertically" to describe different levels of abstraction or refinement. A module at a high level of abstraction is implemented in terms of objects and operations from a lower-level, more detailed module.

  EHDM is supported by an automated environment for preparing specifications, performing syntactic and semantic analysis, theorem proving, and report generation.

- *Larch*: Larch [Gutt85] is a property-oriented method for specification of sequential programs and abstract data types. A Larch specification is "two-tiered;" it consists of an axiomatic component for specifying state-dependent behavior and an algebraic component for specifying state-independent properties. State-dependent behavior (written using a Larch interface language) is specified by giving preconditions, postconditions for operations along with a list of objects whose values may be modified by the operation. State-independent behavior (written using the Larch Shared Language) is specified by providing declarations for operators. The declaration consists of the name of the operator and its signature (the types of its input and output arguments).

- *CSP*: Communicating Sequential Processes (CSP) [Hoar85] is a formalism for describing concurrent systems. With CSP, the system is modelled as a network of sequential processes that communicate via named channels (rather than a common data area or global state). Processes executing in parallel synchronize when one process sends output and another receives input over a given channel.

  A CSP specification corresponds to a (possibly infinite) set of event sequences or traces. For a single sequential process, the trace is just the sequence of events that occur in the execution of that process. For parallel processes, the trace corresponds to interleavings of events that can occur concurrently. With CSP, a set of processes is specified and required properties of their event traces are stated. It is then possible to verify whether a CSP specification satisfies the specification on its traces.

- *Temporal Logic*: Temporal logics are formal methods which are based on the addition of time to first-order logic. Temporal logic is typically used for specifying properties of concurrent and distributed systems. Temporal logic uses modal operators to state assertions about a system's behavior by referring to past, present, or future states or events. Many different temporal logics exist, each with its own modal operators and inference rules. They all, however, have the common feature that they extend predicate logic by introducing the notion of time into predicates. Temporal logic operators may, for example, be used to assert that a predicate *P* is true in all states, will be true in the next state, or may be true in some future state.

  In describing concurrent systems, temporal logic specifications are typically written as assertions on sequences of states or events. In specifying a communication channel for example, we might specify that a message must be placed in the channel before it can be sent and that, if a message is placed in the channel, it will eventually be sent.

The above methods are based on textual notations. Graphical notations which have a well-defined syntax and semantics may also serve as the basis for formal methods. Graphical notations have several advantages over textual ones. They are often easier to understand and the symbols used in a graphical notation can often convey information more compactly than text. In addition, textual notations are read and understood sequentially. Graphical notations, however, do not have this inherently sequential nature. They can, therefore, represent concurrency more easily and more clearly.

One such graphical notation, StateCharts, is described below. Other, comparable notations include the Transformation Schema [Ward86], the notation developed by Hatley and co-workers [Hatl87], and the Extended System Modeling Language (ESML) [Bruy88].

- *StateCharts*: StateCharts are used to specify state transitions in reactive systems [Hare85] [Hare87]. Reactive systems, in contrast to transformational systems, cannot be described in terms of a simple function that maps inputs to outputs. The response that a reactive system provides to an input event, for example, depends on the current state of the system. The current state, in turn, is a function of the inputs that have already been received. Safety-critical systems are typically reactive systems.

  StateCharts are based on higraphs [Hare88], a visual formalism that allows compact representation of hierarchies of state-transition diagrams. A StateChart description actually consists of three components: StateCharts for describing dynamic and behavioral characteristics, Activity Charts for describing functional properties, and Module Charts for describing the physical structure of the system. Activity Charts are conceptually similar to data flow diagrams. Module Charts are an addition that makes it possible to use the StateChart approach for design as well as specification. StateCharts may be used to specify the behavior of both sequential and parallel programs.

  The use of StateCharts is supported by the STATEMATE™ tool [Hare90] which provides a graphical editor and analysis tool for the notation. STATEMATE™ provides assistance in checking the completeness and consistency of a StateChart model. It also provides the capability of executing StateCharts to produce event traces.

Because formal models of software systems are based on notations that have a well-defined syntax and semantics, like a programming language, they have at least the potential to be executable. An executable model is known as an *operational specification* [Zave84], a specification described in terms of implementation independent model structures that produces the desired behavior of the target system. An operational specification may be executed in the development environment to provide insight into the program's dynamic behavior. Test cases can be applied and the model exercised to produce output which can be compared to the specification, much in the same way that a conventional program is tested. Executable specifications can also serve as rapid prototypes of the system under development, allowing more effective communication and feedback between developers and users.

While not all formal models are executable, many are. The PAISLey notation [Zave82], OBJ [Gogu79], as well as StateCharts and equivalent graphical notations may, among others, be used to construct executable specifications. Portions of formal specifications written in other, non-executable notations may also sometimes be executed by translating them into programming languages such as Prolog.

2.2 Tools

The use of computer-aided software engineering (CASE) tools to support the use of formal methods is in its infancy. A number of tools have, however, been developed. While most of them are research tools, some have been applied to real-world development projects. Here we review the status of several categories of tools to support the use of formal methods.

- *Documentation support*: Many of the text-based formal methods make heavy use of special symbols (see the Appendix, for example). In many cases, support for typesetting these symbols is weak or non-existent, making it difficult to produce adequate documentation of formal specifications. Support for typesetting several popular mathematical notations, in the form of macros for the TEX® word processor, is available.

- *Analysis tools*: Simple tools for checking the syntax of specifications written using some formal notations, performing type checking within a specification, and performing static semantic analysis (for sequential programs) are available. More sophisticated tools that perform symbolic execution (i.e., execution where actual values for variables are replaced with symbols, such as X) of specifications also exist.

- *Theorem provers*: Theorem provers automate part or all of the proof process. They help eliminate errors in proofs, make it possible to prove theorems more quickly, and make it possible to undertake more difficult proofs. Several different theorem provers have been developed and have been used successfully in projects of realistic size (see, e.g., [Wood89]).

3. Formal Methods in the Software Process

Formal methods may be used in many different ways throughout the software process. Traditional approaches have used formal methods in a retrospective or after-the-fact fashion. In those cases, the software was developed using a standard, non-formal approach. After the software had been

developed, a separate team prepared a formal specification and applied formal validation/verification techniques. The validation and verification techniques were used to demonstrate or prove certain properties about the specification, or proof techniques were applied to demonstrate that the code is consistent with the specification.

Alternatively, formal specification and verification may be performed in parallel with standard development methods, often also using two different development teams. Both of these approaches involve significant overhead, both in terms of additional development effort and communication between the teams. A more reasonable approach [Kemm90] is to integrate formal methods into the development process. Rather than consider formal methods to be separate from, or parallel to, the "real" development process, they are a fundamental part of it.

The integrated process begins with a restatement of the user's requirements, which are typically expressed informally, in formal terms. The product of this part of the process may be a formal specification or a formal statement of the "critical requirements" [Kemm90] which is then translated into a formal specification. In either case, the initial restatement of the user's requirements must be validated. This validation process is necessarily informal due to the informal nature of the initial requirements. Several authors have, however, noted that a formal specification can aid the validation process by removing the ambiguity associated with natural language specifications and by improving communication with, and feedback from, users (see, e.g., [Hall91]).

From here, the next step depends on the particular approach which is being used and the complexity of the problem. In simple cases, the formal specification may be refined into successively more detailed descriptions by adding details. At each refinement step, a proof is constructed to verify the consistency between each level and the previous one. The refinement process continues until the detailed specification is at a level which can be translated into a high-level programming language. Finally, the consistency between the high-level code and the lowest-level specification is proven.

In more complex cases, such as those involving a high level of concurrency, a formal, high-level design which describes the mapping of the specification onto the target environment must be constructed. This high-level design describes how specification-level information is to be realized in specific software structures, such as sequential modules and concurrent tasks.[4] Again, consistency between the specification and high-level design must be proven. The design may then be refined in successive steps and verified as described above. Ultimately, the design is translated into code in a high-level target language and the implementation is verified against the design.

While the discussion has, thus far, focused on the use of formal methods to construct and refine software specifications, they can also be used in other ways. Formal methods can, for example, be used to help identify test cases. One such approach is to use a formal statement of the pre- and postconditions on a module's use to identify test cases that probe both valid and invalid input as well as boundary conditions. These test cases can be applied to the code or, in the case of executable specifications, such as StateCharts, to a specification-level description of the software.

If we define formal methods to be the application of mathematical techniques to software development, it is possible to identify other mathematical techniques which are also applicable. Statistical techniques can, for example, be used to guide testing strategies [Musa89]. Statistical

[4] The design must also specify additional details, such as processor allocation in a multi-processor environment, but these are not typically addressed by formal software development methods.

testing techniques are based on the specification of a required level of reliability[5] and an operational profile for the software. The required reliability is stated in terms of the expected time between failures under specified conditions (the operational profile). The operational profile is used to select test inputs, together with their relative frequencies, to exercise the software in a realistic manner. A record of the amount of execution time between failures is maintained and any failures observed are eliminated by correcting their underlying faults. Testing is continued until the statistical model indicates that the required level of reliability has been attained. Alternatively, the statistical models can be used to predict the expected reliability after a fixed testing period.[6]

The use of formal methods can have a significant impact on the software development process as well as its products. Critics claim that formal methods can increase development time and costs unacceptably. Certainly, approaches that use separate teams to perform development and formal specification/verification can add substantial amounts of overhead to the process. This overhead includes the extra time to prepare yet another specification (the formal one) and verify it as well as the time required for communication between the teams. The extra personnel required for the two-team approach also add cost to the project.

This time and cost penalty can be reduced by integrating the formal methods into the development process [Kemm90] and by having a set of standards and tools which support the use of the methods [Parn90]. Section 5 describes several projects that have applied formal methods to large scale software systems. In those projects where formal methods were used outside of the main stream of development (e.g., [Parn90]) their use was, indeed, felt to be an extra burden.

Even if the use of formal methods does not increase the total amount of time required to develop a software product, it may alter the distribution of that time among the various phases of the project. Traditional approaches to development of non-critical software emphasize coding at the expense of other phases, such as specification and design. As a result, the bulk of development time (perhaps more than 60% [Zelk78]) is spent on coding and testing. Much of this testing is concerned with removing errors that would have been found early in the development process (or, perhaps, not even made) using formal methods.

Integrating formal methods into such a software development process typically requires that more time be spent in the specification phase. The extra time spent in the specification phase is recovered by spending less time in coding and testing. It is possible to spend less time in the coding and testing phases because coding is more straightforward and because errors that would normally be discovered during testing have already been detected and eliminated.

If maintenance is included, the savings can be even larger. Without formal methods, maintenance can consume two thirds of the effort expended on a system over its entire lifetime [Zelk78]. Much of this effort is devoted to finding and fixing errors that were introduced during development. Most of these errors occur because either the user's requirements or the specification were

---

[5] Software reliability is defined as "The probability that software will not cause the failure of a system for a specified time under specified conditions" [IEEE83]. Note that safety and reliability, while obviously related, are not the same. A system may be extremely unreliable but, if no failure produces a hazard, it would not be considered unsafe.

[6] Reliability, as estimated in terms of expected time between failures, may not be an accurate indicator of the safety of a software system. Failures associated with high-risk hazards are frequently due to unanticipated or low probability events which are unlikely to be reflected in the operational profile. As noted by Parnas [Parn90], reliability estimates obtained in this fashion will only be as good as the operational profile on which they are based.

ambiguous and, therefore, misinterpreted [Basi84]. The use of formal methods can help discover and eliminate these errors at an earlier stage of the development process.

Data which would indicate whether the use of formal methods would have a similar impact on the development process for safety-critical systems is scarce. In one project involving development of nuclear reactor protection software, however, engineers at Rolls Royce and Associates estimated that as much as 75% of the post integration errors could have been eliminated if formal methods had been used [Hill88].

4. Limitations of Formal Methods

Formal methods have the potential to help in reducing the number of errors in a software product. They do, however, have some limitations. These limitations are both practical and theoretical.

Limitations of formal methods become apparent at both ends of the development process. At the "front end," it is necessary to translate the customer's informally stated requirements into a formal specification. The correctness of this translation cannot be formally verified. Thus, it is not possible to, in a formal sense, be certain that the formal specification is an accurate representation of the user's actual requirements. The informal requirements may have been misinterpreted or a key requirement may have been left out. Experience indicates that the use of a formal specification can remove ambiguities and help clarify the requirements, improving communication with users. Improved communication, particularly through the use of executable specifications or simulations, can result in better feedback and can help reduce, but not eliminate, the chances of errors or omissions.

Other problems arise due to the fact that formal models describe only some aspects of a system's behavior [Hall91]. While behavioral characteristics of sequential programs are straightforward to model, concurrent systems present greater difficulty. Techniques for describing concurrent systems are available (e.g., CSP). The results of analyzing the descriptions are, however, often difficult to interpret. Approaches to simplifying the analysis of descriptions of concurrent systems exist [Avru86] but are still in the research stage. It has been estimated that "verification of a concurrent system written in a real language" may be as much as ten years away [Spei90]. For other properties, such as timing constraints, it is even more difficult to construct and use formal models. Finally, the real world is not a formal system and certain aspects of its behavior, such as noisy sensors, cannot be adequately modeled using formal software methods.

Throughout the process, it is possible for errors to occur in proofs. Even automated theorem provers have limitations.

At the "back end" of the process, it is difficult to ensure that the actual behavior of the running system will match that described in a formal model. The correspondence is limited by factors such as the programming language, compiler, operating system, and hardware. A complete proof would require that the correctness of the compiler, operating system, and hardware all be formally verified as well. Because of this, it is not possible to assume that a program will function correctly simply because correspondence between the high-level source code and the specification has been proven.

Statistical techniques for predicting failure rates also have limitations when applied to safety-critical systems. In order for these techniques to be valid, it is necessary to observe a statistically significant number of failures. Safety-critical systems, however, are typically constructed with more care than their non-critical counterparts and failure rates may be too low to achieve statistically valid results. In any case, it is not usually acceptable to exercise the software under conditions where failures might occur so such data is difficult to collect [Hill88].

5. Practical Applications of Formal Methods

The use of formal methods in software development is maturing and experience with their use is growing. We can divide the use of formal methods on real-world projects into two categories: those that use formal methods in conjunction with proofs of correctness and those that use formal methods primarily as a means of stating system specifications in a concise and unambiguous fashion but do not perform proofs of correctness. Bjorner [Bjor87] distinguishes between these two approaches by classifying the first as *formal* and the second as *rigorous*. The former approach has been pursued primarily by researchers and, at times, developers of commercial and government-contracted systems in the United States. These proofs of correctness have been applied principally to secure systems. The latter approach has been used primarily in Europe and has been applied to a wider variety of systems.

Standards governing system and software development have also begun to address the use of formal methods. While these have, until recently, focused primarily on security systems, newer standards are concerned with safety-critical systems.

The following section reviews the application of formal methods to real-world projects by describing several projects for which summary information is available in the open literature. This is followed by a discussion of standards for system development which require the use of formal techniques.

5.1 Experience with Formal Methods

Formal methods have now been applied in the development a significant number of software systems, some of substantial size and complexity. A survey conducted by the Microelectronics and Computer Technology Corporation (MCC) identified approximately 60 such case studies [Rals91]. Of these, about 12 involved the development of commercial products. The others were either large-scale experiments or work performed under government contracts. Several of these projects are described below.

- *IBM (UK)*: One of the most widely-known applications of formal methods to a commercial product is the use of Z by IBM in the United Kingdom for specification of key modules in its Customer Information Control System (CICS) [Nix88]. The use of Z was part of a larger effort aimed at improving the quality of this large (>500,000 lines of code) and widely-used product. This effort included precise definition of the development process, extensive use of inspections for all software products, well-defined software test criteria, more extensive use of high-level languages (instead of assembler), and introduction of specification and design languages.

In this project, Z was used for description only, proofs that the implementation matched the specification were not carried out. Proofs of some portions of the CICS system have subsequently been performed [Wood89] and it is likely that proofs of correctness will play a more important role in future releases.

The introduction of Z involved an extensive training program that included courses on software engineering techniques, set theory, and the Z notation. The current release of CICS consists of over 268,000 lines of code. Of this, 39,000 lines were developed using Z and another 11,000 were developed partially using Z. Fewer errors were found in Z specified modules during inspections, leading the developers to conclude that the Z specified components were of higher quality [Nix88]. The developers estimated that there was a savings of 9% over traditional techniques for those 39,000 lines developed using Z [Wood90a].

- *Atomic Energy Control Board (Canada)*: A formal approach was used in the development of a computer-controlled shutdown system for the Darlington nuclear power plant [Parn90]. This project used a set of techniques developed by Parnas and co-workers [Heni80] to verify the software for the shutdown system. The technique makes use of a tabular form of finite state specification known as condition and event tables or function tables.

  The verification was retrospective. Function tables were produced independently for the requirements and the code. These were then manually compared to determine if discrepancies existed. Discrepancies were classified as benign or malignant. Note that this project did not involve a formal proof of correctness. The formal specifications derived from the requirements and code were simply compared for discrepancies.

  In describing the results of the project, Parnas and co-workers [Parn90] note that the techniques were "hard to use." This was due to two factors: 1) producing the function tables from the informal specification and the code was a tedious process, and, 2) there was a lack of automated tools to support the techniques. They conclude, however, that "our experience demonstrates that precise requirements for specifications and mathematical analysis of code are feasible."

- *Praxis (UK)*: Formal specifications were used in development of a set of software engineering (CASE) tools to support SSADM, a structured systems analysis and design method [Hall91]. The formal specifications were written in Z and were used to clarify requirements, uncover errors or ambiguities, and assist in making decisions about the functionality of the system. Again, formal program verification was not performed.

- *Rolls Royce (UK)*: Rolls Royce and Associates used a combination of informal and formal specifications in the development of nuclear reactor protection software [Hill90], [Hill88]. In this project, an informal English specification of the requirements was written first. A formal specification was then written from the English version using VDM. Design then proceeded more informally using "Yourdon analysis and design techniques" [Hill90] and the system was implemented using a "safe" subset of Pascal. A variety of approaches were used for validation and verification including "animation" of the specification, static analysis and formal proofs based on the VDM specification, and testing.

Rolls Royce engineers noted that the use of formal methods led to fewer changes due to errors as the project evolved and that confidence in the safety of the system was increased. Modifications to the design were evaluated and it was determined that the majority of changes after integration were due to errors or ambiguities at the specification stage. As noted above, it was estimated that, of these, 75% could have been avoided if formal specifications and proofs had been used [Hill88].

Use of formal methods did not increase either the cost or the development time for this project. While the amount of effort expended in specification was larger than in previous projects, this was offset by a reduced amount of rework at the end of the process. They also noted that one of the main benefits of their development process was the increased level of discipline in checking and validating requirements which was imposed on the project team.

Other interesting practical applications of formal methods include Z specification of the IEEE standard for binary floating-point arithmetic [Barr89], development of a formal model and abstract specification (in Z) for a new line of oscilloscopes at Tektronix [Deli90], and development of a secure release terminal for moving information between computer environments at different levels of security [Kemm90].

5.2 Formal Methods in Standards

The British Ministry of Defence (MoD) has developed an interim standard (MoD-Std-00-55) [MoD91] for development of safety critical software which mandates the use of formal methods.[7] This standard establishes standards for good software engineering practice. These include: development of, and adherence to, a project plan; use of trained and qualified staff; enforcement of configuration management; independent safety assessment as well as independent validation and verification of safety-critical software; and, the use of formal, mathematical specification and design techniques.

The MoD standard requires that specifications be written using both English and "formal mathematical techniques." The formal specification is to be validated against the informal English version by "techniques such as prototyping or animation." The design is also required to be given in a formal specification which must be verified, using formal proof of correctness or rigorous arguments, against the formal specification. Similarly, the code must be verified against the design and only approved, validated compilers may be used to translate the high-level language into executable form.

Ministry of Defence Standard 00-55 currently has the status of a draft, interim standard and is, therefore, still undergoing review and modification. An earlier draft (dated May 1989) provided a list of approved formal methods for use in developing safety critical software. The methods were: VDM [Jone86], OBJ [Gogu79], Z [Spiv89], HOL [Gord85], CCS [Miln86], CSP [Hoar85], temporal logic, and LOTOS [ISO89]. This list has been omitted from the current draft .

A series of MoD-funded studies has, however, indicated that proofs of correctness are expensive to perform and that the techniques may not be sufficiently mature to be applied to large systems

[7] A companion standard, MoD-Std-00-56 [MoD89], specifies requirements for the analysis of hazards in safety critical systems.

[Lee91]. While proofs may be applied to smaller programs or portions of systems, there are substantial risks involved in inferring properties of a system from proofs about properties of parts of the system. At this point, it is not clear what impact this will have on future versions of MoD-Std-00-55.

A related standard, IEC-880 [IEC86], notes that "a formal specification language may be a help to show coherence and completeness of the software functional requirements." While this standard requires a verification procedure at the end of requirements specification, design, and coding, it does not require that formal proofs of correctness be used in these verification procedures. There is, instead, heavy reliance on testing as the primary form of verification. "Program analysis," using automated tools if possible, is strongly recommended and proofs of correctness are suggested as supplementary techniques.

6. Semi-Formal and Informal Methods

Several well-known analysis and design methods are often classified as semi-formal or informal. These classifications are often applied inconsistently. Jackson Structured Programming (JSP) [Jack75], for example, is classified as informal or semi-formal by Wing [Wing90] but is classified as formal by Ralston and Gerhart [Rals91]. The Jackson notation is a graphical form of regular expressions and is, therefore, based on a finite-state machine formalism. Similarly, StateCharts are classified as formal by Wing, but the Transformation Schema is typically classified as semi-formal or informal. This section examines several such methods, the reasons behind their classification as informal or semi-formal, and discusses their applicability in a development process that uses formal techniques.

StateCharts, Transformation Schemas, the Hatley notation, and ESML all use similar visual notations for modeling systems. They belong to a class of graphical languages that may be called "embedded behavior pattern" languages [Ward89] because they embed a mechanism for describing patterns of behavior within a flow diagram notation. Behavior patterns describe different qualitative behaviors or modes, together with the events that cause changes in mode, for the entity being modeled. The flow notation models the movement of information through the system together with processes that use or change this information. Combining these two modeling capabilities makes it possible to model control of processes. A process may, for example, be turned on or off when a change in mode occurs.

Each of these notations uses a form of finite-state machine to model behavior patterns. StateCharts, as noted above, use higraphs a form of hierarchical state-machine. The Transformation Schema, Hatley's notation, and ESML all use more traditional state-machine models. The Transformation Schema and ESML use Mealy-type finite automata while the Hatley notation uses Moore-type state machines. In each case, use of finite-state machines as a basis for the behavior modeling component makes a model developed in the given notation sufficiently formal to perform static analysis (e.g., reachability, deadlock, non-termination) and proofs of correctness. Models using these notations can also be executed with appropriate tools (see, e.g., [Webb86], [Reil87], [Blum88], [Hare90]).

Problems of formality arise, however, with respect to the various flow diagram notations. In each case, the flow diagrams are based on the data flow diagram introduced by DeMarco in the mid-1970s [DeMa78]. This notation is currently in wide use for systems analysis and design and is

supported by a wide range of CASE (Computer Aided Software Engineering) products. The original formulation of data flow diagrams was informal and their use remains largely informal today. This is especially true with respect to specification of the processes that transform information. They are typically specified using informal "structured English."

Data flow diagrams are not, however, inherently informal. They can be formally defined using formalisms from graph theory and proofs can be constructed to demonstrate their properties [Tao91]. Formal methods (e.g., VDM, Z) can also be used to specify processes, making individual process specifications completely formal and providing access to the inference systems provided by the formal methods. Such an integration of VDM and data flow diagrams has been demonstrated by Fraser and co-workers [Fras91]. The fact that they are typically used in an informal or semi-formal way, however, has led many software professionals to classify methods which use data flow notations as semi-formal or informal.

These methods have the advantage that they are widely known and automated support for their use is available in the form of CASE products. They are also perceived (correctly or incorrectly) as easier to understand and use than methods such as Z or VDM and their graphical nature offers advantages over textual notations. By using formal definitions of flow diagrams and formal specifications of their processes, it is possible to bring these methods into the realm of formal methods. They can also be used in conjunction with other formal methods, as described by Fraser, et. al. They can also serve as a "human interface" for other formal methods. Note, however, that current CASE tools do not provide the level of analysis capabilities that are required for safety-critical systems. It would also be difficult to integrate these CASE products with other tools that support formal methods.

7. Conclusions

Based on the preceding discussion, it is possible to draw several conclusions about the role of formal methods in the development of safety-critical systems.

- Formal methods are maturing. There is as growing body of experience in their use for developing commercial systems as well as security and safety-critical systems. As noted by Ralston and Gerhart [Rals91], "formal methods are increasingly looked to as means that can be demonstrated to be the 'best possible practices' from the standpoint of legal or regulatory responsibility."

  Aspects of these methods, such as proofs of correctness, may, however, not be sufficiently mature for inclusion in standards for development and review of safety-critical software. Formal methods are also least mature in the areas of concurrency and timing, both of which are important for safety-critical control systems. These areas require additional research.

- Formal methods cannot, by themselves, guarantee that software meets its requirements or that it is error-free. Thus, other techniques, such as testing, will continue to be an important component of safety-critical systems development. Formal verification and testing can, however, be used in a complimentary fashion. As noted by Parnas [Parn90], "verification can reveal problems that might never be found in testing, but testing can

reveal errors in the assumptions made during the verification." Formal methods can also assist in establishing properties of the software which cannot be addressed by testing.

Statistical testing techniques [Musa89] offer a means of quantitatively estimating the reliability of a tested software system. These techniques may, however, not be applicable to safety critical systems. Additional research is needed to determine their applicability.

- The use of formal methods can increase the time and cost required to develop software systems. This penalty can be reduced by integrating the formal methods into the development process and by having a set of standards and tools which support the use of those methods.

- The level of mathematical sophistication required to construct and understand formal specifications is not excessively difficult. Experience on several projects indicates that this level of mathematics is well within the capabilities of practicing engineers. More advanced skills are, however, required for those performing proofs of correctness.

- Formal methods can play an important role in the development of safety critical systems. They cannot, however, guarantee their safety.

- It is likely that no single formal approach will be adequate for describing all of the properties of safety critical systems. Combinations of methods may, however, provide the necessary capabilities. An example of such a combination is the use of Z to model the sequential aspects of a system and CSP to model concurrency.

  It is likely, however, that problems will arise in combining formal methods in this way. It may, for example, not be possible to combine the proof systems associated with the different methods, making complete verification impossible [Wood90b]. This is an area in which further research is required.

Formal methods can be valuable in increasing the reliability of safety-critical software systems. They must, however, be used in conjunction with other techniques. These include: good software engineering practices (a well-defined process, configuration management, etc.), safety analysis, and testing.

8. References

[Avru86] G. S. Avrunin, L. K. Dillon, J. C. Wileden, and W. E. Riddle, "Constrained Expressions: Adding Analysis Capabilities to Design Methods for Concurrent Software Systems," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, pp. 278-292, 1986.

[Barr89] G. Barrett, "Formal Methods Applied to a Floating-Point Number System," *IEEE Transactions on Software Engineering*, vol. 15, no. 5, pp. 611-621, 1989.

[Basi84] V. R. Basili and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM*, vol. 27, no. 1, pp. 42-52, 1984.

[Bjor87] D. Bjorner, "On The Use of Formal Methods in Software Development," *Proceedings of the Ninth International Conference on Software Engineering*, Monterey, CA, March 1987, pp. 17-29.

[Blum88] R. Blumofe and A. Hecht, "Executing Real-Time Structured Analysis Specifications," *Software Engineering Notes*, vol. 13, no. 3, pp. 32-40, 1988.

[Boeh84] B. W. Boehm, "Verifying and Validating Software Requirements and Design Specifications," *IEEE Software*, vol. 1, no. 1, pp. 75-88, 1984.

[Boeh81] B. W. Boehm, *Software Engineering Economics*, Englewood Cliffs, NJ, Prentice-Hall, 1981.

[Bruy88] W. Bruyn, R. Jensen, D. Keskar, and P. Ward, "ESML: An Extended Systems Modeling Language," *Software Engineering Notes*, vol. 13, no. 1, pp. 58-67, 1988.

[Deli90] N. Delisle and D. Garlan, "A Formal Specification of an Oscilloscope," *IEEE Software*, vol. 7, no. 5, pp. 29-36, 1990.

[DeMa78] T. DeMarco, *Structured Analysis and System Specification*, New York, NY, Yourdon, Inc., 1978.

[Dows91] M. Dowson, Software Design and Analysis, Inc., personal communication, October 1991.

[Fras91] M. D. Fraser, K. Kumar, and V. K. Vaishnavi, "Informal and Formal Requirements Specification Languages: Bridging the Gap," *IEEE Transactions on Software Engineering*, vol. 17, no. 5, pp. 454-466, 1991.

[Gogu79] J. A. Goguen and J. Tardo, "An Introduction to OBJ: A Language for Writing and Testing Software Specifications, in *Specification of Reliable Systems*, IEEE, 1979.

[Gord85] M. Gordon, "HOL: A Machine-Oriented Formulation of Higher-Order Logic," Technical Report No. 68, University of Cambridge, 1985.

[Gutt85] J. V. Guttag, J. J. Horning, and J. M. Wing, "The Larch Family of Specification Languages," *IEEE Software*, vol. 2, no. 5, pp. 24-36, 1985.

[Hall91] A. Hall, "Seven Myths of Formal Methods," *IEEE Software*, vol. 7, no. 5, pp. 11-19, 1991.

[Hare90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *IEEE Transactions on Software Engineering*, vol. 16, no. 4, pp. 403-414, 1990.

[Hare88] D. Harel, "On Visual Formalisms," *Communications of the ACM*, vol. 31, no. 5, pp. 514-530, 1988.

[Hare87] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," in *Science of Computer Programming*, vol. 8, D. Gries, ed., 1987, pp. 231-274.

[Hare85] D. Harel and A. Pnueli, "On the Development of Reactive Systems," in *Logic and Models of Concurrent Systems*, NATO ASI Series, vol. 133, K. R. Apt, ed., Berlin, Springer-Verlag, 1985, pp. 477-498.

[Hat87] D. J. Hatley and I. A. Pirbhai, *Strategies for Real-Time System Specification*, New York, NY, Dorset House, 1987.

[Heni80] K. L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 1, pp. 2-13, 1980.

[Hill90] J. V. Hill, P. Robinson, and P. A. Stokes, "Safety Critical Software in Control Systems," in *Computers and Safety*, Stevenage, Herts, England, Institute of Electrical Engineers, 1990, pp. 92-96.

[Hill88] J. V. Hill, "The Development of High Reliability Software: RR&A's Experience for Safety Critical Systems," *Proceedings of Software Engineering 88*, Liverpool, 1988, pp. 169-172.

[Hoar85] C. A. R. Hoare, *Communicating Sequential Processes*, Englewood Cliffs, NJ, Prentice-Hall, 1985.

[IEC86] International Electrochemical Commission, "Software for Computers in the Safety Systems of Nuclear Power Stations," Publication 880, Geneva, Switzerland, 1986.

[IEEE83] *IEEE Standard Glossary of Software Engineering Terminology*, IEEEStd. 729-1983, Institute of Electrical and Electronics Engineers, Inc., New York, NY, 1983.

[ISO89] International Organization for Standardization, "Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour," 1989.

[Jack75] M. A. Jackson, *Principles of Program Design*, New York, NY, Academic Press, 1975.

[Jone86] C. B. Jones, *Systematic Software Development Using VDM*, Englewood Cliffs, NJ, Prentice-Hall, 1986.

[Kemm90] R. A. Kemmerer, "Integrating Formal Methods into the Development Process," *IEEE Software*, vol. 7, no. 5, pp. 37-50, 1990.

[Lee91] G. Lee, Vitro Corporation, personal communication, November 1991.

[Leve91] N. G. Leveson, "Software Safety in Embedded Computer Systems," *Communications of the ACM*, vol. 34, no. 2, pp. 34-46, 1991.

[Miln86] R. Milner, "A Calculus of Communicating Systems," Technical Report No. ECS-LFCS-86-7, Laboratory for the Foundations of Computer Science, Edinburgh University, Edinburgh, 1986.

[MoD91] Ministry of Defence, "Requirements for the Procurement of Safety Critical Software in Defence Equipment," Ministry of Defence, Glasgow, UK, April 1991.

[MoD89] Ministry of Defence, "Requirements for the Analysis of Safety Critical Hazards," Ministry of Defence, Glasgow, UK, May 1989.

[Musa89] J. D. Musa and A. F. Ackerman, "Quantifying Software Validation: When to Stop Testing?," *IEEE Software*, vol. 6, no. 3, pp. 19-27, 1989.

[Neum87] P. G. Neumann, "Risks to the Public in Computers and Related Systems," *Software Engineering Notes*, vol. 12, no. 3, 1987.

[Nix88] C. J. Nix and B. P. Collins, "The Use of Software Engineering, Including the Z Notation, in the Development of CICS," *Quality Assurance*, vol. 14, no. 3, pp. 103-110, 1988.

[Parn90] D. L. Parnas, G. J. K. Asmis, and J. D. Kendall, "Reviewable Development of Safety Critical Software," *Proceedings of the International Conference on Control and Instrumentation in Nuclear Installations*, Glasgow, May, 1990, pp. 1-17.

[Rals91] T. J. Ralston and S. L. Gerhart, "Formal Methods: History, Practice, Trends, and Prognosis," Technical Report No. STP-FT-009-91, Software Technology Program, Microelectronics and Computer Technology Corporation, Austin, TX, April 1991.

[Reil87] E. Reilly and J. Brackett, "An Experimental System for Executing Real-Time Structured Analysis Models," *Proceedings of the Twelfth Structured Methods Conference*, Chicago, IL, 1987, pp. 301-314.

[Rush91] J. Rushby, F. von Henke, and S. Owre, "An Introduction to Formal Specification and Verification Using EHDM," Technical Report No. SRI-CSL-91-02, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1991.

[Spei90] K. Speierman, "Working Group 2: What Are the Theoretical and Practical Limits of Formal Methods?," in *Formal Methods for Trustworthy Computer Systems (FM89)*, D. Craigen and K. Summerskill, ed., London, Springer-Verlag, 1990, pp. 68-74.

[Spiv89] J. M. Spivey, *The Z Notation: A Reference Manual*, Englewood Cliffs, NJ Prentice-Hall, 1989.

[Tao91] Y. Tao and C. Kung, "Formal Definition and Verification of Data Flow Diagrams," *Journal of Systems and Software*, vol. 16, no. 1, pp. 29-36, 1991.

[Ward89] P. T. Ward, "Embedded Behavior Pattern Languages: A Contribution to a Taxonomy of CASE Languages," *Journal of Systems and Software*, vol. 9, no. 2, pp. 109-128, 1989.

[Ward86] P. T. Ward, "The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, pp. 198-210, 1986.

[Webb86] M. Webb and P. Ward, "Executable Data Flow Diagrams: An Experimental Implementation," *Proceedings of Structured Development Forum IX*, Seattle, WA, 1986.

[Wing90] J. M. Wing, "A Specifier's Introduction to Formal Methods," *IEEE Computer*, vol. 23, no. 9, pp. 8-24, 1990.

[Wood90a] J. C. P. Woodcock, "Z," in *Formal Methods for Trustworthy Computer Systems (FM89)*, D. Craigen and K. Summerskill, ed., London, Springer-Verlag, 1990, pp. 57-62.

[Wood90b] J. C. P. Woodcock, "Working Group 1: What Is the Applicability of Formal Methods in Systems Engineering?," in *Formal Methods for Trustworthy Computer Systems (FM89)*, D. Craigen and K. Summerskill, ed., London, Springer-Verlag, 1990, pp. 63-67.

[Wood89] J. C. P. Woodcock, "Calculating Properties of Z Specifications," *Software Engineering Notes*, vol. 14, no. 5, pp. 43-54, 1989.

[Zave84] P. Zave, "The Operational Versus the Conventional Approach to Software Development," *Communications of the ACM*, vol. 27, no. 2, pp. 104-118, 1984.

[Zave82] P. Zave, "An Operational Approach to Requirements Specification for Embedded Systems," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 3, pp. 250-269, 1982.

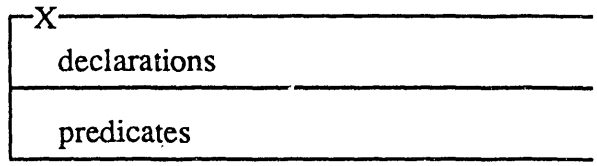
[Zelk78] M. V. Zelkowitz, "Perspectives on Software Engineering," *ACM Computing Surveys*, vol. 10, no. 2, 1978.

Appendix: An Example Formal Specification

This appendix presents an example of a formal specification using Z. The specification is for a symbol table, a basic data structure, and is taken from [Wing90]. We begin with a presentation of the necessary features of the Z notation. The Z notation is more extensive than indicated here. The features presented are sufficient to understand the example. This is followed by the specification for the symbol table.

A.1 The Z Notation

The Z notation is based on set theory and first-order logic. Z uses *schemas* to group declarations and predicates that constrain the values of the constant or function being declared. The format of a schema declaration is



A declaration specifies the type of the constant or function. The predicates specify constraints on their values.

The Z notation combines elements of the mathematical notation of sets and logic. It also makes use of symbols for functions. The following Z symbols are used in the example. Many more symbols are available in the Z syntax but are not shown here to keep the size of the presentation manageable.

$x \in S$	Membership: indicates that the object x is a member of the set S
$x \notin S$	Non-membership: indicates that the object x is not a member of the set S
$S \cup T$	Set union: denotes the set of all elements which belong to S and T or both
dom f	Domain: denotes the set of values for which the function f(x) is defined
$X \nrightarrow Y$	Partial function: represents the set of partial functions from X to Y (the partial function might be defined on only a subset of X)
$x \mapsto y$	Maplet: denotes that x is related to y or x maps to y
$\{x\} \lhd f$	Domain anit-restriction (subtraction): denotes a function like f except that x is not in its domain
$P \wedge Q$	Logical AND: evaluates to TRUE if both P and Q are TRUE
$S' = S$	No elements of S are changed. By convention, if S denotes the state of set S before an operation, S′ denotes its state after the operation.

A.2 Example

The following example illustrates the specification of a symbol table using Z. A symbol table is a table in which entries are found by providing keys that correspond to the values of the entries. The symbol table may be viewed as a mapping from keys to values. A number of different operations may be performed on a symbol table, including initializing the table, inserting items into the table, looking-up items in the table, and deleting items from the table.

$ST = KEY \pfun VAL$

INIT

$st' : ST$

$st' = \{\}$

INSERT

$st, st' : ST$
$k : KEY$
$v : VAL$

$k \notin dom(st) \land st' = st \cup \{k \mapsto v\}$

LOOKUP

$st, st' : ST$
$k : KEY$
$v : VAL$

$k \in dom(st) \land v' = st(k) \land st' = st$

DELETE

$st, st' : ST$
$k : KEY$

$k \in dom(st) \land st' = \{k\} \ndres st$

The first line defines the type symbol table (ST) as a partial mapping from a set of keys (KEY) to a set of values (VAL). The partial mapping indicates that a given key may not have a corresponding value currently in the table. INIT initializes an instance of the symbol table (st) to be empty. INSERT adds a new element to the table, provided that its key, k, is not already in the domain of st. LOOKUP returns the value corresponding to k, provided that k is in the domain of st. DELETE removes the value corresponding to k from the table, provided that k is in the domain of st.

DATE
FILMED
2/15/94
END