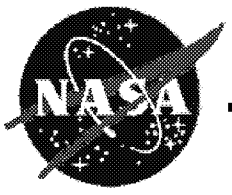
OFFICE OF SAFETY AND MISSION ASSURANCE

NASA-GB-002-95
RELEASE 1.0

FORMAL METHODS SPECIFICATION AND VERIFICATION GUIDEBOOK FOR SOFTWARE AND COMPUTER SYSTEMS

VOLUME I: PLANNING AND TECHNOLOGY INSERTION

JULY 1995



NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
WASHINGTON, DC 20546

FORMAL METHODS SPECIFICATION AND VERIFICATION GUIDEBOOK FOR SOFTWARE AND COMPUTER SYSTEMS

VOLUME I: PLANNING AND TECHNOLOGY INSERTION

FOREWORD

The Formal Methods Specification and Verification Guidebook for Software and Computer Systems describes a set of techniques called Formal Methods (FM), and outlines their use in the specification and verification of computer systems and software. Development of increasingly complex systems has created a need for improved specification and verification techniques. NASA's Safety and Mission Quality Office has supported the investigation of techniques such as FM, which are now an accepted method for enhancing the quality of aerospace applications. The guidebook provides information for managers and practitioners who are interested in integrating FM into an existing systems development process. Information includes technical and administrative considerations that must be addressed when establishing the use of FM on a specific project. The guidebook is intended to aid decision makers in the successful application of FM to the development of high-quality systems at reasonable cost. This is the first volume of a planned two-volume set. The current volume focuses on administrative and planning considerations for the successful application of FM. Volume II will contain more technical information for the FM practitioner, and will be released at a later date.

Major contributors to the guidebook include, from the Jet Propulsion Laboratory: Rick Covington (editor), John Kelly (task lead), and Robyn Lutz; from Johnson Space Center: David Hamilton (Loral) and Dan Bowman (Loral); from Langley Research Center: Ben DiVito (VIGYAN) and Judith Crow (SRI International); and from NASA HQ Code Q: Alice Robinson. Special thanks go to other contributors and numerous reviewers for their assistance in preparing the guidebook. A special acknowledgment goes to Alice Robinson for her leadership and guidance since the inception of the task.

This document is a product of the NASA Software Program, an agencywide program to promote continual improvement of software engineering within NASA. The goals and strategies for this program are documented in the NASA Software Strategic Plan, July 13, 1995. Additional information is available from the NASA Software IV&V at the World Wide Web site http://www.ivv.nasa.gov.

Office of Safety and Mission Assurance

Formal Methods Specification and Verification Guidebook for Software and Computer Systems

Volume I: Planning and Technology Insertion

Approvals

John C. Kelly, Jet Propulsion Laboratory
Task Lead

Kathryn Kemp
Deputy Director, NASA IV&V Facility

TABLE OF CONTENTS

I. GENERAL

I.1. PURPOSE

Formal Methods (FM) consist of a set of techniques and tools based on mathematical modeling and formal logic that are used to specify and verify requirements and designs for computer systems and software. The use of FM on a project can assume various forms, ranging from occasional mathematical notation embedded in English specifications, to fully formal specifications using specification languages with a precise semantics. At their most rigorous, FM involve computer-assisted proofs of key properties regarding the behavior of the system. Project managers choose from this spectrum of FM options as appropriate to optimize the costs and benefits of FM use and to achieve a level of verification that meets the customer's needs and budget constraints. Experience suggests that these choices are most successful if based on certain managerial and technical considerations, which are the major focus of the guidebook. FM play an important role in many activities including certification, reuse, and assurance. Although the focus of this guidebook is restricted to the role of FM in requirements analysis, much of the discussion is also relevant to these other activities.

I.2. BENEFITS

The growing criticality and complexity of NASA applications and the increasingly prominent role of software in these applications has led to NASA's interest in FM techniques. This interest grows out of concerns such as the following, which can be effectively addressed by the application of FM:

- Fault protection and safety functions can no longer be allocated solely to hardware devices. Software in aerospace applications is needed to detect failures, isolate them, and execute recovery routines.
- Software-intensive systems fail in ways that are characteristically different from hardware components.
- Aerospace systems continue to become more complex, and development of such systems places ever-increasing demands on existing development and verification techniques.
- Organizations exercising existing techniques with a high degree of discipline are experiencing "quality ceilings". In these projects, traditional verification techniques have been improved and fine-tuned to the point that major quality improvements can no longer be achieved, even though some defects still remain in the developed product.
- Although it is desirable to detect problems as early as possible after they are introduced (because problems are cheaper to fix the earlier they are

detected), few existing techniques which are appropriate for early life cycle phases such as requirements and high-level design offer the rigor and automatic support now considered necessary to verify the quality of engineering products during these life cycle phases.

In addition, the development of requirements and design for software systems can be particularly prone to errors, cause costly repairs, and have lasting adverse effects. Studies of current and past software systems show the necessity of building a better foundation for high quality systems during the early phases of the developmental life cycle.

Software continues to play an increasingly prominent and critical role in complex systems. Since development life cycles, failure models, and verification methods that have performed well for hardware systems are not always optimal for systems that include a significant software component, the identification and evaluation of better verification techniques for such systems will be an ongoing need within the systems development discipline. This need, coupled with substantial improvements in FM techniques and tools, have made FM specification and verification a technique for consideration by most projects delivering a product that includes software. FM complement inductive techniques such as testing and help projects move beyond traditional quality ceilings.

The following are some of the benefits realizable from effective applications of FM:

- FM help find defects; as evidence of this, when applied to high-quality software systems, FM have found defects that went undetected during extensive testing [Miller2]. The inductive nature of testing ensures that complex systems will always have scenarios which cannot be tested due to practical considerations.
- Formal specifications allow defects in requirements and designs to be detected earlier than they would be otherwise and greatly reduce the incidence of mistakes in interpreting and implementing correct requirements and designs.
- Formalized statements can be analyzed and their consequences calculated in a repeatable manner. The risks of drawing conclusions about a system's behavior by extrapolating from a finite number of tests often can be avoided by using proof methods based on mathematics. Such methods allow large (potentially infinite) classes of test cases to be fully covered in a finite proof, and they support reasoning that can be checked by colleagues or by machine, with minimal dependence on subjective reasoning.
- Use of FM causes more defects to be detected than would otherwise be the case and in certain circumstances guarantees the absence of certain defects.

I.3. READER'S GUIDE

This guidebook is written for project decision makers, including managers, engineers, and assurance personnel, who are considering the use of FM on their project. It is intended to be an easily understood overview of important management issues associated with the use of formal specifications and a useful guide to planning and implementing FM on a project. It is presented in a tutorial rather than prescriptive style. The current volume is Volume I of a planned two-volume set. Volume II will contain detailed information for technical practitioners of FM, and will be released at a later date. The second volume will also address the needs of technologists whose role it is to evaluate new technologies, to transfer those technologies into practice in their organization, and to help projects in planning, training, and implementation.

FM offer significant potential for improving defect detection early in the life cycle. The guidebook is appropriate for candidate projects that use defect prevention techniques such as formal inspections. The reader should be aware that FM require commitment and a disciplined approach. This guidebook will make it easier to start a serious investigation of how appropriate FM are for a specific environment.

This guidebook includes some basic FM concepts and definitions. It illustrates how FM facilitate the precise modeling of requirements and high-level design using specifications based on the notations of discrete mathematics. FM also support automated consistency checking and testing specifications by proving key properties. The guidebook summarizes specific FM tools and languages in Appendix B.

The use of formal specifications and proofs is not an "all-or-nothing" approach. One can tailor the use to the level of rigor appropriate to specific budget, schedule, and technical needs. This guidebook discusses the tailoring necessary to integrate FM into an existing development process and the tailoring to establish FM on a specific project. It also discusses how to gain experience by applying FM on a relatively small trial project before committing to wider project use. FM consist of many techniques that are applied to different application domains in different ways.

This guidebook addresses the many benefits of FM, from enhancing the likelihood of a correct implementation to finding more defects through consistent, repeatable, and effective analysis. These benefits are directly related to the use of precise unambiguous specifications and proofs supported by computer-based tools. There are also indirect benefits. FM help engineers focus on what a system should accomplish instead of how to accomplish it. FM enhance existing review processes by encouraging rigorous arguments of why and in what ways the specification is correct. Perhaps the biggest benefit

is that FM are applicable to any life cycle phase, including the early phase where a significant need currently exists for better analysis approaches.

Formal methods offer tangible benefits, but are not a panacea. FM have their own limitations and potential pitfalls. This is precisely why this guidebook has been developed: to help an organization reap the benefits and avoid the pitfalls. In particular, the guidebook is intended to help a project choose the level of FM appropriate for its schedule, budget, development environment, and application domain. In the end, the reader will see that FM have demonstrated unique capabilities that complement and go beyond existing testing and analysis approaches.

I.4. ORGANIZATION OF THE GUIDEBOOK

The organization of the rest of this guidebook is as follows. In Section II, we introduce FM concepts and definitions. In Section III, we discuss how to integrate FM techniques into the systems development process, followed in Section IV by a detailed discussion of factors relevant to establishing FM on a project. Section V provides an overview of FM tools and techniques. Finally, we provide a summary and conclusions in Section VI. Case study information on several small applications of FM to NASA pilot projects is included in Appendix A. Appendix B offers a comprehensive list of FM tools and more detailed descriptions of the most widely used tools.

II. CONCEPTS AND DEFINITIONS

II.1. CONCEPTS

Formal Methods (FM) refer to the use of techniques from formal logic and discrete mathematics in the specification, design, and construction of computer systems and software. FM allow the logical properties of a computer system to be predicted from a mathematical model of the system by means of a logical calculation, which is a process analogous to numerical calculation. That is, FM make it possible to calculate whether a certain description of a system is internally consistent, whether certain properties are consequences of proposed requirements, or whether requirements have been interpreted correctly in the derivation of a design. These calculations provide ways of reducing or in some cases replacing the subjectivity of informal and quasi-formal review and inspection processes with a repeatable exercise. This is analogous to the role of mathematics in all other engineering disciplines; mathematics provides ways of modeling and predicting the behavior of systems through calculation. The calculations of FM are based on reasoning methods drawn mainly from formal logic. Systematic checking of these calculations may be automated.

Formal modeling of a system usually entails translating a description of the system from a non-mathematical model (data-flow diagrams, object diagrams, scenarios, English text, etc.) into a formal specification, using one of several formal languages. This results in a system description that possesses a high degree of logical precision. FM tools can then be employed to logically evaluate this specification to reach conclusions about the completeness and consistency of the system's requirements or design. Manual analyses (e.g., peer reviews) of the formal model are used as an effective first check to assure the general reasonableness of the model. These are followed by tool-based analyses, which raise the level of reliability and confidence in the system specification even further. FM analysis techniques are based on deductive rather than inductive reasoning about system descriptions, allowing entire classes of issues to be resolved before requirements are committed to the design and implementation phases. FM complement the inductive testing that follows implementation by allowing the testing phase to focus on a potentially smaller or more problematic range of test cases.

FM techniques and tools can be applied to the specification and verification of products from each development life cycle: requirements, high-level and low-level design, and implementation[1]. The process of applying FM to

[1]Cost-benefit analysis generally favors FM applied to early life cycle phase products (requirements and high-level design).

requirements or design differs mainly in the level of detail at which the techniques are applied. These techniques include: writing formal specifications, internal checking (e.g., parsing and type correctness), traceability checking, specification animation, and proof of assertions. Although this entire suite of techniques could be applied to all requirements and design elements, this is not the usual approach. Instead, an important subset of the requirements is chosen to undergo FM, then a subset of the techniques is chosen for application. This enables the project to choose a level of verification rigor appropriate to its budget, schedule, and to the development team's technical needs.

In addition to the function FM perform within a single development life cycle phase, FM can also be used to establish and maintain strict traceability between system descriptions across different life cycle phases. We can think of a hierarchy of system description documents, each of which describes the system at a different level of detail. Moving from the most abstract to the most concrete, there are requirements, high-level design, low-level design, and implementation. These documents also correspond to different life cycle phases. FM can be used to demonstrate that a property at some level in the hierarchy gets implemented correctly by the next-lower level. In a thorough and rigorous treatment, FM can help demonstrate that requirements are correctly reflected in a subsequent design and that design features are correctly reflected in a subsequent implementation.

II.2. DEFINITIONS

The following are working definitions for basic terms and concepts discussed in this guidebook.

A **formal specification** is a concise description of the behavior and properties of a system written in a mathematically-based language, specifying what a system is supposed to do as abstractly as possible, thereby eliminating distracting detail and providing a general description resistant to future system modifications. The most formal specifications are written in a language with a well-defined semantics that supports formal deduction and allows the consequences of the specification to be calculated through proof of putative theorems.

A **formal proof** is a complete and convincing argument for the validity of a statement about a system description. A proof proceeds in a series of steps, each of which draws conclusions from a set of assumptions. Justification for each step is derived from a small set of rules which state what conclusions can be reasonably drawn from assumptions. Such justification eliminates ambiguity and subjectivity from the argument. Formal proofs may be

prepared manually or, preferably, with the assistance of an automated FM tool.

Abstraction is the process of simplifying and ignoring irrelevant details and focusing, distilling, and generalizing what remains. In FM, abstraction is a tool for eliminating distracting detail, avoiding premature commitment to implementation choices, and focusing on the essence of the problem at hand.

Specification animators (also called emulators) are executable programs which reinterpret a formal specification into a high-level dynamically executable form. Specification animations are not formal in a strict sense, but support the formal requirements and design verification process by providing analysts with an early view of the high-level dynamic behavior of the requirements.

II.3. A FORMAL METHODS EXAMPLE

At this point we introduce a small example to clarify many of the concepts introduced earlier. The example illustrates the use of formal specifications to model a system, to enhance the consistency of the specification, and to suggest the role of proof in establishing desired system properties. The purpose of this discussion is to provide a concrete, albeit small and highly simplified example. Readers interested in a more detailed tutorial discussion should consult [Butler], [Weber-Wulf], and [Wordsworth]. Those interested in more realistic or industrial-scale applications can find excellent discussions in papers, technical reports, and books, including [Miller1] and [Bowen2].

Consider the following typical informal requirements expressed in English:

> A tank of cooling water shall be refilled when its low level sensor comes on. Refilling consists of adding 9 units of water to the tank.
>
> Notes:
> - The maximum capacity of the tank is 10 units of water.
> - From one reading of the water level to the next reading of the water level, 1 unit of water will be used.
> - The low level sensor comes on when the tank contains 1 unit of water or less.

The above statement contains several descriptions, including two key notions: the water level in the tank and the water usage. Formally, these

notions can be modeled as follows (statements 1 and 2):

1 **level** is represented by a restricted integer type: a number between 0 and 10, inclusive
2 **usage** is represented as the integer constant 1

That is, level describes an amount of water that the tank may hold at any point in time and usage describes the amount of water used during one cycle.

The primary requirement is that 9 units of water will be added to the tank whenever the level is less than or equal to 1. This can be more precisely[2] stated as (statement 3):

3 Function **fill** takes, as input, a water level and returns, as output, a water level. Given an input of L units of water, fill returns L+9 if L is one or less, otherwise it returns L.

That is, we claim that fill(L) accounts for any filling of water in the tank.

A commonsense property of this system is that, at the next cycle, the new water level will be the current water level, plus any amount that was added, minus the amount that was used. That is, given L as the current level of water, the level at the next cycle should be given by statement 4:

```
4    level = L + fill(L) - usage
```

One approach to checking this specification is to ensure that each reference to a level of water is consistent with the definition of level, i.e., it should always be a number between 0 and 10. It turns out that the specification for fill given in 3 above is consistent with the definition of level if the following two logical statements are true:

```
5    FORALL levels L
          (L <= 1) IMPLIES THAT
               (0 <= L + 9) AND
               (L + 9 <= 10)

6    FORALL levels L
          (0 <= L + fill(L) - usage) AND
          (L + fill(L) - usage <= 10)
```

(Statements 5 and 6 can be derived straightforwardly by means of FM techniques. Many FM tools can produce such expressions automatically from

[2] This specification is given in a form of structured English so that the reader can easily follow it without having to learn a formal specification language. Such specifications are more precise than those written in conversational English but are still less precise than those written in a formal specification language.

a set of system definitions.) The following statements (statements `5.1` and `5.2`) constitute an informal proof that the *first* `FORALL` statement (statement `5`) is true:

`5.1` L+9 >= 0 because L >= 0 (and the sum of any two numbers greater than zero is greater than zero)

`5.2` L+9 <= 10 because L <=1 (and any number less than or equal to 1 plus 9 is less than or equal to 10)

However, the *second* `FORALL` statement (statement `6`) is not true. Consider the case when L is 9:

L + fill(L) - 1 = L+L-1 = 9+9-1 = 17 (which is not <= 10)

So clearly, something is wrong. Upon closer examination, it is found that statement 4, our expression for the water level at the next cycle, is in error:

```
4     level = L + fill(L) - usage    (incorrect)
```

This statement is inconsistent with the definition of `fill` because `fill` returns the new level of water, not just the amount of water added. The (corrected) expression for `level`, denoted by 4', is simply:

```
4'    level = fill(L) - usage   (correct)
```

and the (corrected) `FORALL` statement (statement 6) is:

```
6'    FORALL levels L:
            (0 <= fill(L) - usage) AND
            (fill(L) - usage <= 10)
```

This example illustrates the following:

- **Formal Specification**: Modeling informal English statements using mathematical expressions
- **Type Checking**: Checking that all types of items are used consistently (e.g., level)
- **Stating Properties**: Identifying and defining expected behavior of the system (e.g., the expected new level in the tank).
- **Proving Logical Conditions**: Constructing logical proofs which show that a given condition holds under all possible situations.

This example also illustrates how formal analysis can expose errors and inconsistencies in a specification. In the example, the name chosen for the "fill" function in statement 3 is misleading because the function returns the "actual level" rather than the "amount added". Statement 4, although

wrong, is consistent with the casual reader's expectations, so the error is easy to overlook.

In simple cases such as this, an informal inspection of the specification can be expected to find the error. However, the use of FM resulted in a systematic and reproducible approach to uncovering the problem. Similar results can be achieved in challenging industrial-scale specifications, where such errors can be obscured within many pages of requirements.

This example does not show how tools can be used to assist in formal analysis. That topic will be addressed in Volume II of this guidebook.

III. INTEGRATING FORMAL METHODS INTO THE DEVELOPMENT PROCESS

The purpose of this section is to provide guidance on identifying changes necessary to integrate formal methods (FM) into an existing software process.

III.1. PROCESS PREREQUISITES

An effective introduction of FM assumes that a sufficiently well-defined process with the following characteristics has already been established:

- Discrete phases or steps are clearly defined and documented, e.g., requirements phase, high-level design phase, etc.
- Work products are specified for each phase, e.g., requirements document, high-level design diagrams, etc.
- Analysis procedures are established to ensure correctness of work products, e.g., proofs of key system properties.
- Reviews of major work products are scheduled, e.g., design inspections.

A process which lacks these aspects is unlikely to be mature enough to realize substantial benefit from the application of FM. Put somewhat differently, FM is not a "silver bullet" that solves all development problems. For example, quality problems in a new or immature process are more likely to benefit from establishing a well-defined process and including basic defect prevention techniques such as formal inspections. On the other hand, if existing techniques are well-established and performing effectively, then the addition of FM-based strategies can further enhance quality assurance activities.

III.2. WHERE TO ADD FORMAL METHODS

As was pointed out in Section II.1., FM can be applied to any or all phases of the process, although the benefit-to-cost ratio of applying FM seems to be best during the requirements and high-level design phases. FM complement early development phases, which are currently less automated and less tightly coupled to specific languages and notations, and for which work products are typically less effectively analyzed than those of later development stages. FM compensate for these limitations without intruding on the existing process. For example, requirements are currently maintained as English language statements that are hard to check with automated tools. This deficiency is mitigated by the systematic, repeatable analysis supported by FM requirements specification and proof, while necessitating no changes to the natural language requirements statements.

As FM are injected into later life cycle phases, integration raises more technically challenging problems and the injection of FM becomes more intrusive. For example, the languages used for FM specification and proof and those used for programming generally exhibit fundamental semantic differences that make it difficult to synthesize a process that effectively uses both. Extreme care is required to ensure that the semantic differences between the formal specification language and the programming language are not a source of ambiguity or other type of error during development.

The best strategy is to apply FM to the earlier life cycle phases where it will have the most positive impact and consider adding it to selected later phases based on the guidance in Section IV. The application of formal specifications at the requirements life cycle phase will help ensure that the resulting software is verifiable. The addition of FM will usually add a certain amount of cost to these phases while saving cost in later phases and during maintenance of the work products. In this respect, the use of FM is similar to other defect prevention techniques such as formal inspections. If heavy emphasis is already placed on analysis of early work products (e.g., requirements), the use of FM could potentially reduce the cost in these early phases by replacing expensive ad-hoc techniques (e.g., manual verification of interface tables) with more effective and systematic ones.

III.3. PROCESS CHANGES

To each phase in which FM is applied, some of the following products and activities may be added:

1. A new analysis activity called "modeling", during which an initial, often graphical, description of the relationship between system entities is proposed. Various methodologies (finite-state machines, object-oriented design, etc.) are possible.
2. A new development activity called "formalization" during which the formal specification is created.
3. A new type of work product called a "formal specification". This can be a separate product or an addition to an existing work product such as a requirements document.
4. A new analysis activity called "specification animation" (defined in Section II.2.) to better understand the behavior implied by the formal specification.
5. A new analysis activity called "proving assertions" (see Section II for details) to enhance the correctness of the formal specification and to understand the implications of the design captured in the requirements and specification.

6. A review of the formal specification to check the coverage, "correctness," and comprehensibility of the formal specification.
7. An enhancement of traceability tools and techniques to track new products such as formal specifications and proofs, and their relationships to existing products.

While the above activities can be broad in scope, they pose no significant technical challenge. Additions 1-3 and 7 are typically a minimal set, while additions 4-6 are optional. Consult Section IV for guidance on integrating additions 4-6.

III.4. ORDERING OF ACTIVITIES

There is no rigid ordering of the activities for FM; in fact, an iterative approach is the most effective for developing and analyzing specifications. Reviews can be productive at any point after the specification is reasonably well-developed, either before or after key properties have been proved. At a minimum, a review should be held after the specification is complete. If an extensive set of assertions are to be proven after the initial specification review, a subsequent review will be useful to assess the adequacy of the proven assertions, and to motivate discussion of changes to the specification, if any, that might have been introduced to support the proofs.

III.5. SAFETY ANALYSIS

Standard analysis focuses on functional correctness, i.e., behavior that the system should exhibit. Safety analysis generally focuses on behavior that the system should not exhibit because it would create an unsafe or hazardous condition, e.g., the system should not send an erroneous command or fail to respond in a timely fashion. Safety analysis requires looking at a work product from a safety point of view, and can be combined with traditional analysis or performed as a separate activity. Analysis techniques for software safety are currently not as well-defined as those for hardware safety, but FM complement existing techniques by providing methods for stating and analyzing safety properties. More specifically, FM provide a way of stating functional correctness and safety properties within a formal specification, and then demonstrating that the specification satisfies the given safety properties.

FM can be used to formalize and automate an existing safety analysis step or to assist and reinforce the addition of a safety analysis step to an existing process.

III.6. MEASURING THE EFFECTIVENESS OF FORMAL METHODS

Little is known about effective metrics for FM [Craigen1, Fenton]. Nevertheless, a mature process should include provisions for collecting data on the effectiveness of FM, or on the effectiveness of any process activity. Due to the iterative nature of the process of specification and proof, it is best to combine the two activities for an assessment of cost-effectiveness. Potentially useful metrics include:

- Number of pages of English description that were used as the basis for the formal specification, along with a subjective indication of their level of detail and completeness (e.g., high, medium, low).
- Number of lines of formal specification produced.
- Amount of time spent in developing specifications, including properties and proofs.
- Number of issues found in the original requirements (i.e., the requirements in their English description form, before being formalized), along with a subjective ranking of importance (e.g., major, minor).
- Amount of time spent in reviewing and in inspection meetings, along with a number and type of issues found during this activity.
- Number of issues found after requirements analysis, along with a description of why the issue was not found (e.g., inadequate analysis, outside the scope of the analysis, etc.)

IV. ESTABLISHING FORMAL METHODS ON A PROJECT

The previous section provided a general discussion of the impact of introducing and integrating formal methods (FM) into the development process. In this section, we move to more specific considerations that should be reviewed each time FM are proposed for a given project. There are basically two types of considerations, one of which is largely administrative, the other largely technical. A summary of each appears below.

Administrative Factors:

- Project Staffing: The team responsible for planning the role of FM on a project should include at least one person knowledgeable in FM and one person knowledgeable about the application domain. The team responsible for applying FM must have FM expertise or be provided with hands-on training.
- Project Scale: The scale of the project should be taken into consideration. If project staff has little or no previous FM experience, an initial study may be advisable either as a final objective or as a lead-in to the full-scale project.
- FM Training: The training available to those project staff responsible for applying FM should be rigorous and include hands-on experience with the tool(s) and type of application that will be encountered on the project.
- Process Integration: The strategy for integrating FM into a new or existing process should be thoroughly planned and documented, preferably early in the project.
- Project Guidelines: Project guidelines, standards, and conventions, both for documentation and specification, should be developed early and adhered to.

Technical Factors:

- Type of Application: FM are not equally appropriate for all applications; they are best suited to analyzing complex problems, taken singly and in combination, and less suited for numerical algorithms or highly computational applications.
- Size and Structure of Application: The size and structure of an application determine the difficulty of using FM; ideally, applications should be of moderate size (guidance on how to assess size will be addressed in this item's section below), decomposable into subsystems or components, and based on a coherent underlying structure.
- Type of Analysis/Formal Method: The type of analysis, i.e., the reasons for applying FM, determine the most appropriate level of formalization and the most suitable FM and FM tools. Objectives in using FM range from producing clear, unambiguous documentation to

mechanically verifying the correctness of crucial algorithms or components.

- Levels of Rigor in FM: FM may be applied at varying levels of rigor. The rigor, or extent to which a method is "truly formal" and "really calculates," can range from the occasional appearance of mathematical notation in an otherwise informal document, through "rigorous" methods that employ a standardized specification language, to "fully formal" methods that make use of mechanically-checked theorem proving.
- Scope of Formal Method Use: There are at least three dimensions to the scope of formal method use: (1) all/selected stages of development life cycle, (2) all/selected system components, (3) full/selected (system) functionality.
- Type of Formal Method Tool: The choice of FM tool, if any, should be directly determined by the application profile generated by evaluating the five preceding factors. Primary considerations include the type of specification language and the need for mechanical proof support.

These administrative and technical considerations are, of course, closely coupled, each having implications for the other. This is particularly true because the process of determining whether a given application is a good candidate for FM is not cut and dried and because the use of FM entails a serious technical commitment by project staff and a corresponding commitment to support and invest in the FM activity on the part of management. This discussion offers useful guidance, but can not supplant the judgment that comes with experience, i.e., with diligent practice and accumulated expertise.

The discussion is organized as follows. Section IV.1. provides a more detailed account of the administrative considerations listed above, Section IV.2. similarly elaborates the technical considerations, Section IV.3. collapses the administrative and technical considerations into a generic plan, Section IV.4. sketches cost considerations, and Section IV.5. summarizes general caveats with respect to FM use.

IV.1. ADMINISTRATIVE CONSIDERATIONS

FM offer significant potential for improving system and software analysis on many types of projects. The adoption of FM requires careful planning and management, ideally including a planning activity that addresses the five administrative considerations introduced previously and discussed further in the following paragraphs.

Project Staffing Construction of a successful plan for using FM on a project requires the participation of people with the right combination of skills --

people with FM expertise and people with project domain knowledge. FM skills are required to ensure that suitable applications are paired with effective tools, and domain knowledge is needed to identify candidate applications. It will not be possible for people with domain knowledge to learn FM or for people with FM knowledge to learn the application domain during the initial planning period; the transition staffing plan should include at least one FM lead and one key project lead to head the planning phase.

After the initial planning phase, staffing for project execution must take into account the discipline and commitment required for effective FM use. It is also essential to identify domain and FM leads willing and available to act as project advisors and to field the questions about tools, strategies, domain issues, etc. that inevitably arise during formalization and proof.

Project Scale When FM are applied to a project for the first time, it may be advisable to use FM on a scale less than the entire project, i.e., to define an initial study. Although many FM pilot projects have been performed, a project may choose to perform its own study

- as a training exercise,
- to better understand what parts of the system will most benefit from FM use,
- to learn what types of FM are most suitable for project use, or
- to validate the feasibility of using FM in the given project environment.

By performing one or more small trial studies, the project can introduce a few key people to FM and demonstrate that FM do indeed produce benefits in the given environment. People introduced to FM on the trial study can later serve as sources of expertise for this and subsequent projects, providing moral support as necessary. Support and consultation from peers and colleagues have been shown to be one of the most effective strategies for introducing new techniques and systems (a "product champion" approach).

Project Training Effective FM use requires staff with existing FM expertise or a management commitment to rigorous, hands-on training that includes exposure to the tool(s) and type of application(s) that will be encountered on the project. It is not realistic to expect untrained project staff to make significant use of sophisticated specification languages and mechanical theorem checkers. The amount of training required depends on the person's technical background, as well as predictable traits such as discipline, perseverance, willingness to experiment, ability to assimilate new knowledge quickly, etc. The level of training required also varies depending on project responsibilities; staff responsible for writing and

analyzing specifications will require more training than staff using specifications largely as documentation. The fact that some FM are easier to learn and use than others will also affect the level of training required. If FM expertise is not available within the project, expertise may need to be brought in for training purposes and retained during the early phases of the project.

Process Integration If the existing process includes defined requirements analysis steps and reviews, the integration of FM will probably involve little, if any, change to the established process; FM can generally be effectively inserted at relevant points in the existing process. For example, formal specifications can be used to complement or replace the existing documentation used to conduct formal or quasi-formal reviews. If the existing process is new or not well established or defined, the process itself, as well as the integration of FM, should be explicitly planned and documented. A possible exception is the integration of FM on a pilot project, in which case process definition and documentation may follow, rather than precede the project. Specific process considerations are discussed in Section III above.

Project Guidelines Writing specifications in a language designed to support FM is analogous to writing programs in a conventional programming language; the same considerations of configuration management, language conventions, reusable modules, standards, and documentation apply. As in the conventional software domain, such guidelines are most effective if they are in place before the project (including training) begins. From an administrative perspective, the benefits of timely, well-established guidelines are improved project communication and productivity; sharing and reuse of specifications is one of many possible benefits realizable in the context of explicit project guidelines.

IV.2. TECHNICAL CONSIDERATIONS

FM cover a wide range of techniques that have different characteristics and utility. In this section, we discuss the scope and implications of these differences with respect to five technical factors that should be evaluated when considering the use of FM for a given application. The factors are introduced in the suggested order of consideration; e.g., before choosing a formal method tool, it is important, first, to define the type and scope of application, second, to specify the type of analysis to be performed, and third, to determine the rigor and scope of the analysis.

Type of Application FM are not equally suitable for all types of applications. Although, in principle, the methods can be applied to nearly any application, in practice, the benefits that can be realized and the difficulty

of achieving them will differ significantly from one application to another, and from one subsystem to another within a single application. Suitability should be evaluated with respect to the characteristics of the problem domain and their implications for the modeling domain.

Higher complexity applications stand to gain from FM much more than lower complexity ones simply because less complex problems can be solved dependably using less rigorous methods. Of particular interest are problem domains whose complexity stems not so much from the size and structure of the design, but from inherently difficult algorithms such as those for fault tolerance and parallel or distributed processes.

A further consideration is the mathematical domain of discourse. Applications that are heavily based on numerical processing, especially those using floating point arithmetic, pose some difficulties for FM[3], while those that can be modeled using the domains of logic and discrete mathematics benefit from easier formalization, more tractable reasoning, and better FM tool support.

Size and Structure of Application The size of an application is a major factor in the cost and difficulty of its formalization. To make the issue of size more concrete, consider the experience base of industrial software projects that have made serious use of FM with automated tool support. A common measure of application size used in this environment is thousands of source lines of code (KSLOC). For design-level specification and verification efforts, most of the industrial systems or subsystems have been in the neighborhood of tens of KSLOC in size, with an upper limit of perhaps 100 KSLOC. For code-level verification, which is less commonly employed and usually limited to R&D efforts, the sizes have been under 10 KSLOC [Polak, Smith]. For applications using less rigorous FM, i.e., those lacking tool support and limited to formal specifications only, there have been efforts in the hundreds of KSLOC range[4].

Due to considerable variation in the level of detail represented, it is more difficult to get a good measure of size in the case of FM used primarily to model requirements. A reasonable estimate is that requirements analysis efforts have been performed for architectures ultimately expanding into systems on the order of hundreds of KSLOC.

As these figures suggest, FM are most effectively applied to systems or subsystems of moderate size; currently, FM cannot be applied in full to the largest systems implementable using conventional programming

[3]Historically, working with axiomatizations of real numbers to reason with rigor about traditional engineering mathematics has been found to be an awkward and daunting task.

[4] See [Craigen1], Figure 2 on p. 8 of Volume 1.

techniques. An alternative is to limit the scope of the formal method activity to critical properties or components of a very large system, assuming, of course, that the system is decomposable into small or medium-sized subsystems or components with well-defined interfaces. This clean structuring property is vital in any medium- or large-scale application to ensure that the results of separate FM analyses can be combined and valid inferences drawn about the composite behavior of cooperating subsystems.

A second structural property, loosely referred to as structural entropy, is also important. If an application has intrinsically high entropy, i.e., is primarily a random collection of special cases with weak cohesion or few unifying principles, little can be expected from a formalization activity. Conversely, if an application exhibits strong underlying structural principles, well understood and easily expressed in a logically meaningful way, FM can effectively capture and exploit this structure.

Type of Analysis/Formal Method The type of analysis or formal method to be employed is determined largely by project objectives; the purpose for which FM are to be applied should be clearly defined and explicitly documented. For example, one application may use FM primarily to develop specifications for documentation, another may exploit the precision inherent in formally specified requirements to catch errors early in the life cycle, a third may use FM to analyze and assure the correctness of critical properties or algorithms. These equally legitimate objectives have very different implications for the rigor of the formal method analysis and the type of formal method tool appropriate for the project, as discussed below.

Levels of Rigor in Formal Methods FM techniques may be applied at varying levels of rigor. Here, rigor is used in a technical sense to mean the degree of formality of a method, i.e., the extent to which a method formulates specifications in an axiomatic style, explicitly enumerates all assumptions, and reduces proofs to explicit applications of elementary rules of inference. Increasing formality allows the products of FM (i.e., specifications and proofs) to be less dependent on subjective reviews and consensus and more amenable to systematic analysis and replication. (Note that "rigorous" in a broader sense is sometimes used to mean "painstakingly serious and careful", which implies nothing about the level of formality in the mathematical sense used here.) Since it is extremely difficult to be truly formal with pencil and paper (cf., for example, [Rushby]), increasing formality is usually associated with increasing dependence on mechanical support.

Listed in order of increasing formality and effort, a suggestive guide to levels of rigor includes:

1. Use of manual review and inspection (e.g., "structured walk-throughs" and "formal inspections") [Fagan1, Fagan2, NASAGB1, NASAGB2, Weller], relying on documents written in a natural language, pseudocode, or programming language, possibly augmented with diagrams and equations, and validated with conventional testing techniques. Activities at this level are not "formal" in a strict sense, but represent current recommended practice, and serve as a baseline of discipline and structure necessary to support the additional activities at higher levels of formality.
2. Use of notations and concepts derived from logic and discrete math to develop more precise requirements statements and specifications. Proof, if any, is informal. This level of FM typically augments existing processes without imposing wholesale revisions. Examples include the "A7" or Software Cost Reduction (SCR) methodology [vanSchou, Heninger] and various case and object-oriented modeling techniques [Rumbaugh] and Mills and Dyer's Cleanroom methodology [Dyer, Mills], although the latter is an exception in that it supplants rather than augments existing processes.
3. Use of formalized specification languages with mechanized support tools ranging from syntax checkers and prettyprinters to typecheckers. This level of formality usually includes support for modern software engineering constructs, e.g., modules, abstract data types, and objects, all with explicit interfaces, but has not historically offered mechanized theorem proving.[5] Examples include Larch [Guttag], RAISE [Nielsen], VDM [Jones], and Z [Spivey].
4. Use of fully formal specification languages with rigorous semantics and correspondingly formal proof methods that support mechanization. Examples include HOL, Nqthm, PVS [Owre], Eves [Craigen2], and SDVS [Cook]. State exploration [Dill], model checking [McMillan], and language inclusion [Kurshan] technologies also exemplify this level, although these technologies are highly specialized, automatic theorem provers that are limited to checking properties of finite-state systems.

Higher levels of rigor are not necessarily superior to lower levels; factors that determine the appropriate level of rigor include: project objectives, criticality of the application, and available resources. For example, if FM are used simply as documentation, Level 2 may be appropriate; if they are used to justify the design of a new and critical component, Level 4 may be the best choice. On the other hand, routine applications adequately handled by conventional processes are probably most appropriately left to

[5]Formal methods are evolving and many of these methods are in the process of migrating "upward" as increasing mechanization occurs. The distinctions in this classification should be interpreted broadly, as a guide to a diverse range of techniques; the characteristics of individual techniques change and need to be reevaluated before use on a given application.

Level 1. Finally, it is possible to use a formal method at a level of rigor lower than its ultimate capability, e.g., by using the specification language, but not the theorem-proving capability of a Level 4 formal method.

Scope of Formal Method Use The extent to which FM are applied can also vary. There are at least the following three dimensions to the notion of extent.

1. All or selected stages of the development life cycle: It is generally felt that the biggest payoff from the use of FM occurs in early life cycle stages, given that errors become more expensive to correct as they proceed undetected through later development stages; early detection leads to lower life cycle costs. Moreover, the use of FM in the early stages provides additional precision where it is currently most needed in the conventional development process.
2. All or selected system components: Criticality assessments, assurance considerations, and architectural characteristics are among the key factors used to determine which subsystems or components to analyze with FM. Since large systems are typically composed of components with widely differing criticalities, the extent of formal method use should be dictated by project-specific criteria. For example, a system architecture that provides fault containment for a critical component through physical or logical partitioning provides an obvious focus for FM activity and enhances its ability to assure key system properties.
3. Full or selected system functionality: Although FM have traditionally been associated with "proof of correctness," i.e., ensuring that a system component meets its functional specification, they can equally well be applied to only the most important system properties. Moreover, in some cases it is more important to ensure that a component does not exhibit certain negative properties or failures, rather than to prove that it has certain positive properties, including full functionality.

These are the three most commonly used variations on the extent of FM application, although others are certainly possible. Varying the degree of rigor along each of these three dimensions yields a wide range of options and provides maximal benefit from a limited investment in FM.

Type of Formal Method Tool The choice of tool is dictated by the application profile defined by consideration of all of the preceding factors, although the issue of tools is clearly moot if the most appropriate level of rigor falls below Level 3. For example, Level 3 documentation of sequential components is consistent either with a typical Level 3 notation supported by a typechecker, or, if more powerful mechanization and stronger guarantees of consistency are desired, with a system normally used to support Level 4. Similarly, when choosing a Level 4 tool, the capability of the tool, the constraints of the problem domain, and the objectives of the

analysis must be well matched. For example, verifying the correctness of fault-tolerant algorithms is probably best pursued with a general-purpose theorem prover, while exploring the properties of mode-switching or other complex control logic is probably more effectively pursued with a state-exploration system.

The process of selecting a formal method tool is in many ways similar to selecting any other software system; the usual considerations of documentation, tutorials, history of use, ease of use, etc. apply. In this case, effective support for the selected formal method(s) is also important. A suggestive, but by no means exhaustive, list of the additional considerations necessary for judicious tool selection appears below.[6] These considerations are largely technical in nature, and the reader new to FM may wish to skip to Section IV.3.

- Specification Language: Is the language adequately expressive for the given application and which of the following features important for the application does the language offer: well-defined semantics, modern programming language constructs (including support for abstraction, modularity, and encapsulation), familiar and convenient syntax, strong typing, encapsulation, parameterization, built-in model of computation, executable subset or other provision for animating specifications, support for state exploration, model checking, and related methods?
- Theorem Prover: Does the FM tool offer a theorem prover or proof checker? If so, how is the theorem prover controlled and guided; is there automated support for arithmetic reasoning, efficient handling of large propositional expressions, and rewriting; what support is there for developing and viewing the proof; can lemmas be used before they are proved and can new definitions be introduced and existing definitions modified during proof; how is the proof presented to the user (e.g., user input or canonical expressions, with or without quantifiers); are the foundations (i.e., all axioms, definitions, assumptions, lemmas) of the proof identified; are there facilities for editing proofs; is it reasonably easy to reverify a theorem after slight changes to the specification?
- Utilities: Does the formal method offer a reasonably comprehensive library of standard types, functions, and other constructions and is the library validated; what, if any, editing and document preparation tools does the system provide; are there facilities for cross-referencing, browsing, and requirements tracing; is there support for incremental

[6]A more detailed discussion of these and other considerations can be found in [NASAFAA, pp. 154-173] Technical aspects of tool selection will be discussed in detail in Volume II of this guidebook.

development across multiple sessions and for change control and version management?

IV.3. INTEGRATING TECHNICAL AND ADMINISTRATIVE CONSIDERATIONS

The two preceding sections discussed administrative and technical factors that should be evaluated when considering the use of FM. In this section, these two types of factors are integrated into a generic plan summarizing the steps involved in establishing FM on a project. The plan is presented in tabular form and includes 8 steps listed (from top to bottom) in chronological order.

Planning Step	Notes
Identify FM and Application Domain Expertise	FM & Application Expertise Essential
Define Scale	Trial, Partial or Full Scale Project?
Choose Application	Application type, available personnel, etc.
Select Methods	Use FM Expertise to identify suitable FM
Select Tools	Consider application type, human & system resources
Implement Training	View training as an investment
Develop Project Guidelines	Considerations analogous to those for conventional software
Track & Document Process Changes	Update & revise process & documentation with project feedback

For additional information on general issues of technology transfer, see also [Davis] and [Potts].

IV.4. COST CONSIDERATIONS

Data collected in several pilot projects [NASAFM] show that the act of formally stating specifications is generally cost-effective. For critical applications, the act of proving key properties also appears to be cost-effective, although there is less data to support this claim. Due to the difficulty of using statistical techniques to analyze software engineering methods [Fenton], reliable data on FM cost and effectiveness is hard to come by, although available data strongly suggests that judiciously applied FM are cost-effective.

Given the above, prudent advice to projects would be the following. In the context of a stable, controlled software process that includes an emphasis on quality assurance in the requirements phase[7], generate a formal specification for a core subset of important requirements. Conscientiously and

[7]Without a stable software process and a commitment to ensuring correct requirements, it is not clear that the use of FM or any other analytical approach will result in significant benefits to the project.

competently performed, this formalization step will yield tangible, cost-effective benefits. Next, identify the most critical parts of the core requirements and (1) define, (2) formalize, and (3) prove key properties of these critical system components or algorithms. Iterate the specification and proof steps until the process shows noticeable signs of diminishing returns. This point is reached when either no further requirements issues/problems are detected or a large increase in effort is required to carry out the next step.

The only way to ascertain that the money spent on FM is indeed well spent is to collect cost-benefit data for the given application. As with any method, the cost-effectiveness of using FM depends on the characteristics of the project, the productivity of the staff, the nature of the work environment, and the available resources. All of these factors vary over time; regular sampling and analysis of data and overall system quality with respect to cost considerations are strongly advised.

One additional factor should be considered when evaluating cost, namely, the potential effect of reuse. Like software development itself, FM can benefit greatly by reusing assets. Abstract specifications and general theories can be reused on other parts of the same project or in entirely different projects. This is especially true when mechanized forms of FM are employed. If FM are approached with a view toward future reuse of specifications, significant cost savings can be realized in subsequent efforts and amortized over a long period. This effect is more pronounced than a mere learning-curve phenomenon. It stems from an emphasis on generic modes of expression encouraged by the formalization process. Only experience can determine how much of a factor reuse will be, but its potential should be recognized from the outset.

IV.5. FORMAL METHODS LIMITATIONS

FM do not guarantee a superior product. As with all tools, the potential benefits of FM can be realized only if the tools are judiciously applied to suitable applications. FM may provide less benefit than anticipated due to anomalies such as the following.

- Erroneous specifications: Writing formal specifications, like writing correct programs, requires dedication and attention to detail. In both cases, an informal requirement must be turned into something that can be mechanically interpreted, with the potential for undetected gaps, misconceptions, or defects in the informal requirements, or for misinterpretations or erroneous formalizations of correctly stated requirements.
- "Flawed" verifications: In logic and FM, "proof" is a technical term that describes a certain type of symbolic manipulation or "logical calculation."

Like numerical calculations, logical calculations can fail due to: a mistake in the calculation, a specification or system of equations that does not accurately model the real world or the requirements, or a mistake in interpreting the result calculated.

Although these anomalies are not unique to FM, the most effective responses are generally available only through FM techniques. For example, informal as well as formal specifications can be inconsistent, but only FM provide an effective response to these potential problems in the form of typechecking to insure certain forms of internal consistency; theorem proving to challenge the content and implications of the specification; and clear, unambiguous specifications to facilitate peer review.

V. OVERVIEW OF FORMAL METHODS TOOLS AND TECHNIQUES

In this section, we provide a generic description of an automated formal methods (FM) tool. This information is intended to be suggestive, rather than exhaustive; our aim is to provide a starting point that will enable readers to explore the existing literature for further information on general FM techniques as well as specific FM tools.

We begin with a brief classification; FM can be classified according to whether their primary purpose is descriptive or analytic. Descriptive methods focus largely on specification as a tool for review and discussion, whereas analytic methods focus on the utility of specification as a mathematical model for analyzing and predicting the behavior of (hardware and software) systems, in addition to their utility for communication. Not surprisingly, these different emphases are reflected in the type of formal language favored by each of the two methods. Descriptive FM generally use the notations of conventional mathematics, most commonly, notations based on set theory, with quantification restricted to first order functions that are essentially partial, and types imposed on an inherently untyped foundation. These language choices do not readily support automation and descriptive methods typically offer attractive user interfaces and little in the way of deductive machinery. VDM [Jones] and Z [Spivey] are examples of primarily descriptive FM.

Analytic FM place considerable emphasis on mechanization and general design specification languages capable of supporting efficient automated deduction. These methods can be further classified according to the degree of automation provided by the theorem prover, or, conversely, by the amount of user-interaction in the proof process. There are FM systems with automatic theorem proving and virtually no user interaction, FM systems with proof checking and virtually no automatic proof steps, and FM that combine elements of both. Predictably, this spectrum represents tradeoffs with respect to specification language, proof development, and user-accessibility. FM systems that support fully automatic theorem proving typically have restricted specification languages and powerful theorem provers that can be difficult to control and offer little feedback on failed proofs, but perform impressively in the hands of experts, e.g., Nqthm [Boyer]. Most state exploration tools also fall into this category. FM systems based on proof checking generally offer more expressive languages, but require significant manual input for theorem proving, e.g., HOL [Gordon]. FM systems that combine a significant level of automation and user input fall somewhere in between, depending on language characteristics and proof methodology, e.g., Eves [Craigen2], PVS [Owre1].

A typical analytic FM tool consists of the following components:

- User Interface: integrates tool components, manages input and output.
- Parser: checks specifications for syntactic consistency and builds an internal representation used by other components of the system.
- Prettyprinter or unparser: translates the internal representation of the specification into a standard format for user display and output.
- Typechecker: checks specifications for semantic consistency, possibly adding semantic information to the internal representation built by the parser. If the type system of the specification language is not decidable[8], theorem proving may be required to establish the type-consistency of a specification. Systems in which the typechecker and prover are closely integrated attempt to prove type correctness theorems automatically.
- Prover (proof checker): performs proofs over a syntactically and semantically correct specification. As noted above, automated theorem provers differ with respect to level of automation and degree of user interaction.
- Other: Most FM systems also offer some or all of the following:
  - Browser: produces cross-reference and displays cross-reference information; particularly useful for large specifications possibly spread across several files.
  - Status Recorder, Reporter: maintains and reports status of specification (e.g., parsed, typechecked) and proofs (e.g., proof complete or incomplete, axiomatic foundation, status of lemmas used in proof).
  - Output Generator: provides customized and possibly user-customizable formatting for specifications and proofs.

Many toolsets are available to support work in FM. A comprehensive list of tools available as of Spring, 1995, as well as an annotated list of the most widely available, commonly used tools, appear in Appendix B. Most of these tools have been developed in research environments and, consequently, often lack certain features considered standard in the world of commercial software (e.g., features such as sophisticated user interfaces, online technical support, and software maintenance contracts). Nevertheless, most of these tools represent the collective effort of high caliber research teams, some of whom have been refining their tools for nearly 20 years. Many FM tools were created under the sponsorship of government agencies active in the development of software with much higher criticality of requirements than that of most commercial software. Of all the large, publicly available software packages, FM tools are quite possibly the most dependable. As a result, these offerings should be considered viable candidates for project use, provided the users understand they are not dealing with commercial tool vendors and that they need to make appropriate allowances.

[8]Loosely, a problem is "not decidable" if no algorithm or computer program can be described to solve all instances of the problem.

VI. CONCLUSIONS

This guidebook has presented a management overview of formal methods (FM) techniques for systems specification and verification. The overview is based on recent experience in applying FM to real applications (see Appendix A). The sections of the guidebook provide concrete suggestions on how to integrate FM into an existing development process, how to establish FM on a specific project, and what FM tools and techniques are available.

We review below the most important issues to be addressed by managers and systems development personnel contemplating the application of FM to a development process or a specific project:

- What are the key features of FM?
- Does the existing process or intended project meet the prerequisites for application of FM?
- What are the benefits of using FM?

VI.1. KEY FEATURES OF FORMAL METHODS

FM involve the use of logically precise specifications based on discrete mathematics. This type of mathematics is well suited for modeling discrete systems, especially those involving logical interactions. These formal specifications greatly facilitate the modeling of requirements and high level design (modeling of low level design and code is possible, but less cost-effective).

The primary types of analysis supported by FM are checking the internal consistency of a specification and proving that the system specified satisfies desired properties. These types of analyses can be partially automated using computer-based tools that not only support the initial development and analysis of specifications, but also reduce the time required for re-analysis in response to subsequent modifications or extensions.

VI.2. PREREQUISITES

FM are most beneficial when used in a reasonably mature, disciplined environment. Processes that are chaotic, poorly organized or ill-defined and understood, and those that have not yet employed conventional methods such as formal inspections and testing regimes will generally realize less benefit from the application of FM.

Motivation for adopting FM typically arises in environments with a strong quality emphasis, where there exists a commitment to improve system

quality beyond that provided by current practices. Achieving the level of FM mastery necessary to apply them at their most rigorous will require commitment and a disciplined approach. In an environment in which more ad hoc and less systematic approaches are deemed adequate, fully rigorous FM may be perceived as too hard and not worth the effort. Yet even in such an environment, FM applied at a level less than full rigor can, with a reasonable level of effort, make a contribution to overall system quality as a comprehensive checking technique. On the other hand, FM are not appropriate for highly procedural or numerical applications, or for applications that are loosely structured with weak cohesion and few unifying features.

Not all types of applications are equally suitable for FM. Safety-critical systems are generally suitable for FM because they typically satisfy other prerequisites such as a strong quality emphasis and because FM appear to be particularly suitable for analyzing safety properties. Systems involving a high degree of logical interactions (e.g., those with several modes or states determined by Boolean conditions) are well-suited to FM because FM are well-suited for representing logical conditions. Systems required to handle safety or fault protection are particularly suitable for FM because they are both safety critical and involve logical interactions.

Understanding the limitations of FM can help in the choice of which applications and processes are likely to benefit from FM. For example, it is generally not practical to prove an entire system correct. Complete formal proofs have only been achieved for problems of small to moderate size. Managing larger problems requires careful tailoring of the methods or the problem. FM does not eliminate the need for system testing. However, FM can help focus testing and complement the inherent incompleteness of standard testing regimes with an exhaustive analysis, covering all cases. In projects where sufficiently comprehensive testing is too costly or otherwise infeasible, FM may be a viable alternative. Typically, however, FM are used to complement inspections and testing, rather than to replace them.

In summary, positive answers to one or more of the following for a given development environment indicate that FM can make a contribution in that environment:

- Has the development organization achieved most of the elements of a mature process (e.g., the Software Engineering Institute's Level 3)?
- Is greater quality of software subsystems required? Or have there been problems with low quality requirements (or design) in the past?
- Is testing cost and coverage a problem that conventional techniques do not adequately address?
- Would failure of one or more components cause the entire system to fail catastrophically?
- Does the system involve complex switching or multiple modes?

- Is the system required to handle safety or fault protection?

VI.3. BENEFITS OF FORMAL METHODS

- Formal specifications feature a high degree of logical precision which eliminates much of the ambiguity that is found inevitably in informal specifications. This precision translates into a higher likelihood that all requirements writers and readers have a consistent understanding of the requirements and a higher likelihood that the requirements will be implemented correctly. Since formal specifications support abstract descriptions, they help engineers focus on what they want to accomplish instead of how to accomplish it. This may reduce the amount of detail needed in a requirements document.
- Formal proofs eliminate ambiguity and subjectivity from requirements analysis by providing a logical and precise argument for the behavior of the requirements. This enhances the analysis performed in informal reviews and inspections.
- The use of formal specifications and formal proofs provides a systematic, repeatable approach to analysis. This translates into more consistent analysis and a process that is less dependent on the skill and perseverance of a particular analyst.
- The use of formal specifications and proofs is not an all-or-nothing approach. It can be tailored to the level of rigor appropriate to a given budget, schedule, and technical need. That is, it can be scaled to match the needs of a project.
- Formal specifications and proofs can be applied at any life cycle phase, including early in the life cycle where better analysis approaches are currently most needed. Detecting and fixing defects earlier in the process is far cheaper than finding them later in the process. For example, one could tailor the use of specifications and proofs to focus on the verification of critical properties early in the life cycle.
- Formal specification and proofs can be supported by computer-based tools. This provides automation for tasks such as consistency checking and the preparation of proofs. These tools are analogous to the use of automatic calculators (and computers) in the analysis of engineering equations, but rather than "plug in" numbers into a formal specification, one "plugs in" symbolic variables and calculates the equivalent of a closed form solution. This is an important benefit that provides an additional level of assurance as well as reducing the cost of certain aspects of the analysis. These tools greatly enhance the repeatability of the analysis by allowing proofs to be re-executed. This also allows quick answers to the consequences of "What if..." questions early in the developmental life cycle.
- Formal specifications and proofs complement the existing testing approach, but go beyond what testing can accomplish. They complement testing by providing a precise specification from which better test plans can

be derived. They go beyond testing because they have the unique capability to show that key properties are satisfied in entire classes of scenarios.

- There is hard evidence that FM can increase the quality of real systems as well as solve historically difficult problems in computer science. This evidence comes from demonstrations of formal methods on several NASA projects (see Appendix A) as well as from increasing use in commercial systems and other government programs [Craigen1]. FM have been used to find issues in mature requirements and to improve the understanding of complex systems.

In summary, FM enable defects in requirements to be detected earlier than otherwise, and can greatly reduce the incidence of mistakes in interpreting, formalizing, and implementing correct requirements. Furthermore, used early in the life cycle, FM yield formalized statements that can be analyzed and their consequences calculated in a repeatable manner. In addition to these generic benefits attributable to the full spectrum of FM, the most rigorous and fully formal versions of FM cause more defects to be detected than would otherwise be the case and, in certain circumstances, subject to certain caveats, guarantee the absence of certain defects. When used judiciously and skillfully on suitable applications, FM provide compelling evidence of correctness early enough to be useful, cheaply enough to be feasible, and on the basis of modeling that is simple enough to be credible. This guidebook attempts to provide project management the information and insight necessary to make informed decisions concerning the application of FM and to enable them to provide guidance that will allow their projects to realize the benefits of FM use.

REFERENCES

[Ackerman] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski. "Software Inspections: An Effective Verification Process," *IEEE Software*, 6(3):31-36, May, 1989.

[Austin] S. Austin and G. Parkin, *Formal Methods: a Survey*, Division of Information Technology and Computing, National Physical Laboratory, Teddington, Middlesex, UK, March, 1993.

[Bowen1] J. Bowen and V. Stavridou, "Safety-Critical Systems, Formal Methods, and Standards", *Software Engineering Journal*, July, 1993

[Bowen2] J.P. Bowen and M.G. Hinchey, *Applications of Formal Methods*, Prentice-Hall International, Ltd., 1995.

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

[Butler] R.W. Butler, *An Elementary Tutorial on Formal Specification and Verification Using PVS*, NASA Technical Memorandum Number 108991, June, 1993.

[Cook] J.V. Cook, I.V. Filippenko, B.H. Levy, L.G. Marcus, and T.K. Menas, "Formal Computer Verification in the State Delta Verification System (SDVS)", in *AIAA Computing in Aerospace VIII*, pp. 77-87, Baltimore, MD, October, 1991.

[Craigen1] D. Craigen, S. Gerhart, and T. Ralston, *An International Survey of Industrial Applications of Formal Methods*, U.S. National Institute of Standards and Technology, Reports NIST GCR 93/626 (Vols. 1 and 2), March 1993. Also available from the U.S. Naval Research Laboratories, Formal Report 5546-93-9581/9582 (September 1993), and from the Atomic Energy Control Board of Canada, Reports INFO-0474-1 (Vol. 1) and INFO-0474-2 (Vol. 2), January, 1995.

[Craigen2] D. Craigen, S. Kromodimoeljo, I. Meisels, B. Pase, and M. Saaltink, "EVES: An Overview" in S. Prehn and W.J. Toetenel, eds., *VDM '91: Formal Software Development Methods*, pp. 389-405, v. 551 of Lecture Notes in Computer Science, Noordwijkerhout, The Netherlands, October, 1991.

[Craigen3] D. Craigen, S. Gerhart, and T. Ralston, "Formal Methods Reality Check: Industrial Usage", in *Proceedings of Formal Methods Europe '93 (FME'93)*, Springer-Verlag, 1993. Also in *Transactions on Software Engineering*, February 1995.

[Davis] A. M. Davis, "Why Industry often says 'No Thanks' to Research," *IEEE Software* , 9(6):97-99, November, 1992.

[Dill] D. Dill, A. Drexler, A. Hu, and C. Yang, "Protocol Verification as a Hardware Design Aid," *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 522-525, IEEE Computer Society, October, 1992.

[Dyer] M. Dyer, *The Cleanroom Approach to Quality Software Development*, John Wiley and Sons, New York, NY, 1992.

[Fagan1] M. E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal*, 15(3): 182-211, March, 1976.

[Fagan2] M. E. Fagan, "Advances in Software Inspection," *IEEE Transactions on Software Engineering*, SE-12(7):744-751, July, 1986.

[Fenton] N. Fenton, "How Effective Are Software Engineering Methods?," *Journal of Systems and Software*, 22:141-146, 1993.

[Gordon] M.J.C. Gordon and T.F. Melham, eds., *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*, Cambridge University Press, Cambridge, UK, 1993.

[Guttag] J. V. Guttag, J. 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.

[Heninger] 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.

[Jones] C. B. Jones, *Systematic Software Development Using VDM*, 2/e, Prentice Hall International Series in Computer Science, Prentice Hall, Hemel Hempstead, UK, 1990.

[Kurshan] R. Kurshan, *Automata-Theoretic Verification of Coordinating Processes*, Princeton University Press, 1993.

[McMillan] K. McMillan, *Symbolic Model Checking*, Kluwer, Boston, MA, 1993.

[Miller1] S.P. Miller, M.Srivas, *Formal Verification of a Commercial Microprocessor*, Technical Report SRI-CSL-95-4, SRI International, Menlo Park, CA, February, 1995.

[Miller2] S.P. Miller, M. Srivas, "Formal Verification of the AAMP5 Microprocessor -- A Case Study in the Industrial Use of Formal Methods", in *Proceedings of the 1995 Workshop on Industrial-Strength Formal Specification Techniques (WIFT'95)*, IEEE Computer Society, Orlando, FL, April 5-8, 1995.

[Mills] H. D. Mills, M. Dyer, and R. Linger, "Cleanroom Software Engineering," *IEEE Software* , 4(5):19-25, September, 1987.

[NASAFAA] John Rushby, *Formal Methods and Digital Systems Validation for Airborne Systems*, NASA Contractor Report 4551, December, 1993.

[NASAFM] NASA, *Formal Methods Demonstration Project for Space Applications - Phase I Case Study: Space Shuttle Orbit DAP Jet Select*, JPL Document D-11432, December 22, 1993.

[NASAGB1] NASA Office of Safety and Mission Assurance, *Software Formal Inspections Guidebook*, NASA-GB-A302, August, 1993.

[NASAGB2] NASA Engineering Division, *Software Formal Inspections Standard*, NASA-STD-2202-93, April, 1993.

[Nielsen] M. Nielsen, K. Havelund, K. R. Wagner, and C. George, "The RAISE Language, Method and Tools," *Formal Aspects of Computing*, 1(1):85-114, January-March, 1989.

[Owre1] S. Owre, J.M. Rushby, and N. Shankar, "PVS: A Prototype Verification System," in Deepak Kapur, ed., *11th International Conference on Automated Deduction (CADE)*, Saratoga, NY, June 1992, pp. 748-752, v. 607 of Lecture Notes in Artificial Intelligence, Springer-Verlag.

[Owre2] S. Owre, J. Rushby, N. Shankar, and F. von Henke, "Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS", in *IEEE Transactions on Software Engineering*, 21(2), February 1995.

[Potts] C. Potts, "Software-Engineering Research Revisited," *IEEE Software* , 10(5):19-28, September, 1993.

[Polak] W. Polak, "An Exercise in Automatic Program Verification", *IEEE Transactions on Software Engineering*, 5(5), September 1979.

[Rumbaugh] J. Rumbaugh, M. Blaha, W. Pramerlani, F. Eddy, W. Lorenson, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[Rushby] J. Rushby and F. von Henke, "Formal Verification of Algorithms for Critical Systems," *IEEE Transactions on Software Engineering*, 19(1):13-23, January 1993.

[Smith] M. Smith, B. Divito, A. Siebert, and D. Good, "A Verified Encrypted Packet Interface", *ACM Software Engineering Notes*, 6(3), July, 1981.

[Spivey] J. M. Spivey, *Understanding Z, A Specification Language and its Formal Semantics*, Cambridge University Press, 1988.

[vanSchou] A.J. 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.

[Weber-Wulf] D. Weber-Wulf, "Proof-Movie -- A Proof with the Boyer-Moore Prover, in *Formal Aspects of Computing*, 5(2):121-151, 1993.

[Weller] E. F. Weller, "Lessons from Three Years of Inspection Data," *IEEE Software*, 10(5):38-45, September, 1993.

[Wing] J. Wing, "A Specifier's Introduction to Formal Methods," *IEEE Computer*, September, 1990.

[Wordsworth] J.B. Wordsworth, *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering*, Addison-Wesley, Ltd, 1992.

APPENDIX A: FORMAL METHODS CASE STUDIES

This appendix summarizes the results of the application of formal methods (FM) to requirements analysis for several NASA case studies. These applications are drawn from the Space Shuttle, Space Station, and Cassini (JPL) projects. Section A.1. presents a short overview of each case study, including a table at the end that summarizes the goals, cost, and benefits for each study. Section A.2. presents a more detailed report on two of the studies, the Space Shuttle GPS CR (Change Request) study, and the Cassini Fault Protection Software study.

A.1. CASE STUDY DATA

Space Shuttle Jet Select

The first project was the formal specification of a very mature piece of the Space Shuttle flight control requirements called Jet Select. The cost data for this project are very rough because much time was spent organizing the team and learning the toolset. The specification productivity was approximately 10 pages of requirements per work month. Few proofs were produced for the first specification, but 46 issues were identified and several minor errors were found in the requirements. A second specification was produced for an abstract (i.e., high level) representation of the Jet Select requirements. This abstraction, along with the 24 proofs of key properties, was accomplished in under 2 work months, and although it only uncovered 6 issues, several of these issues were significant[9]. Although it would be foolhardy to calculate a productivity rate or error-detection rate based on this data, it is reasonable to conclude that FM, including the proofs, did not take a prohibitively long amount of time and that it did have positive benefits.

Next, two additional specifications were developed for other parts of the Space Shuttle flight control requirements. A high-level formal model of the on-orbit Digital Auto Pilot (DAP) requirements (approximately 200 pages) was developed in approximately 4 work weeks; this model focused on the interfaces between functions. An object-oriented model of the DAP was also developed in 2 work weeks. A goal of this trial project was to see how FM and an object-oriented approach such as Object Modeling Technique (OMT) could be used together. No issues were found (though a large set of redundant requirements was identified), but a much better understanding of how the pieces of the DAP fit together was gained. The study concluded that FM can help the requirements analyst (RA) navigate through a large set of

[9]No error was found in the Space Shuttle requirements that would have affected the software performance during a mission. The issues were significant because they were apparently not previously considered and led to a better understanding of how the software would behave under various off-nominal scenarios.

requirements, focusing on key interface data and logical interactions of components, without significantly increasing the amount of time necessary to analyze the requirements (i.e., 4-6 weeks is comparable to the amount of time it would take a new RA to get a good overview of a subsystem of this size and complexity without FM).

To gain better insight into the cost of proofs, a specification was made of a three-contact, two-switch redundant event triggering system. The goal was to prove that the system, under various assumptions was two fault tolerant. The proof was successful and no errors were found. However, some cases missing from the original analysis were found, some implicit assumptions were documented, and a better understanding of the system was gained. The size of the requirements was about 3 pages. The total cost, including proofs was just over one work week. The overall proof was divided into 15 small pieces each of which took about 10 minutes to prove. See [NASA] and [Kelly] for additional detail.

Space Station FDIR

Next, an assessment was made of the highest level requirements for Failure Detection, Isolation, and Recovery (FDIR) of the International Space Station Alpha (ISSA) U.S. segment. These were system-level requirements which contained a large software content. Fourteen (14) pages of FDIR requirements were specified and analyzed in two-workmonths (2 people working 1/2 time for 2 months); one comprehensive proof was done to assess completeness of the specification. This assessment resulted in 12 issues, including 3 major issues. The requirements development team concurred with all findings and planned to correct the problems in the next release of the requirements document. See [Hamilton] for additional detail.

Space Shuttle GPS CR

A task was undertaken in 1994 involving a Space Shuttle software change request (CR) concerning the integration of new Global Positioning System (GPS) functions. The Shuttle is to be retrofitted with GPS receivers in anticipation of the TACAN navigation system being phased out by the Air Force. Additional navigation software will be incorporated to process the position and velocity vectors generated by these receivers. A decision was made to focus the trial project on just a few key areas because the CR itself is very large and complex. A set of preliminary formal specifications was developed for the new Shuttle navigation principal functions known as GPS Receiver State Processing and GPS Reference State Processing, using the language of SRI's Prototype Verification System (PVS). While writing the formal specifications, 43 minor discrepancies were detected in the CR and these have been reported to Loral requirements analysts.

Cassini CDS FP

Another trial project used FM for the requirements analysis of safety-critical software. The selected applications were the requirements for portions of the Cassini spacecraft's system-level fault protection software. Object-oriented modeling was used to guide the formal specification of the requirements. Fifteen pages of object-oriented diagrams and 25 pages of PVS specifications were produced. Thirty-seven lemmas specified properties essential for the correct and hazard-free behavior of the software. Of these lemmas, 21 were proven to be true and three were disproved using the PVS theorem prover. A total of thirty-seven issues were found in the requirements, including undocumented assumptions, inadequate requirements for boundary cases, inconsistency and traceability issues, imprecise terminology, and one logical error. Appendix A contains a fuller description of the trial projects and their results. See also [Lutz].

Space Shuttle Three Engine Out CR

The Three Engine Out (3 E/O) Task is executed each cycle during powered flight until either a contingency abort maneuver is required or progress along the powered flight trajectory is sufficient to preclude a contingency abort even if three main engines fail. The 3 E/O task consists of two parts: 3 E/O Region Selection and 3 E/O Guidance. 3 E/O Region Selection is responsible for selecting the type of external tank (ET) separation maneuver and assigning the corresponding region index. 3 E/O Guidance monitors ascent parameters and determines if an abort maneuver is necessary. If the ascent phase is nominal, the primary function of this task is to provide display support, indicating the 3 E/O contingency region status. If an abort maneuver is required, 3 E/O guidance switches from display support to an auto guidance steering function that involves calculating and commanding the appropriate maneuvers with respect to external tank separation, post-separation -Z translation, interconnected OMS dump (as necessary), maneuver to entry attitude, and transition to the next abort phase (major mode 602).

We have developed and analyzed a formal model of the series of sequential maneuvers that comprise the 3 E/O algorithm. To date, 20 potential issues have been found, including undocumented assumptions, logical errors, and inconsistent and imprecise terminology. These findings are listed as potential issues pending review by the 3 E/O requirements analyst.

Summary

The costs and benefits are summarized in Table A.1.

Problem	Goals	Cost	Benefits	Notes
Jet Select (low level)	Gain familiarity with Shuttle requirements / FM	Approximately 6 months (very rough estimate), 75 pages of requirements	46 issues raised	most issues were minor, requirements were very mature
Jet Select (high level)	Prove key system properties	Approximated 2 months, 3 pages of high level requirements	6 issues raised	several issues were very significant even though requirements were very mature
Orbit DAP	Compare FM with OMT, investigate internal interfaces	4 weeks (FM) 2 weeks (OMT), 200 pages of requirements	redundant requirements identified, increased understanding gained	requirements were very mature
3 Contact Switch	Prove safety properties	1 week, 3 pages of requirements	uncovered missing cases, clarified assumptions	requirements were very mature
ISSA FDIR	Assess consistency and completeness	2 months, 14 pages of requirements	12 issues	system level requirements with large software content
Cassini fault protection	Requirements analysis of safety-critical software	12 months, 100 pages of requirements	37 issues	system under development

Table A.1.: Summary of Cost and Benefits for Trial Projects

A.2. DESCRIPTIONS OF INDIVIDUAL TRIAL PROJECTS

A.2.1. CASSINI CDS FAULT PROTECTION SOFTWARE

1. Introduction

Another trial project used FM for the requirements analysis of mission-critical software. The selected applications were the requirements for portions of the Cassini spacecraft's system-level fault protection software. This on-board software autonomously detects and responds to spacecraft faults. It was targeted as an application in which the extra assurance possible via formal specification and analysis was merited.

2. Approach

The approach taken in this demonstration project was to:

- Select the application domain. The selected applications were the requirements for portions of the Cassini spacecraft's system-level fault-protection software.
- Model the selected applications using object-oriented diagrams. The object-oriented modeling tool used in this work was Paradigm Plus (registered trademark of Protosoft, Inc.), which is an implementation of OMT, the Object Modeling Technique.

- Develop formal specifications in PVS using the PVS tool set, including the type checker.
- Prove required properties. We determined properties that must hold for the target software to be hazard-free and function correctly, specified them in PVS as lemmas (claims), and proved or disproved them using the interactive theorem-prover.
- Feedback results to the Project. Because we were analyzing requirements that were still being updated, part of our task was to keep current with the changes and to provide timely feedback to the Project as they resolved the remaining requirements issues and began design development.

3. Results

The experiment described here produced 25 pages of PVS specifications and 15 pages of OMT diagrams. 37 lemmas were specified. Of these, 21 were proven to be true and 3 were disproved. An additional 13 lemmas were stated but not proven. Five of these unproved lemmas were obviously true from the formal specifications; four were out of the scope of our application; and four remain to be proven.

The lemmas that were proved or disproved helped in the analysis of whether the specifications were accurate, whether the software could introduce hazards into the system, and whether any hidden assumptions were needed for the software to function correctly.

The results obtained from the specification and analysis (including proofs) of the requirements were of two types, issues found in the requirements and an evaluation of our process of applying FM.

A total of 37 issues were found in the requirements. These were categorized as follows:

- Undocumented assumptions: 11.
- Inadequate requirements for off-nominal or boundary cases: 10.
- Traceability and inconsistency: 9.
- Imprecise terminology: 6.
- Logical error: 1.

The evaluation of the process we used to specify and analyze the requirements led us to three conclusions:

- Using object-oriented models: For the target applications, object-oriented modeling offered several advantages as an initial step in developing formal specifications. First, the object-oriented modeling defined the boundaries and interfaces of the embedded software applications at the level of abstraction chosen as appropriate by the specifiers. In addition, the modeling offered a quick way to gain multiple perspectives on the requirements. They were easy to

understand and review. Finally, the graphical diagrams served as a frame upon which to base the subsequent formal specification and guided the steps of its development. Since the elements of the diagrammatic model often mapped in a straightforward way to elements of the formal specifications, this reduced the effort involved in producing an initial formal specification. We also found that the object-oriented models did not always represent the "why," of the requirements, i.e., the underlying intent or strategy of the software. In contrast, the formal specification often clearly revealed the intent of the requirements.

- Using FM for requirements analysis: Unlike earlier work in this research project on software in which the requirements were very mature and stable and the formal specification entailed reverse engineering (Space Shuttle's Jet Select Subsystem), the work on Cassini's fault-protection subsystem analyzed requirements at a much earlier phase of development. Consequently, the requirements that we analyzed were known to be in flux, with several key issues still being worked (e.g., timing details, number of priority levels). A negative effect of the lack of stability was that time was spent staying current with changes and updating specifications to conform to changes in the requirements. A positive effect was that issues identified during our analysis could be readily fed back into the development process before the design was frozen.

  Based on our experience with this trial project, the formal specification of unstable requirements had the following advantages:
  - Laid the foundation for future work.
  - Allowed rapid review of proposed changes and alternatives.
  - Clarified requirements issues still being worked by elevating undocumented concerns to clear, objective dilemmas.
  - Complemented the lower-level FMEA (Failure Modes and Effects Analysis) already being performed on the software, by providing higher-level verification of system properties.
  - Added confidence in the adequacy of the requirements that had been analyzed using FM.
  - Uncovered issues that might impact the subsequent design phase.

- Using FM for safety-critical software: For a safety analysis it is important to ensure that a hazardous situation does not occur, as well as that the correct behavior does occur. Fault Tree Analysis, which backtracks from a hazard to its possible causes, is one method used for this kind of hazards analysis. However, unlike FM of specification and proof, Fault Tree Analysis is an informal method which in practice permits ambiguous or inadequate descriptions.

FM helped us find hazardous scenarios by forcing us to show every condition and by prompting us to define new, undocumented assumptions through rigorous specification. The process of developing formal specifications and proofs caused us to think about the full range of cases, some of which were unanticipated.

In conclusion, our main contributions to the FM RTOP in the Cassini demonstration project have been:

- Applying FM to the software requirements analysis of a project currently under development,
- Using object-oriented diagrams to guide the formal specification of software requirements,
- Formally specifying and proving a set of properties essential for the correct and hazard-free behavior of the software, and
- Demonstrating that FM can be used to specify and analyze an application involving safety-critical software.

A.2.2. SPACE SHUTTLE GPS SOFTWARE CR TASK

1. Introduction

A task was undertaken in 1994 involving a Space Shuttle software change request (CR) concerning the integration of new Global Positioning System (GPS) functions. The Shuttle is to be retrofitted with GPS receivers in anticipation of the TACAN navigation system being phased out by the DoD. Originally, GPS was required for navigation only during the entry flight phase after the disappearance of TACAN, but the scope has been broadened to cover all mission phases. Additional navigation software will be incorporated in the Shuttle to process the position and velocity vectors generated by these receivers. In particular, the Shuttle GPS software CR will provide the capability to update the Shuttle navigation filter states with selected GPS state vector estimates similar to the way state vector updates currently are received from the ground. In addition, the CR will provide aid to the GPS receivers and will support crew control and operation of GPS/GPC processing.

The goal of this FM task is to develop a model for a core subset of the GPS-extended navigation functions for a single flight phase. Members of the project team are working with requirements analysts from Loral Space Information Systems to derive a set of formal specifications to describe the new requirements and to model interfaces to existing Shuttle navigation functions. This trial project will serve as another opportunity for the Shuttle requirements analysts to evaluate the effectiveness of FM in the requirements analysis process.

2. Approach

A decision was made to focus the trial project on just a few key areas because the CR itself is very large and complex. The general criteria for where to focus were the following:

- Include as much of the new GPS principal functions as possible. It is important to capture the essence of these new functions and their interfaces to existing principal functions.
- Exclude as much of the crew interface functions as possible. We can omit the display and crew input functions for now.
- Focus on entry-phase navigation functions both as the place with the greatest need for GPS as well as the best place to put the archetype model that can be cloned later, if desired, for handling other flight phases.

After preliminary study of the CR and discussions with the responsible RA for this CR, it was felt that the most promising approach was to hone in on the new functions that provide selected GPS state vectors for consumption by the existing entry navigation functions. This means emphasizing the following principal functions:

- GPS Receiver State Processing
- GPS Reference State Processing
- GPS Subsystem Operating Program, except for the portions concerned with low-level I/O processing

If positive results are obtained from this effort, the remaining portion of entry navigation can be undertaken as a follow-on activity.

The FM approach is loosely based on the work conducted during 1993 on Jet Select and Orbit DAP. Those techniques are being adapted as necessary to accommodate the needs of this new area of the Shuttle software. All work is being mechanically assisted by SRI's Prototype Verification System (PVS) toolset.

3. Results

This task began with the Mod C version of the GPS CR. Initially, the relevant portions of the CR were analyzed to determine the basic structure of the principal functions and how they are decomposed into subfunctions. Based on this organization, a general approach for modeling the functions and expressing the formal specifications was devised. A white paper on this prescribed technique for writing formal specifications for the GPS CR was written and sent to the Loral requirements analysts. The formalization of requirements is based on the use of an abstract state machine model. Each principal function is modeled as such a state machine, which takes inputs and local state values and produces outputs and new state values. This method provides a simple computational model that nevertheless accommodates the

key features of a principal function and its software architecture and has a straightforward realization in PVS.

Next, the interfaces of the principal functions and their subfunctions were carefully scrutinized. Particular emphasis was placed on being able to identify the types of all inputs and outputs, and to match up all the data flows that are implicit in the tabular format presented in the requirements. While conducting this analysis and preparing to write the formal specifications, 43 minor discrepancies were detected in the CR and these have been reported to Loral requirements analysts.

A set of preliminary formal specifications was developed for the principal functions known as GPS Receiver State Processing and GPS Reference State Processing, using the language of PVS. Assumptions were made as needed to overcome the discrepancies encountered. Additional detail will be provided to the formal specifications to characterize the functions with adequate precision. In parallel with this activity, a small team of Loral requirements analysts have been learning FM and PVS and positioning themselves to carry out this work after the trial project is completed. The GPS CR task had been on hold pending the outcome of a Space Shuttle decision to shelve the CR, but it has been resumed as of 12/1/94.

A.3. REFERENCES

[Hamilton] D. Hamilton, R. Covington, and J. Kelly, "Experiences in Applying Formal Methods Analysis of Software and System Requirements," *Proceedings of the Workshop on Industrial-Strength Formal Specification Techniques (WIFT'95)*, Boca Raton, FL, April 5-8, 1995.

[Kelly] J. Kelly, R. Covington, D. Hamilton, "Results of a Formal Methods Demonstration Project," *Proceedings, WESCON/94 & Idea/Microelectronics Conference*, Los Angeles, CA, Sept. 27-29, 1994.

[Lutz] R. Lutz and Y. Ampo, "Experience Report: Using Formal Methods for Requirements Analysis of Critical Spacecraft Software," *Proceedings of the Nineteenth Annual Software Engineering Workshop*, NASA Goddard Space Flight Center, Greenbelt, MD, December, 1994.

[NASA] NASA, *Formal Methods Demonstration Project for Space Applications - Phase I Case Study: Space Shuttle Orbit DAP Jet Select*, JPL Document D-11432, December 22, 1993.

APPENDIX B: GUIDE TO INFORMATION ON FORMAL METHODS TOOLS

The following is a comprehensive list of formal methods (FM) tools available throughout the world. Nearly all are available electronically and come free of charge. This list has been compiled (in part) from the FM virtual library maintained on the World Wide Web (WWW) by Jonathan Bowen at the following URL:

http://www.comlab.ox.ac.uk/archive/formal-methods.html

Links to further information about these tools may be followed from the cited home page.

Several of the tools from this list have been described in more detail below for the benefit of the reader. This selection is not intended to be an endorsement of any of these tools, but serves to highlight tools that are better known, better supported, and have been subjected to more widespread use.

Additional information in compiling this list has been drawn from [Craigen] and from several online databases of FM tools:

- The Formal Methods Tools Database maintained by Tim Denvir.
  URL file://chopwell.ncl.ac.uk/pub/fm_tools/fm_tools_db
- The Database of Automated Reasoning Systems maintained by Carolyn Talcott.
  URL ftp://sail.stanford.edu/pub/clt/ARS/README

B.1. A COMPREHENSIVE LIST OF FORMAL METHODS TOOLS

Most of the entries listed here can be found in the FM virtual library at the Oxford University URL cited above. The links from that page can be followed to obtain more detailed information. Those tools not directly accessible via a WWW link from that page are annotated with either an e-mail address for the appropriate contact person or an explicit URL for WWW access.

Acl2 theorem prover, a successor to the Boyer-Moore theorem prover, is under development at Computational Logic, Inc. in Austin, Texas. In progress.

Action Semantics, a framework for specifying formal semantics of programming languages, is being pursued by an international group of researchers, with an archive maintained at the University of Aarhus in Denmark.

Algebraic Design Language, a higher-order software specification language based on algebraic concepts, has been developed at the Oregon Graduate Institute.

ASLAN, a specification language processor/proof obligation generator (email Dick Kemmerer on kemm@cs.ucsb.edu for further details), and **GIL**, a graphical interval logic tool created by Laura Dillon (dillon@cs.ucsb.edu) are available from a formal support tools archive at UC Santa Barbara.

Boyer-Moore theorem prover (a forerunner of **Nqthm**) is available via ICOT Free Software for use under Unix at ICOT (Japan), SICS (Sweden), GMD (Germany) and Univ. of Oregon (USA).

B-Method, developed by Jean-Raymond Abrial (originator of Z) and others, is a formal method for developing software from specifications written in the Abstract Machine Notation. Tool support is provided in the form of an interpreter called the B-Tool and an integrated set of additional tools called the B-Toolkit, both available from B-CORE Ltd., UK. Email Ib.Sorensen@comlab.ox.ac.uk.

Circal (CIRcuit CALculus) is a system supporting a process algebra that may be used to rigorously describe, verify and simulate concurrent systems. It is available from Strathclyde University (UK).

Concurrency Workbench, from the University of Edinburgh, is an automated tool for analyzing concurrent systems using model checking under a variety of different process semantics.

Coq, the Calculus of Inductive Constructions, was developed at INRIA in France by a team led by Gerard Huet. A proof assistant is provided as well as the tool **CtCoq**, a working environment for the Coq theorem prover.

CSP (Communicating Sequential Processes) is a process algebra originally devised by C.A.R. Hoare at Oxford University. CSP is supported by a model checking tool called FDR, developed by Bill Roscoe.

DisCo is specification method for reactive systems including a tool developed at the Tampere University of Technology, Finland. It has a semantics and proof methodology based on the Temporal Logic of Actions (TLA).

Estelle, a formal notation based on an extended state transition model, is supported by EDT (Estelle Development Toolset) and example specifications. Contact estelle-request@cs.umb.edu to join the mailing list.

EVES tool, based on ZF set theory, is from ORA, Canada. See Section B.2 in this guidebook for additional information.

Evolving Algebras, developed at the University of Michigan, is a method that focuses on semantics and seeks to bridge the gap between computation models and specification methods.

Extended ML, a framework for the specification and formal development of modular Standard ML programs, is a method developed at the University of Edinburgh.

HOL is a mechanical theorem proving system. See Section B.2 in this guidebook for additional information.

HyTech, from Cornell University, is an automatic tool for the analysis of embedded systems. It computes the condition under which a linear hybrid system satisfies a temporal-logic requirement. Installation requires a Mathematica license.

IMPS, the Interactive Mathematical Proof System, is intended to provide mechanical support for traditional mathematical techniques and styles of practice. It was developed at the MITRE Corporation in Bedford, Massachusetts.

Isabelle, from Cambridge University, is a generic theorem prover supporting a variety of logics and providing a high degree of automation. See also the Cambridge Automated Reasoning Group and FTP access including an index. Email Larry.Paulson-request@cl.cam.ac.uk for information, including requests concerning the mailing list isabelle-users@cl.cam.ac.uk.

JAPE (Just Another Proof Editor), by Bernard Sufrin and Richard Bornat from Oxford University, is an interactive tool supporting the application of logical reasoning. It is available via anonymous FTP.

LAMBDA toolset from Abstract Hardware Ltd, UK, supports formal verification for hardware/software co-design. Email lamba@ahl.co.uk. To join the usergroup mailing list, email lambda-usergroup-request@dcs.ed.ac.uk.

Larch and **LP** (Larch Prover) support algebraic specification. See Section B.2 in this guidebook for additional information.

LeanTaP, a tableau-based deduction theorem prover for classical first-order logic, is available from the University of Karlsruhe in Germany.

LEGO is an interactive proof checker developed at the University of Edinburgh and based on Standard ML and the Calculus of Constructions.

LOTOS (Language of Temporal Ordering Specifications) is a formal specification technique and process algebra from the University of Twente in the Netherlands.

Maintainer's Assistant, a tool for reverse engineering and re-engineering code using formal methods, was developed at the University of Durham, UK.

Meije tools for the verification of concurrent programs are available from INRIA in France.

Mural, from Manchester University (UK), is a tool to aid formal reasoning about specifications including a proof assistant and VDM support. See also the Mural Project.

Nqthm is a theorem prover and **Pc-Nqthm** is an interactive proof-checker enhancement of the Boyer-Moore Theorem Prover from Computational Logic Inc. See Section B.2 in this guidebook for additional information.

Nuprl is a tool based on intuitionistic type theory. See Section B.2 in this guidebook for additional information.

OBJ, originated by Joseph Goguen, includes the OBJ3 specification language and the 2OBJ theorem prover, available from Oxford University.

Otter, a fourth-generation automated deduction system, is a resolution-based theorem prover developed at the Argonne National Laboratory in Illinois.

Penelope, from Odyssey Research Associates in Ithaca, New York, is an assertion language and theorem proving environment that supports the specification and verification of sequential Ada programs. Email maureen@oracorp.com.

Petri Nets, a formal graphical notation for modeling systems with concurrency, is a well-established technique supported by a variety of tools accessible through the Petri Nets Web.

Pi-calculus is based on CCS (Calculus of Communicating Systems) developed by Robin Milner *et al.* at the University of Edinburgh. Supporting tools include the Mobility Workbench from Uppsala University of Sweden.

ProofPower is a commercial tool, developed and marketed by ICL (UK), supporting development and checking of specifications and formal proofs in Higher Order Logic and/or Z. Support for Z uses a deep(ish) embedding of Z into HOL, but includes syntax and type checking customized for Z.

PVS (Prototype Verification System) is a tool based on classical typed higher-order logic developed at the SRI International Computer Science Laboratory. See Section B.2 in this guidebook for additional information.

RAISE language and tools are available from CRI, Denmark. Email raise@csd.cri.dk. See Section B.2 in this guidebook for additional information.

Refinement Calculus, by Ralph Back et al at Abo Akademi University in Finland, is a method of program construction based on stepwise refinement.

RESOLVE, developed by the Reusable Software Research Group of the Ohio State University, is a framework for component-based software together with a specification language based on abstract data types and a discipline for using the language. An archive is accessible from the URL http://www.cis.ohio-state.edu/hypertext/rsrg/RSRG.html.

RRL, the Rewrite Rule Laboratory, supports theorem proving in first order logic with equational theories. Email kapur@cs.albany.edu or hzhang@cs.uiowa.edu.

SDL (Specification and Description Language) is an ITU-standardized language for modeling communications systems based on an extended state machine formalism. Various tools are available as noted in the SDL WWW Server maintained by Tele Danmark Research of Denmark.

SDVS (State Delta Verification System), from the Aerospace Corp. of El Segundo, California, is based on state deltas/temporal logic with extensive proof support. Email blevy@aero.org.

SPIN is an automated verification tool (model checker), using a language based on CSP for finite state systems such as protocols or validation models of distributed systems. It was developed at AT&T Bell Labs.

STeP, the Stanford Temporal Prover, is being developed by a team of Stanford University researchers to support the formal verification of concurrent and reactive systems based on temporal specifications and model checking techniques.

TAM, the Temporal Agent Model, is a refinement calculus from the Real-Time Systems Research Group at York University (UK) that supports the specification of both functional and timing behavior.

TLA (Temporal Logic of Actions), developed by Leslie Lamport of the DEC Systems Research Center, is a logic for specifying and reasoning about concurrent and reactive systems. Tool support for TLA has been provided by the University of Dortmund in Germany.

TPS and **ETPS**, the Theorem Proving System and the Educational Theorem Proving System, provide an automated proving capability for first order logic and type theory. They are available from Carnegie Mellon University.

TTM/RTTL, from York University in Ontario, Canada, is a framework for the specification and verification of real-time reactive systems based on timed transition systems (TTMs) and real-time temporal logic (RTTL).

UNITY, a programming notation and a logic to reason about parallel and distributed programs, was developed by J. Misra and K.M. Chandy. Various reports and a model checker for UNITY are available from the University of Texas at Austin.

VDM (Vienna Development Method) is a comprehensive software development methodology. See Section B.2 in this guidebook for additional information.

Z is a notation for formal specification with tool support. See Section B.2 in this guidebook for additional information.

B.2. DETAILED DESCRIPTION OF SELECTED TOOLS

B.2.1. EVES

NAME: EVES

LANGUAGE: Verdi. Variant of classical set-theory (ZFC), with a library mechanism for information hiding and abstraction, and an imperative programming language.

FEATURES: GNU Emacs interface, well-formedness checker, integrated automated deduction system (NEVER), proof checker, reusable library framework, interpreter, compiler, facilities for status reporting, portable.

SYNOPSIS: EVES is an integrated environment supporting the formal development of systems from requirements to code. Additionally, it may be used for formal modeling and mathematical analysis. To date, EVES applications have primarily been in the realm of security-critical systems.

The EVES mathematics is based on ZFC set theory without the conventional distinction between terms and formulas. Development is treated as "theory extension:" each declaration extends the current theory with a set of symbols and axioms pertaining to those symbols. Proof obligations are associated with every declaration to guarantee the conservative extension property. The EVES library is the repository of reusable concepts (e.g., a variant of the Z mathematical toolkit is included with EVES releases) and is the main support for scaling, information hiding, and abstraction. Library units are either specification units (axiomatic descriptions), model units (models or implementations of specifications), or freeze units (for saving work in progress). The language Verdi is a wide-spectrum language that provides all linguistic constructs for formally expressing requirements, specifications, code, and EVES prover and system commands.

The automated deduction component of EVES (NEVER) provides powerful automated deduction support with integrated decision procedures. The design of NEVER has aimed for a balance between what a computer can do well with what a developer can do well. Hence, the support for simplification, rewriting, and heuristics, provide significant automated capabilities. Additionally, there are many prover commands that allow the developer to carefully direct the prover as needed. The prover is fully integrated with EVES and

makes full use of the modularization capabilities of the library. EVES distinguishes between "proof discovery" and "proof certification." NEVER aids in proof discovery and makes use of non-axiomatic reasoning (e.g., the linear programming techniques used by the simplifier). However, once a proof has been discovered, NEVER logs the proof and a separate proof checker will certify that the proof meets the requirements of the EVES logic.

DOCUMENTATION

M. Saaltink, S. Kromodimoeljo, B. Pase, D. Craigen and I. Meisels, "Data Abstraction in EVES", in *Proceedings of Formal Methods Europe '93 (FME'93)*, Odense, Denmark, April 1993.

D. Craigen, S. Kromodimoeljo, I. Meisels, B. Pase, and M. Saaltink, "EVES: An Overview", in *Proceedings of VDM'91*, Noordwijkerhout, The Netherlands, October, 1991.

Extensive documentation, including the reference manual, is available electronically through the ORA Canada WWW page at URL http://www.ora.on.ca/

TOOL REQUIREMENTS: EVES is implemented in a disciplined subset of Common Lisp and is currently available on Suns (UNIX) and PCs (under DOS and OS/2). EVES requires at least 16Mb RAM.

AVAILABILITY: EVES is available by tape or, by arrangement, FTP. All installations of EVES must be licensed by ORA Canada. For academic and research use, there is no charge for FTP access and a nominal distribution fee for tapes. Requests should be sent to eves@ora.on.ca or to the following contact person.

Dan Craigen
ORA Canada
Suite 100, 267 Richmond Road
Ottawa, Ontario K1Z 6X3 CANADA

Email: dan@ora.on.ca
Phone: +1 613 722 3700
Fax: +1 613 722 3531

B.2.2. HOL

NAME: HOL (HOL, HOL2, HOL88, HOL90)

LANGUAGE: Higher order logic with definition and polymorphic extensions.

FEATURES: Parser, pretty-printer, typechecker, forward and goal oriented theorem prover.

SYNOPSIS: The HOL system is an interactive mechanized proof assistant. The system supports both forward and backward proofs. The forward proof style applies inference rules to existing theorems to obtain new theorems and eventually the desired theorem. Backward or goal oriented proofs start with the goal to be proven. Tactics are applied to the goal and subgoals until the goal is decomposed into simpler existing theorems.

Higher order logic is used as the description language of the HOL system. Higher order logic provides a general and expressive vehicle for reasoning about various classes of systems. Some of the applications of the HOL system include the specification and verification of compilers, microprocessors, interface units, algorithms, and the formalization of process algebras, program refinement tools, and distributed algorithms.

The HOL system is used by research groups throughout the world. A technical discussion and support group exists via the electronic mailing list info-hol@leopard.cs.byu.edu. Users regularly contribute to the HOL theory library and an extensive library exists that can be obtained with the HOL system distribution.

DOCUMENTATION:

M. J. C. Gordon and T. F. Melham (eds.). *Introduction to HOL*. Cambridge University Press, 1993.

A system manual is provided with the distribution in four volumes: Tutorial, Description, Reference, and Libraries. The release also contains case studies and LaTeX sources for training course slides.

HOL system information is available on the World Wide Web with URL:

http://lal.cs.byu.edu/lal/hol-documentation.html

TOOL REQUIREMENTS: The HOL system requires a platform running Common Lisp (HOL88) or Standard ML (HOL90). The HOL system has been used on workstations and PCs. A minimum of 8 megabytes of memory is recommended for HOL88 and 24 megabytes for HOL90.

AVAILABILITY: HOL is available by tape (hol-support@cl.cam.ac.uk) or FTP:
HOL88 (lal.cs.byu.edu: pub/hol/holsys.tar.gz)
HOL90 (research.att.com: dist/ml/hol90/hol90.5.tar.Z)
SML/NJ (princeton.edu: pub/ml)

B.2.3. LARCH

NAME: Larch

LANGUAGE: First order logic with equational rewrite rules.

FEATURES: Parser, typechecker, user-directed prover.

SYNOPSIS: Larch is a specification language supporting equational theories embedded in a first order logic. The Larch Prover (LP) is designed to treat equations as rewrite rules and carry out other inferences such as induction and proof by cases. A user may introduce operators and assertions about the operators as part of the formalization process.

LP is designed to work midway between proof checking mode and fully automatic theorem proving mode. Users may direct the proof process at a fairly high level. LP attempts to carry out routine steps in a proof automatically and provide useful information about why proofs fail. LP is not designed to find difficult proofs automatically.

Larch and LP have been used in a variety of applications including digital circuit specification and verification, reasoning about concurrency, programming language semantics, and mathematics. There has also been some Larch work with mainstream programming languages such as the Larch/C Interface Checker (LCL) and C Program Checker (LCLint).

DOCUMENTATION:
S. J. Garland and J. V. Guttag. *A Guide to LP, the Larch Prover*. DEC Systems Research Center Report 82, 1991.

TOOL REQUIREMENTS: The Larch Prover is written in CLU and runs on DEC MIPS, Alpha, and VAX computers as well as Sun workstations.

AVAILABILITY: LP is available by FTP (larch.lcs.mit.edu: pub/Larch/lp2.4.*). Contacts include garland@lcs.mit.edu and guttag@lcs.mit.edu.

B.2.4. NQTHM

NAME: Nqthm (Boyer-Moore Theorem Prover)

LANGUAGE: A variant of Pure Lisp.

FEATURES: Parser, pretty-printer, limited typechecker (language is largely untyped), theorem prover, animator.

SYNOPSIS: Nqthm-1992, the final release of the Nqthm (Boyer-Moore) prover, is a toolset based on a powerful heuristic theorem prover for a restricted logic. There is no explicit specification language; one writes specifications directly in the Lisp-like language that encodes the quantifier-free, untyped logic. Recursion is the main technique for defining functions and mathematical induction is the main technique for proving theorems.

The highly automated prover can be driven by large databases of previously supplied (and proved) lemmas. The tool distribution comes with many megabytes of formalized and proved applications. For over a decade, the Nqthm series of provers has been used to formalize a wide variety of computing problems including critical algorithms, operating systems, compilers, security devices, microprocessors, and pure mathematics.

An interactive enhancement (Pc-Nqthm-1992) is also available. This front-end tool adds a higher degree of user control to the proof process making the system act more like a proof checker than an automatic prover. Acl2, the successor to Nqthm, is currently under development by Boyer, Moore and Kaufmann.

DOCUMENTATION:
Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, 1988.

Additional documentation comes with the tool distribution, including updated chapters of the Handbook.

TOOL REQUIREMENTS: Can be built on top of any Common Lisp on a platform with about 8 megabytes of memory.

AVAILABILITY: Available by ftp (ftp.cli.com: pub/nqthm/nqthm-1992) or tape.

Computational Logic, Inc.
Suite 290
1717 W. 6th Street
Austin, TX 78703 USA

+1-512-322-9951 +1-512-322-0656
software-request@cli.com

B.2.5. NUPRL

NAME: Nuprl

LANGUAGE: Based on constructive type theory with an extensible syntax.

FEATURES: Window-based proof development system including library management and structure editor.

SYNOPSIS: Nuprl was designed originally by Joseph Bates and Robert Constable at Cornell University and has been expanded and improved over the past 15 years by a large group of students and research associates. Nuprl is a highly extensible open system that provides for interactive creation of proofs, formulas, and terms in a typed language. The Nuprl system supports higher order logics and rich type theories. The logic and the proof system are built on a highly regular untyped term structure, a generalization of the lambda calculus. Mechanisms are given for reduction of these terms. The style of the Nuprl logic is based on the stepwise refinement paradigm for problem solving in that the system encourages the user to work backwards from goals to subgoals until one reaches what is known.

As a computer system, Nuprl supports a window-based interactive environment for editing, proof generation and function evaluation. The system incorporates a sophisticated display mechanism that allows users to customize the display of terms, even allowing for the use of user-extended fonts. Based on structure editing, the system is free to display terms without regard to parsing of syntax. The system also includes the functional programming language ML as its meta-language; users extend the proof system by writing their own proof-generating programs (tactics) in ML. Since tactics invoke the primitive Nuprl inference rules, user extensions via tactics cannot corrupt system soundness. The system includes a library mechanism and is provided with a set of libraries supporting the basic types including the integers,

lists, and Booleans. The system also provides an extensive collection of tactics.

The Nuprl system has been used as a research tool to solve open problems in constructive mathematics. It has been used in formal hardware verification and as a research tool in software engineering and to teach mathematical logic to Cornell undergraduates. It is now being used to support parts of computer algebra and is linked to the Weyl computer algebra system.

DOCUMENTATION: An earlier version of the system is documented in the book, Implementing Mathematics with the Nuprl Proof Development System, by Constable et al. and published by Prentice Hall in 1986. Documentation for version 4.1 comes online with the system. There is also a Mosaic Nuprl Library Browser accessible from the Cornell Computer Science Department home page on the World Wide Web.

TOOL REQUIREMENTS: Lucid or Allegro Common Lisp, X11R5.

AVAILABILITY: The system is available free via FTP or on tape.

Professor Robert Constable
Computer Science Department
Upson Hall
Cornell University
Ithaca, NY 14853

phone: +1-607-255-9204
email: nuprl@cs.cornell.edu
WWW: http://www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html

---

B.2.6. PVS

NAME: PVS (Prototype Verification System)

LANGUAGE: Classical, typed higher-order logic with predicate subtypes, dependent typing, and abstract data types.

FEATURES: Customized GNU Emacs interface, parser, typechecker, integrated proof checker, BDD simplifier, prettyprinter, browser, specification libraries, and facilities for status-reporting, cross-reference generation, and LaTeX-printing.

SYNOPSIS: PVS provides an integrated environment for the development and analysis of formal specifications and is intended primarily for the

formalization of requirements and design-level specifications, and for the rigorous analysis of difficult problems. PVS has been applied to algorithms and architectures for fault-tolerant flight control systems, to problems in real-time system design, and to hardware verification.

PVS specifications are organized into parameterized theories that may contain assumptions, definitions, axioms and theorems. Definitions are guaranteed to provide conservative extension. Libraries of proved specifications from a variety of domains are available.

PVS offers a rich type system, strict typechecking, and powerful automated deduction with integrated decision procedures for linear arithmetic and other useful domains, and a comprehensive support environment. A PVS specification is typically expressed using type constraints that are enforced through automatically generated proof obligations, many of which are automatically discharged by the system. The expressive specification language allows concise and natural specifications across a wide range of problem domains. The proof checker provides direct control by the user for the higher levels of proof development, and powerful automation for the lower levels, using a collection of primitive inference procedures that can also be combined by the user to develop higher-level proof strategies. Proofs yield scripts that are displayed in a readily understood format and can be edited and reused. Context is preserved across sessions.

DOCUMENTATION:

S. Owre, N. Shankar, J.M. Rushby. "User Guide for the PVS Specification and Verification System (Beta Release)". Computer Science Laboratory, SRI International. Three volumes: Language, System, and Prover Reference Manuals.

These and other manuals, papers, and technical reports both by the FM group at SRI and outside users are documented in the SRI WWW page and available by anonymous FTP.

FTP: ftp to ftp.csl.sri.com, connect to directory /pub/reports
WWW: http://www.csl.sri.com/sri-csl-fm.html
User group: pvs@csl.sri.com

TOOL REQUIREMENTS: PVS is implemented in Common Lisp and runs on most modern workstations; the requirements are a Unix machine that runs Gnu Emacs and a Common Lisp compiler with integrated CLOS. If typeset specifications are of interest, LaTeX and an appropriate viewer must also be available. The standard version of PVS is implemented in Allegro Lisp and runs on Sun SPARCstations. PVS requires about

20 megabytes of disk space, 50 megabytes of swap space, and 32 megabytes of real memory.

AVAILABILITY: PVS is available by tape or by FTP. All installations of PVS must be licensed by SRI. There is no license fee and no charge for a PVS system obtained via FTP. A nominal distribution fee is charged for tapes and for nonstandard versions. Requests should be addressed to pvs-request@csl.sri.com or to one of the following contacts.

John Rushby, N. Shankar, Sam Owre

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025 USA

Email: {rushby, shankar, owre}@csl.sri.com
Phone: +1-415-859-5456/5272/5114
Fax: +1-415-859-2844

B.2.7. RAISE

NAME: RAISE (Rigorous Approach to Industrial Software Engineering)

LANGUAGE: Wide-spectrum language for specifying operations and processes plus derivations from one level of specification to the next.

FEATURES: Window-based editor, parser, typechecker, proof tools, database, translators to C, Ada.

SYNOPSIS: RAISE is an approach that is based on a rigorous development methodology with less emphasis on mechanical theorem proving and formal analysis. Under the RAISE methodology, development steps are carefully organized and formally annotated using the RAISE specification language. The CORE requirements method is also provided as a requirements approach for front-end analysis.

Derivations from one level to the next generate proof obligations. These obligations may be addressed using the proof tools. A notion of validation (establishing system properties) is also supported. Detailed descriptions of the development steps and overall process are available under the tools. The final implementation step may be partially mechanized for common languages (C, Ada).

RAISE evolved from the VDM formal approach. It has been supported by the European ESPRIT projects and the VDM Europe support organization. The LaCoS project is the primary user community, under support from ESPRIT. The specification language and the toolset are still evolving.

DOCUMENTATION:

M. Nielsen, K. Havelund, K. Wagner, and C. George. *The RAISE Language, Methods, and Tools*. Formal Aspects of Computing, 1:85-114, 1989.

Other documentation includes overviews, method manual, tool manuals, and specification language manuals.

TOOL REQUIREMENTS:

AVAILABILITY: RAISE tools are available from Computer Resources International in Denmark (raise@csd.cri.dk).

B.2.8. VDM

NAME: VDM (Vienna Development Method)

LANGUAGE: First order logic with abstract data types.

FEATURES: Parsers, typecheckers, pretty-printers, proof support, animators, test case generators.

SYNOPSIS: VDM is a model-oriented formal specification and design method based on discrete mathematics. The formal specification language component of VDM is known as META-IV. In VDM, a system is to be developed by first specifying the system formally and proving that the specification is consistent, then iteratively refining and decomposing the specification while proving that each refinement satisfies the previous specification. In theory, this continues until the implementation level is reached.

Specifications are written as constructive specifications of an abstract data type, by defining a class of objects and a set of operations that act upon the objects. The model of a system or subsystem is then based on such an abstract data type. A number of primitive data types are provided in the language along with facilities for user-defined types. Included are conventional first order logic features.

VDM has been used extensively in Europe. Its methods are closely related to those of RAISE and Z. A number of tools have been developed to support formalization using VDM including the Mural tool to aid formal reasoning via a proof assistant.

DOCUMENTATION:

D. Bjorner and C. B. Jones (eds.). *The Vienna Development Method: The Meta-Language*. Lecture Notes in Computer Science, 61, Springer-Verlag, 1978.

D. Bjorner and C. B. Jones. *Formal Specification and Software Development*. Prentice-Hall, 1982.

S. Hekmatpour and D. Ince. *Software Prototyping, Formal Methods and VDM*. Addison-Wesley, 1988.

TOOL REQUIREMENTS: VDM tools are provided by a variety of sources and are not integrated into a single toolset. Most will run on conventional Unix workstation platforms.

AVAILABILITY: VDM tools are available from a variety of sources. See the VDM WWW page (file://hermes.ifad.dk/pub/docs/vdm.html) for details.

B.2.9. Z

NAME: Z (pronounced "zed")

LANGUAGE: Set theory and first order logic with graphical representations.

FEATURES: Parsers, typecheckers, pretty-printers, proof support.

SYNOPSIS: Z has evolved from being initially only a loose representation for formal specifications to a semi-standardized language with tool support provided by a variety of third parties. It has been under development by the Programming Research Group at Oxford University. Z is based on set theory and is oriented toward constructing models. The basic form used is called a "schema," which is used to introduce an axiomatization of a function. Models are constructed by specifying a series of schemas using (typically) a state transition style.

Tools for Z initially were limited to formatting and typechecking, but are progressing into proof support, primarily through the use of HOL as the underlying theorem proving engine. These include both limited proof checking tools as well as more aggressive theorem proving tools.

The standardization of Z should solidify the tool base and enhance interest in mechanized support.

Z is currently undergoing standardization in the UK and internationally through ISO. Z has been used extensively in Europe, primarily in the UK, but has seen little North American use. It has been used to write formal specifications for various industrial software development efforts and has resulted in two awards for technological achievement: for the IBM CICS project and for a specification of the IEEE standard for floating-point arithmetic.

DOCUMENTATION:

A. Diller. *Z: An Introduction to Formal Methods*. John Wiley & Sons, 1990.

J. M. Spivey. *Understanding Z, A Specification Language and its Formal Semantics*. Cambridge University Press, 1988.

J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1989.

TOOL REQUIREMENTS: Z tools are provided by a variety of sources and are not integrated into a single toolset. Most will run on conventional Unix workstation platforms. Heavy use is made of LaTeX for printing formatted specifications.

AVAILABILITY: Z tools are available from a variety of sources. See the Z WWW page (http://www.comlab.ox.ac.uk/archive/z.html) for details.

B.3. STATE-SPACE EXPLORATION TOOLS

B.3.1. COSPAN

NAME: COSPAN (COordination SPecification ANalysis)

LANGUAGE: S/R (selection/resolution) belongs to a class of languages, the omega-regular languages, that are expressible as finite-state automata on infinite strings or behavioral sequences. The s/r language is used to define both the system and its requirements and is particularly suited to developing distributed or state-machine-based environments viewed in terms of data flow.

FEATURES: COSPAN is a general-purpose, rapid-prototyping tool developed at AT&T that provides a theoretically seamless interface between an

abstract model or standard and its target implementation, thereby supporting top-down system development and analysis. COSPAN offers facilities for documentation, conformance testing, software maintenance, debugging, and statistical analysis, as well as libraries of abstract data types and reusable pretested components.

SYNOPSIS: COSPAN has been used in the commercial development of both software and hardware systems, a partial list of which includes: analysis of high-level models of several communications protocols (e.g., the X.25 packet switching link layer protocol, the file transfer and management protocol (FTAM) of the International Telegraph and Telephone Consultative Committee (CCITT), and AT&T's Datakit universal receiver protocol (URP) level C), verification of a custom VLSI chip to implement a packet layer protocol controller, and analysis and implementation of AT&T's Trunk Operations Provisioning Administration System (TOPAS).

COSPAN is based on homomorphic reduction and refinement of omega-automata, i.e., the use of homomorphisms to relate two automata in a process based on successive refinement that guarantees that properties verified at one level of abstraction hold in all successive levels. Reduction of the state space is achieved by exploiting symmetries and modularity inherent in large, coordinating systems. Verification is framed as a language-containment problem; checking consists of determining whether the language of the system automaton is contained in the language of the specification automaton. Omega-automata are particularly well-suited to expressing liveness properties, i.e., events that must occur at some finite, but unbounded time.

DOCUMENTATION:

Z. Har'EL and R. Kurshan. "Software for Analytical Development of Communications Protocols." *AT&T Technical Journal*, pp. 45-59. Jan, Feb 1990.

R. Kurshan. *Automata-Theoretic Verification of Coordinating Processes*. Princeton University Press, 1993. A compressed post-script version of this book is available as notes.PS.Z via ftp (host: ftp.research.att.com; directory: /dist/COSPAN).

TOOL REQUIREMENTS:

AVAILABILITY: COSPAN may be obtained by universities for educational/research purposes through a written request by the department chair on departmental letterhead to the contact listed below. Once a non-disclosure agreement is signed, COSPAN binaries are made available at no charge by tape or ftp.

Contact: R. Kurshan
AT&T Bell Labs, Room 2C-353
Murray Hill, NJ 07974
Email: k@research.att.com

B.3.2. MURPHI

NAME: Murphi

LANGUAGE: High-level, transition-rule-based description language for concurrent systems.

FEATURES: Automatic state exploration tool that can be used as a verifier or simulator.

SYNOPSIS: Murphi is a complete finite-state verification system that has been tested on extensive industrial-scale examples including cache coherence protocols and memory models for commercially-designed multiprocessors.

The Murphi Verification System consists of the Murphi Compiler, and the Murphi description language for finite-state asynchronous concurrent systems which is loosely-based on Chandy and Misra's Unity model and includes user-defined datatypes, procedures, and parameterized descriptions. A version for synchronous concurrent systems is under development. A Murphi description consists of constant and type declarations, variable declarations, rule definitions, start states, and a collection of invariants. The Murphi compiler takes a Murphi description and generates a C++ program that is compiled into a special-purpose verifier that checks for invariant violations, error statements, assertion violations, deadlock, and (in certain versions) liveness. The verifier attempts to enumerate all possible states of the system, while the simulator explores a single path through the state space. Efficient encodings, including symmetry-based techniques, and effective hash-table strategies are used to alleviate state explosion.

DOCUMENTATION:

D. Dill, A. Drexler, A. Hu, and C. Yang. "Protocol Verification as a Hardware Design Aid." *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 522-525. IEEE Computer Society, October, 1992.

C.N. Ip and D. Dill. "Better Verification through Symmetry." *International Conference on Computer Hardware Description Languages*, pp. 87-100. April, 1993.

These and other papers and manuals are available by FTP from host snooze.stanford.edu in directory /pub/papers/verification.

TOOL REQUIREMENTS: C++ compiler.

AVAILABILITY:

Available by FTP; host: snooze.Stanford.Edu, directory: /pub/murphi
Email inquiries: murphi@snooze.stanford.edu

Contact: David L. Dill
Computer Science Department
Stanford University
Stanford, CA 94305

Email: dill@hohum.Stanford.Edu or
Phone: +1-415-725-3642
FAX: +1-415-725-6278

B.3.3. SMV

NAME: SMV (Symbolic Model Verifier)

LANGUAGE: The input language, SMV, is a relatively high-level description language that provides modular hierarchical descriptions and definition of reusable components. The specification language, Computation Tree Logic (CTL), is a propositional, branching-time temporal logic.

FEATURES: Symbolic model checker that verifies finite state systems described in the SMV language against specifications written in CTL. Implemented with BDDs (reduced, ordered Binary Decision Diagrams), SMV can handle both synchronous and asynchronous systems, and arbitrary safety and liveness properties.

SYNOPSIS: The SMV system has been distributed widely and used to verify industrial-scale circuits and protocols, including the cache coherence protocol described in the IEEE Futurebus+ standard and the cache consistency protocol developed at Encore Computer Corporation for their Gigamax distributed multiprocessor.

SMV is designed to provide largely automatic verification of finite state system descriptions that run the gamut from completely synchronous to completely asynchronous, and from detailed to abstract. The SMV input language offers a set of basic data types consisting of bounded integer subranges and symbolic enumerated types, which can be used to construct static, structured types. CTL provides a concise syntax for expressing a rich class of temporal properties including safety, liveness, fairness, and deadlock freedom. SMV uses a BDD-based symbolic model checking algorithm to avoid explicitly enumerating the states of the model. With carefully-tuned variable ordering, the BDD algorithm yields a system capable of verifying circuits with extremely large numbers of states. Examples of the scalability of this approach include a pipelined ALU with over 10^{120} states and an asynchronous stack with over 10^{50} states.

DOCUMENTATION:

J. Burch, E. Clarke, D. Long, K. McMillan and D. Dill. "Symbolic Model Checking for Sequential Circuit Verification." *IEEE Transactions on Computer-Aided Design*, Vol. 13, No. 4, April, 1994.

K. McMillan. *Symbolic Model Checking*. Kluwer, Boston, MA, 1993.

Additional papers and manuals are available by FTP from the CMU host and directory shown below.

TOOL REQUIREMENTS: SMV runs on Sun3's and Encore Multimaxen under the Mach operating system. It may also run on Sun4, DecStation 3000, and VAX, but performance varies. Certain dependencies on the MACH operating system may need to be removed via minor modifications to makefiles for non-MACH sites.

AVAILABILITY: SMV is available by FTP or by tape. All users are asked to sign a license agreement available online. There is no license fee.

FTP: host: emc.cs.cmu.edu, directory: /pub/tape

Contact: Edmund Clarke
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Email: emc+@cs.cmu.edu.

B.4. REFERENCES

[Craigen] D. Craigen, S. Gerhart, and T. Ralston, *An International Survey of Industrial Applications of Formal Methods*, U.S. National Institute of Standards and Technology, Reports NIST GCR 93/626 (Vols. 1 and 2), March 1993. Also available from the U.S. Naval Research Laboratories, Formal Report 5546-93-9581/9582 (September 1993), and from the Atomic Energy Control Board of Canada, Reports INFO-0474-1 (Vol. 1) and INFO-0474-2 (Vol. 2), January, 1995.

SUGGESTIONS FOR IMPROVEMENTS FORM

Product Name: Formal Methods Specification and Verification Guidebook for Software and Computer Systems
Volume I: Planning and Technology Insertion

Product Version Number: NASA-GB-002-95
RELEASE 1.0

NASA Change Request Tracking Number: _____

Name of Submitting Organization: _____
Organization Contact: _____ **Telephone:** _____
Mailing Address:

Date: _____ **Short Title:** _____
Change Location Tag (use section or paragraph #, figure #, key process area ID, practice ID, etc.): _____

Proposed Change:

Rationale for Change:

Note: For NASA to take appropriate action on the change request, we must have a clear description of the recommended change along with supporting rationale.

Send US Mail To:

FM Specification and Verification Guidebook, Vol. I
NASA IV&V Facility
100 University Drive
Fairmont, WV 26554

Send via Internet Email To:

John.C.Kelly@ccmail.jpl.nasa.gov