A Review of Formal Methods

May 26, 1993

PREPARED FOR:

Rome Laboratory
RL/C3C
Griffiss Business Park
Rome, NY 13441

PREPARED BY:

Robert L. Vienneau
Data & Analysis Center for Software
Griffiss Business and Technology Park
775 Daedalian Drive
Rome, New York 13441-4909

This abridged version is from *Software Engineering,* edited by Merlin Dorfman and Richard H. Thayer, Computer Society Press, 1996.

A full version of this report, (see table of contents) with a description of Dijkstra's predicate calculus and Harlan Mills functional approach, appendices, and an annotated bibliography is available for more information follow the Abstract and Ordering Information button below.

Table of Contents   Abstract & Ordering

DoD DACS Home Page   Technical Reports

A Review of Formal Methods

This is the table of contents for the full version which may be purchased for **$30** from the DACS Product Order Form.

The table of contents for the on-line version is here.

DoD DACS Home Page

Technical Reports

A Review of Formal Methods

Table of Contents

This abridged version is from *Software Engineerings*, edited by Merlin Dorfman and Richard H. Thayer, Computer Society Press, 1996.

A Review of Formal Methods

1.0 Introduction

The seventies witnessed the structured programming revolution. After much debate, software engineers became convinced that better programs result from following certain precepts in program design. Recent imperative programming languages provide constructs supporting structured programming. Achieving this consensus did not end debate over programming methodology. Quite the contrary, a period of continuous change began, with views on the best methods of software development mutating frequently. Top-down development, modular decomposition, data abstraction, and, most recently, object oriented design are some of the jargon terms that have arisen to describe new concepts for developing large software systems. Both researchers and practitioners have found it difficult to keep up with this onslaught of new methodologies.

There is a set of core ideas that lies at the base of these changes. Formal methods have provided a unifying philosophy and central foundation upon which these methodologies have been built. Those who understand this underlying philosophy can more easily adopt these and other programming techniques. This report provides the needed understanding of formal methods to guide a software manager in evaluating claims related to new methodologies. It also provides an overview for the software engineer and a guide to the literature. Ample examples are provided to fully convey the flavor of using formal methods.

The underlying philosophy for formal methods has not changed in over two decades. Nevertheless, this approach is a revolutionary paradigm shift from conventional notions about computer programming. Many software engineers have adopted the new methodologies that grew out of this work without understanding or even being aware of the root concepts. The traditional notion of programming looks at the software engineer's task as the development of code to instruct a physically existing machine to compute a desired result. Existing computers possess many idiosyncrasies reflecting hardware engineering concerns. Likewise user interfaces and the desired function can be expected to introduce additional complexities. In the traditional view of programming, these details can be expected to appear in a design, or even a specification, at all levels of abstraction. The engineer's job is seen as the introduction of more details and tricks to get the utmost in speed and performance out of computers. Since software development is therefore a "cut and fit" process, such complex systems can be expected to be full of bugs. A careful testing process is seen as the means of detecting and removing these bugs.

The mindset behind formal methods is directly opposite to the traditional view. It is the job of the hardware engineer, language designer, and compiler writer to provide a machine for executing code, not the reverse:

> Originally I viewed it as the function of the abstract machine to provide a truthful picture of the physical reality. Later, however, I learned to consider the abstract machine as the 'true' one, because that is the only one we can 'think;' it is the physical machine's purpose to supply 'a working model,' a (hopefully!) sufficiently accurate physical simulation of the true, abstract machine...It used to be the program's purpose to instruct our computers; it became the computer's purpose to execute our programs. [Dijkstra 76]

The software engineer's task is to produce several models or descriptions of a system for an abstract machine, with accompanying proofs that models at lower levels of abstraction correctly implement higher-level models. Only this design process can ensure high levels of quality, not testing. Edsger Dijkstra has asserted that testing can demonstrate the presence of faults, not their absence. Since software engineers must be able to read and reason about designs, implementation details must be prevented from influencing the expression of designs as long as possible. A separation of concerns exists here, and if microefficiency concerns are allowed to dominate, produced code will reflect a design that cannot be convincingly demonstrated correct by anyone.

The contrast between these views is controversial (for example, see the discussion engendered in the ACM Forum by [DeMillo 79], [Fetzer 88], [Dijkstra 89], or [Gries 91]). Advocates of formal methods argue that many have adopted structured programming and top-down development without really understanding the underlying formalism [Mills 86]. A concern with formal methods can produce more rigorous specifications, even if they are expressed in English [Meyer 85]. Designs and code will be easier to reason about, even if fully formal proofs are never constructed. Critics focus on the difficulties in scaling up to large systems, the impracticalities of formalizing many inherently complex aspects of systems (for example, user interactions and error-checking code), and the radical retraining needed for the large population of already existing software engineers.

This report's purpose is not to advocate one or another position on formal methods. Rather, it overviews the technical basis for formal methods, while critically noting weaknesses. Polemics are avoided, but enough information is reviewed to allow the reader to form an informed judgment on formal methods. Formal methods are beginning to see more widespread industrial use, especially in Europe. Their use is characteristic of organizations with a Defined (Level 3) process or better, as specified in the process maturity framework developed by the Software Engineering Institute [Humphrey 88]. Formal methods have the potential of engendering further revolutionary changes in practice and have provided the underlying basis of past changes. These reasons make it imperative that software managers and engineers be aware of the increasingly widespread debate over formal methods.

A Review of Formal Methods

2.0 Definition and Overview of Formal Methods

Wide and narrow definitions of formal methods can be found in the literature. For example, Nancy Leveson states:

> A broad view of formal methods includes all applications of (primarily) discrete mathematics to software engineering problems. This application usually involves modeling and analysis where the models and analysis procedures are derived from or defined by an underlying mathematically- precise foundation. [Leveson 90]

A more narrow definition, however, better conveys the change in practice recommended by advocates of formal methods. The definition offered here is based on that in [Wing 90], and has two essential components. First, formal methods involve the essential use of a formal language. A formal language is a set of strings over some well-defined alphabet. Rules are given for distinguishing those strings, defined over the alphabet, that belong to the language from strings that do not.

Second, formal methods in software support formal reasoning about formulae in the language. These methods of reasoning are exemplified by formal proofs. A proof begins with a set of axioms, which are to be taken as statements postulated to be true. Inference rules state that if certain formulas, known as premises, are derivable from the axioms, then another formula, known as the consequent, is also derivable. A set of inference rules must be given in each formal method. A proof consists of a sequence of well-defined formulae in the language in which each formula is either an axiom or derivable by an inference rule from previous formulae in the sequence. The last formula in the sequence is said to be proven. The following definition summarizes the above discussion:

> *A* **formal method** *in software development is a method that provides a formal language for describing a software artifact (e.g. specifications, designs, source code) such that formal proofs are possible, in principle, about properties of the artifact so expressed.*

Often, the property proven is that an implementation is functionally correct, that is, it fulfills its specification. Thus, either the formal language associated with a method permits a system to be described by at least two levels of abstraction or two languages are provided for describing a specification and its implementation. The method provides tools with which an implementation can be proven to satisfy a specification. To be practically useful, the method should also provide heuristics and guidelines for developing elegant specifications and for developing implementations and proofs in parallel.

The concept of formalism in formal methods is borrowed from certain trends in 19th and 20th century mathematics. The development of consistent non-Euclidean geometries, in which supposedly parallel lines may intersect, led mathematicians to question their methods of proof and to search for more rigorous foundations. Eventually, these foundations came to be seen as describing numbers, sets, and logic. Leading mathematicians in this movement included Karl Weierstrass, Gottlob Frege, Giuseppe Peano, and David Hilbert. By the turn of the century, a foundation seemed to be in place, but certain strange examples and antinomies caused mathematicians to question the security of their foundations and even their own intuition on fundamental matters. A mechanical method of manipulating symbols was thus invented to investigate these questions. Due to fundamental discoveries of Kurt Godel, Thoralf

Skolem, and Leopold Lowenheim, the results of using this method were ambiguous. Nevertheless, the axiomatic method became widely used in advanced mathematics, especially after impetus was added to this tendency by an extremely influential group of French mathematicians writing around World War II under the pseudonym of Nicholas Bourbaki [Kline 80].

Formal methods are merely an adoption of the axiomatic method, as developed by these trends in mathematics, for software engineering. In fact, Edsger Dijkstra has suggested, somewhat tongue-in-cheek, that computer science be renamed Very Large Scale Application of Logic (VLSAL) [Dijkstra 89]. Mastery of formal methods in software requires an understanding of this mathematics background. Mathematical topics of interest include formal logic, both the propositional calculus and predicate logic, set theory, formal languages, and automata such as finite state machines. The full flavor of the relevant mathematics cannot be conveyed here, but Appendix A does contain a complete description of a formal system that has been intensely studied by mathematicians and is of use in software development. That example, the propositional calculus, can serve as a paradigm of formal systems in mathematics and helps to explicate the definition of formal methods offered here.

2.1 Use of Formal Methods

How are the mathematics of formal languages applied in software development? What engineering issues have been addressed by their application? Formal methods are of global concern in software engineering. They are directly applicable during the requirements, design, and coding phases and have important consequences for testing and maintenance. They have influenced the development and standardization of many programming languages, the programmer's most basic tool. They are important in ongoing research that may change standard practice, particularly in the areas of specifications and design methodology. They are entwined with lifecycle models that may provide an alternative to the waterfall model, namely rapid prototyping, the Cleanroom variant on the spiral model, and "transformational" paradigms.

2.1.1 What can be Formally Specified

Formal methods support precise and rigorous specifications of those aspects of a computer system capable of being expressed in the language. Since defining what a system should do, and understanding the implications of these decisions, are the most troublesome problems in software engineering, this use of formal methods has major benefits. In fact, practitioners of formal methods frequently use formal methods solely for recording precise specifications, not for formal verifications ([Hall 90] and [Place 90]).

Some of the most well-known formal methods consist of or include specification languages for recording a system's functionality. These methods include:

- Z (pronounced "Zed")
- Communicating Sequential Processes (CSP)
- Vienna Development Method (VDM)
- Larch
- Formal Development Methodology (FDM)

Formal methods can be used to specify aspects of a system other than functionality. The emphasis of this report is on functionality since such techniques are currently the most well-known, developed, and of general interest. Software safety and security are other areas where formal methods are sometimes applied in practice. The benefits of proving that unsafe states will not arise, or that security will not be violated, can justify the cost of complete formal verifications of the relevant portions of a software system. Formal methods can deal with many other areas of concern to software engineers, but have not been much used, other than in research organizations, for dealing with issues unrelated to functionality, safety, and security. Areas in which researchers are exploring formal methods include fault tolerance, response time, space efficiency, reliability, human factors, and software structure dependencies [Wing 90].

Formal methods can include graphical languages. Data Flow Diagrams (DFDs) are the most well-known graphical technique for specifying the function of a system. DFDs can be considered a semi-formal method, and researchers have explored techniques for treating DFDs in a completely formal manner. Petri nets provide another well-known graphical technique, often used in distributed systems [Peterson 77]. Petri nets are a fully formal technique. Finally, finite state machines are often presented in tabular form. This does not decrease the formalism in the use of finite state machines. So the definition of formal methods provided earlier is quite encompassing.

Software engineers produce models and define the properties of systems at several levels of abstraction. Formal methods can be employed at each level. A specification should describe what a system should do, but not how it is done. More details are provided in designs, with the source code providing the most detailed model. For example, Abstract Data Types (ADTs) frequently are employed at intermediate levels of abstraction. ADTs, being mathematical entities, are perfect candidates for formal treatment and are often so treated in the literature.

Formal methods are not confined to the software components of large systems. System engineers frequently use formal methods. Hardware engineers also use formal methods, such as VHSIC Hardware Description Language (VHDL) descriptions, to model integrated circuits before fabricating them.

2.1.2 Reasoning about a Formal Description

Once a formal description of a system has been produced, what can be done with it? Usable formal methods provide a variety of techniques for reasoning about specifications and drawing implications. The completeness and consistency of a specification can be explored. Does a description imply a system should be in several states simultaneously? Do all legal inputs yield one and only one output? What surprising results, perhaps unintended, can be produced by a system? Formal methods provide reasoning techniques to explore these questions.

Do lower level descriptions of a system properly implement higher level descriptions? Formal methods support formal verification, the construction of formal proofs that an implementation satisfies a specification. The possibility of constructing such formal proofs was historically the principle driver in the development of formal methods. Prominent technology for formal verification includes Edsger Dijkstra's "weakest precondition" calculus ([Dijkstra 76] and [Gries 81]) and Harlan Mills' "functional correctness" approach [Linger 79].

2.1.3 Tools and Methodology

Developments in supporting tools and methodologies have accompanied the development of technology for formalizing software products. The basic idea is that the ultimate end-product of development is not solely a working system. Of equal importance are specifications and proofs that the program meets its specification. A proof is very hard to develop after the fact. Consequently, proofs and programs should be developed in parallel, with close interconnections in the development history. Since programs must be proven correct, only those constructions that can be clearly understood should be used. This is the primary motivation that many early partisans had for advocating structured programming.

A challenge is to apply these ideas on large scale projects. Formal specifications seem to scale up much easier than formal verifications. Nevertheless, ideas relating to formal verifications are applicable to projects of any size, particularly if the level of formality is allowed to vary. David Gries recommends a design methodology incorporating certain heuristics that support more reliable and provable designs [Gries 81]. Harlan Mills has spent considerable effort developing the Cleanroom approach; a lifecycle in which formal methods, inspections, and reliability modeling and certification are integrated in a social process for producing software ([Mills 87] and [Dyer 92]).

Formal methods have also inspired the development of many tools. These tools may bring formal methods into more widespread practice, although interestingly enough, many advocates of formal methods are not strong believers in tools. An obvious example of such tools are program provers. Some tools animate specifications, thereby converting a formal specification into an executable prototype of a system. Other tools derive programs from specifications through various automated transformations. Under some approaches a program is found as a solution to an equation in a formal language. Transformational implementation suggests a future in which many software systems are developed without programmers, or at least with more automation, higher productivity, and less labor ([Agresti 86] and [KBSE 92]).

In some sense, no programmer can avoid formal methods, for every programming language is, by definition, a formal language. Ever since Algol 60 was introduced, standards defining programming languages have used a formal notation for defining language syntax, namely Backus-Naur Form (BNF). A complete description of BNF is given in Appendix B. Usually, standards do not formally define the semantics of programming languages, although, in principle, they could. The convention of using natural language descriptions for defining language semantics is due to not having yet developed settled techniques for defining all constructs included in large languages. Nevertheless, formal methods have resulted in one widely agreed criterion for evaluating language features: how simply can one formally reason about a program with a proposed new feature? The formal specification of language semantics is a lively area of research. In particular, formal methods have always been an interest of the Ada community, even before standardization ([London 77], [McGettrick 82] and [Preston 88]).

2.2 Limitations of Formal Methods

Given the applicability of formal methods throughout the lifecycle, and their pervasive possibilities for almost all areas of software engineering, why are they not more widely visible? Part of the problem is educational. Revolutions are not made by conversion, but by the old guard passing away. More recent university graduates tend to be more willing to experiment with formal methods.

On the other hand, the only barrier to the widespread transition of this technology is not lack of knowledge on the part of practitioners. Formal methods do suffer from certain limitations. Some of these limitations are inherent and will never be overcome. Other restrictions, with research and practice, will be removed as formal methods are transitioned into wider use.

2.2.1 The Requirements Problem

The inherent limitations in formal methods are neatly summarized in the often repeated aphorism, "You cannot go from the informal to the formal by formal means." In particular, formal methods can prove that an implementation satisfies a formal specification, but they cannot prove that a formal specification captures a user's intuitive informal understanding of a system. In other words, formal methods can be used to verify a system, but not to validate a system. The distinction is that validation shows that a product will satisfy its operational mission, while verification shows that each step in the development satisfies the requirements imposed by previous steps [Boehm 81].

The extent of this limitation should not be underemphasized. One influential field study, [Curtis 88], found that the three most important problems in software development are:

- The thin spread of application domain knowledge
- Changes in and conflicts between requirements
- Communication and coordination problems.

Successful projects were often successful because of the role of one or two key exceptional designers. These designers had a deep understanding of the applications domain and could map the applications requirements to computer science concerns. These findings suggest the reduction of informal application knowledge to a rigorous specification is a key problem area in the development of large systems.

Empirical evidence does suggest, however, that formal methods can make a contribution to the problem of adequately capturing requirements. The discipline of producing a formal specification can result in fewer specification errors. Furthermore, implementors without an exceptional designer's knowledge of the application area commit less errors when implementing a formal specification than when relying on hazy knowledge of the application [Goel 91]. These benefits may exist even when the final specification is expressed in English, not a formal language [Meyer 85]. A specification acts as a "contract" between a user and a developer. The specification describes the system to be delivered. Using specifications written in a formal language to complement natural language descriptions can make this contract more precise. Finally, developers of automated programming environments, which use formal methods, have developed tools to interactively capture a user's informal understanding and thereby develop a formal specification [Zeroual 91].

Still, formal methods can never replace deep application knowledge on the part of the requirements engineer, whether at the system or the software level. The application knowledge of the exceptional designer is not limited to one discipline. For example, an avionics application might require knowledge of flight control, navigation, signal processing, and electronic countermeasures. Whether those drawing on interdisciplinary knowledge in developing specifications come to regard formal methods as just another discipline making their life more complicated, or an approach that allows them to simply, concisely, and accurately record their findings, will only be known with experience and experimentation.

2.2.2 Physical Implementations

The second major gap between the abstractions of formal methods and concrete reality lies in the nature of any actual physically existing computer. Formal methods can verify that an implementation satisfies a specification when run on an idealized abstract machine, but not when run on any physical machine.

Some of the differences between typical idealized machines and physical machines are necessary for humanly-readable correctness proofs. For instance, an abstract machine might be assumed to have an infinite memory, while every actual machine has some upper limit. Similarly, physical machines cannot implement real numbers, as axiomatically described by mathematicians, while proofs are most simply constructed assuming the existence of mathematically precise reals. No reason in principle exists why formal methods cannot incorporate these limitations. The proofs will be much messier and less elegant, and they will be limited to a particular machine.

A limitation in principle, however, does exist here. Formal proofs can show with certainty, subject to mistakes in calculation, that given certain assumptions, a program is a correct implementation of a specification. What cannot be formally shown is that those assumptions are correct descriptions of an actual physical system. A compiler may not correctly implement a language as specified. So a proof of a program in that language will fail to guarantee the execution behavior of the program under that compiler. The compiler may be formally verified, but this only moves the problem to a lower level of abstraction. Memory chips and gates may have bugs. No matter how thoroughly an application is formally verified, at some point one must accept that an actual physical system satisfies the axioms used in a proof. Explanations must come to an end sometime.

Both critics [Fetzer 88] and developers ([Dijkstra 76] and [Linger 79]) of formal methods are quite aware of this limitation, although the critics do not always seem to be aware of the developers' explicit statements on this point. This limitation does not mean formal methods are pointless. Formal proofs explicitly isolate those locations where an error may occur. Errors may arise in providing a machine that implements the abstract machine, with sufficient accuracy and efficiency, upon which proofs are based. Given this implementation, a proof vastly increases confidence in a program [Merrill 83].

Although no prominent advocate of formal methods recommends testing be avoided entirely, it is unclear what role testing can play in increasing confidence in the areas not addressed by formal methods. The areas addressed by testing and formal methods may overlap, depending on the specific methodologies employed. From an abstract point of view, the question of what knowledge or rational belief can be provided by testing is the riddle of the rational basis for induction. How can an observation that some objects of a given type have a certain property ever convince anyone that all objects of that type have the property? Why should a demonstration that a program produces the correct outputs for some inputs ever lead to a belief that the program is likely to produce the correct output for all inputs? If a compiler correctly processes certain programs, as defined by a syntactical and semantic standard, why should one conclude that any semantic axiom in the standard can be relied upon for a formal proof of the correctness of a program not used in testing the compiler? Over two centuries ago, the British philosopher David Hume put related questions at the center of his epistemology.

Two centuries of debate have not reached a consensus on induction. Still, human beings are inclined to draw these conclusions. Software developers exhibit the same inclination in testing computer programs. Formal methods will never entirely supplant testing, nor do advocates intend them to do so. In principle, a gap always exists between physical reality and what can be formally verified. With more widespread use of formal methods, however, the role of testing will change.

2.2.3 Implementation Issues

The gaps between users' intentions and formal specifications and between physical implementations and abstract proofs create inherent limitations to formal methods, no matter how much they may be developed in the future. There are also a host of pragmatic concerns that reflect the current state of the technology.

The introduction of a new technology into a large-scale software organization is not a simple thing, particularly a technology as potentially revolutionary as formal methods. Decisions must be made about whether the technology should be completely or partially adopted. Appropriate accompanying tools need to be acquired. Current personnel need to be retrained, and new personnel may need to be hired. Existing practices need to be modified, perhaps drastically. All of these issues arise with formal methods. Optimal decisions depend on the organization and the techniques for implementing formal methods. Several schemes exist, with various levels of feasibility and impact.

The question arises, however, whether formal methods are yet suitable for full-scale implementation. They are most well-developed for addressing issues of functionality, safety, and security, but even for these mature methods, serious questions exist about their ability to scale up to large applications. In much academic work, a proof of a hundred lines of code was seen as an accomplishment. The applicability of such methods to a commercial or military system, which can be over a million lines of code, is seriously in doubt. This issue of scaling can be a deciding factor in the choice of a method. Harlan Mills claims his program function approach applies easier on large systems than Dijkstra's competing predicate calculus method [Mills 86]. Likewise, the languages considered in academic work tend to be extremely simplified compared to real-world programming languages.

One frequently adopted scheme for using formal methods on real-world projects is to select a small subset of components for formal treatment, thus finessing the scalability problem. These components might be selected under criteria of safety, security, or criticality. Components particularly amenable to formal proof might be specifically selected. In this way, the high cost of formal methods is avoided for the entire project, but only incurred where project requirements justify it. Under this scheme, the issue of scaling is avoided, for formal methods are never applied on a large scale.

Decisions about tool acquisition and integration need to be carefully considered. Advocates of formal methods argue that they should be integrated into the design process. One does not develop a specification and an implementation and then attempt to prove the implementation satisfies the specification. Rather, one designs the implementation and proof in parallel, with continual interaction. Sometimes discussion about automated verifiers suggests that the former approach, not the latter, provides an implementation model ([DeMillo 79] and [Merrill 83]). Selective implementation of formal methods on small portions of large projects may make this integration difficult to obtain.

Another approach can have much more global impacts. Perhaps, the entire waterfall lifecycle should be scrapped. An alternate approach is to develop formal specifications at the beginning of the lifecycle and then automatically derive the source code for the system. Maintenance, enhancements, and modifications will be performed on the specifications, with this derivation process being repeated. Programmers are replaced by an intelligent set of integrated tools, or at least given very strong guidance by these tools. Knowledge about formal methods then becomes embodied in the tools, with Artificial Intelligence techniques being used to direct the use of formal methods. This revolutionary programmerless methodology is not here yet, but it is providing the inspiration for many tool developers. For example, this vision is close to that of Rome Laboratory's Knowledge-Based Software Engineering project [KBSE 92].

A third alternative is to partially introduce formal methods by introducing them throughout an organization or project, but allowing a variable level of formality. In this sense, informal verification is an argument meant to suggest that details can be filled in to provide a completely formal proof. The most well-known example of this alternative is the Cleanroom methodology, developed by Harlan Mills [Mills 87]. Given varying levels of formality, tools are much less useful under this approach. The Cleanroom methodology involves much more than formal methods, but they are completely integrated into the methodology. Other technologies involved include the spiral lifecycle, software reliability modeling, a specific testing approach, reliability certification, inspections, and statistical process control. Thus, although this approach allows partial experimentation with formal methods, it still requires drastic changes in most organizations.

No matter to what extent an organization decides to adopt formal methods, if at all, training and education issues arise. Most programmers have either not been exposed to the needed mathematical background, or do not use it in their day-to-day practice. Even those who thoroughly understand the mathematics may have never realized its applicability to software development. Set theory is generally taught in courses in pure mathematics, not computer programming. Even discrete mathematics, a standard course whose place in the university curriculum owes much to the impetus of computer science professional societies, is often not tied to software applications. Education in formal methods should not be confined to degreed university programs for undergraduates newly entering the field. Means need to be found, such as seminars and extension courses, for retraining an existing workforce. Perhaps this educational problem is the biggest hurdle to the widespread transition of formal methods.

Table of Contents NEXT

DoD DACS Home Page Technical Reports

3.0 Specification Methods

Formal methods were originally developed to support verifications, but higher interest currently exists in specification methods. Several methods and languages can be used for specifying the functionality of computer systems. No single language, of those now available, is equally appropriate for all methods, application domains, and aspects of a system. Thus, users of formal specification techniques need to understand the strength and weaknesses of different methods and languages before deciding on which to adopt. This section briefly describes characteristics of different methods now available.

The distinction between a specification method and a language is fundamental. A method states what a specification must say. A language determines in detail how the concepts in a specification can be expressed [Lamport 89]. Some languages support more than one method, while most methods can be used in several specification languages. Some methods are more easily used with certain languages. Appendix C illustrates this distinction by giving two specifications of a queue. These specifications illustrate different methods, but both are in the specification language Z.

3.1 Semantic Domains

A formal specification language contains an alphabet of symbols and grammatical rules that define well-formed formulae. These rules characterize a language's "syntactic domain." The syntax of a language shows how the symbols in the language are put together to form meaningless formulae. Neither the nature of the objects symbolized nor the meanings of the relationships between them are characterized by the syntax of a language. For example, the presentation of the propositional calculus in Appendix A is entirely syntactical.

Meanings, or interpretations of formulae, are specified by the semantics of a language. A set of objects, known as the language's "semantic domain," can provide a model of a language. The semantics are given by exact rules which state what objects satisfy a specification. For example, Cartesian Geometry shows how theorems in Euclidean Geometry can be modeled by algebraic expressions. A language can have several models, but some will seem more natural than others.

A specification is a set of formulae in a formal language. The objects in the language's semantic domain that satisfy a given specification can be nonunique. Several objects may be equivalent as far as a particular specification is concerned. Because of this nonuniqueness, the specification is at a higher level of abstraction than the objects in the semantic domain. The specification language permits abstraction from details that distinguish different implementations, while preserving essential properties. Different specification methods defined over the same semantic domain allow for specifying different aspects of specified objects. These concepts can be defined more precisely using mathematics. The advantage of this mathematics is that it provides tools for formal reasoning about specifications. Specifications can then be examined for completeness and consistency.

Specification languages can be classified by their semantic domains. Three major classes of semantic domains exist [Wing 90]:

- Abstract Data Type specification languages
- Process specification languages
- Programming languages

ADT specification languages can be used to specify algebras. An ADT "defines the formal properties of a data type without defining implementation features" [Vienneau 91]. Z, the Vienna Development Method, and Larch are examples of ADT specification languages. Process specification languages specify state sequences, event sequences, streams, partial orders, and state machines. C. A. R. Hoare's Communicating Sequential Processes (CSP) is the most well-known process specification language.

Programming languages provide an obvious example of languages with multiple models. As explained in Section 4, predicate transformers provide one model. Functions provide another model, as explained in Section 5. The executable machine instructions that are generated by compiling a program provide a third model. Formal methods are useful in programming because programs can be viewed both as a set of commands for physical machines and as abstract mathematical objects as provided by these alternative models.

3.2 Operational and Definitional Methods

The distinction between operational and definitional methods provides another important dimension for classifying formal methods [Avizienis 90]. Operational methods have also been described as constructive or model-oriented [Wing 90]. In an operational method, a specification describes a system directly by providing a model of the system. The behavior of this model defines the desired behavior of the system. Typically, a model will use abstract mathematical structures, such as relations, functions, sets, and sequences. An early example of a model-based method is the specification approach associated with Harlan Mills' functional correctness approach. In this approach, a computer program is defined by a function from a space of inputs to a space of outputs. In effect, a model-oriented specification is a program written in a very high-level language. It may actually be executed by a suitable prototyping tool.

Definitional methods are also described as property-oriented [Wing 90] or declarative [Place 90]. A specification provides a minimum set of conditions that a system must satisfy. Any system that satisfies these conditions is functionally correct, but the specification does not provide a mechanical model showing how to determine the output of the system from the inputs. Two classes of definitional methods exist, algebraic and axiomatic. In algebraic methods, the properties defining a program are restricted to equations in certain algebras. Abstract Data Types are often specified by algebraic methods. Other types of axioms can be used in axiomatic methods. Often these axioms will be expressed in the predicate calculus. Edsger Dijkstra's method of specifying a program's function by preconditions and postconditions is an early example of an axiomatic method.

3.3 Use of Specification Methods

Different specification methods are more advantageous for some purposes than others. In general, formal methods provide for more precise specifications. Misunderstandings and bugs can be discovered earlier in the lifecycle. Since the earlier a fault is detected, the cheaper it can be removed, formal specification methods can dramatically improve both productivity and quality. Cost savings can only be achieved if formal methods are used appropriately. How to best use them in a specific environment can only be determined through experimentation.

Formal specifications should not be presented without a restatement of the specification in a natural language. In particular, customers should be presented with the English version, not a formal specification. Very few sponsors of a software development project will be inclined to read a specification whose presentation is entirely in a formal language.

Whether an ADT or process specification language should be adopted depends on the details of the project and the skills of the analysts. Choosing between operational and definitional methods also depends on project-specific details and experience. Generally, programmers are initially more comfortable with operational methods since they are closer to programming. Operational specifications may lead to overspecification. They tend to be larger than definitional specifications. Their complexity thus tends to be greater, and relationships among operations tend to be harder to discern.

Definitional specifications are generally harder to construct. The appropriate axioms to specify are usually not trivial. Consistency and completeness may be difficult to establish. Usually completeness is more problematic than consistency. Intuition will tend to prevent the specification of inconsistent axioms. Whether some axioms are redundant, or more are needed, is less readily apparent. Automated tools are useful for guidance in answering these questions [Guttag 77].

4.0 Conclusions

This report has briefly surveyed various formal methods and the conceptual basis of these techniques. Formal methods can provide:

- More precise specifications
- Better internal communication
- An ability to verify designs before executing them during test
- Higher quality and productivity

These benefits will come with costs associated with training and use. Hard and fast rules do not exist on how to properly vary the level of formalism on a project or on how to transition the use of formal methods into an organization. Their enthusiastic use certainly depends on the organization's members perceiving a need that formal methods can fill. No change is likely to be achievable in an organization that is satisfied with its current practice.

Even if formal methods are not integrated into an organization's process, they can still have positive benefits. Consider a group whose members have been educated in the use of formal methods, but are not encouraged to use formal methods on the job. These programmers will know that programs can be developed to be fault-free from the first execution. They will have a different attitude to both design and testing, as contrasted to programmers who have not been so exposed to formal methods. They will be able to draw on a powerful set of intellectual tools when needed. They will be able to use formal methods on a personal basis and, to a limited extent, to communicate among one another. If management provides the appropriate milieu, this group can be expected to foster high quality attitudes with consequent increases in both productivity and quality.

To get their full advantages, formal methods should be incorporated into a software organization's standard procedures. Software development is a social process, and the techniques employed need to support that process. How to fully fit formal methods into the lifecycle is not fully understood. Perhaps there is no universal answer, but only solutions that vary from organization to organization.

4.1 The Cleanroom as a Lifecycle with Integrated Use of Formal Methods

Harlan Mills has developed the Cleanroom methodology [Mills 87], which is one approach for integrating formal methods into the lifecycle. The Cleanroom approach combines formal methods and structured programming with Statistical Process Control (SPC), the spiral lifecycle and incremental releases, inspections, and software reliability modeling. It fosters attitudes, such as emphasizing defect prevention over defect removal, that are asociated with high quality products in non-software fields.

Cleanroom development begins with the requirements phase. Ideally, specifications should be developed in a formal language, although the Cleanroom approach allows the level of formality to vary. The Cleanroom lifecycle uses incremental releases to support SPC. Cleanroom-developed specifications include:

- Explicit identification of functionality to be included in successive releases
- Failure definitions, including levels of severity
- The target reliability as a probability of failure-free operation
- The operational profile for each increment, that is, the probability distribution of user inputs to the system
- The reliability model that will be applied in system test to demonstrate reliability

The design and coding phases of Cleanroom development are distinctive. Analysts must develop proofs of correctness along with designs and codes. These proofs use the functional correctness techniques described in Section 5 and are meant to be human-readable. They serve a social role and are not intended to be automatically checked by automated verification tools. Inspections are emphasized for reviewing designs, proofs, and code. The design process is intended to prevent the introduction of defects. In keeping with this philosophy, the Cleanroom methodology includes no unit or integration test phase. In fact, coders are actually forbidden to compile their programs. Cleanroom development takes its name from just this aspect of the methodology. Testing is completely separated from the design process, and analysts are not permitted to adopt the attitude that quality can be tested in. Instead, they must produce readable programs which can be convincingly shown correct by proof.

Testing does play a very important role in Cleanroom development. It serves to verify that reliability goals are being attained. Given this orientation, testing is organized differently than in traditional methods. Unit and integration testing do not exist. Functional methods, not structural testing methods, are employed. Furthermore, the testing process is deliberately designed to meet the assumptions of the chosen software reliability model. Test cases are statistically chosen from the specified operational profile. Although faults are removed when detected, the testing group's responsibility is not to improve the product to meet acceptable failure-rate goals. Rather, the testing group exists to perform reliability measurement and certification.

When testing fails to demonstrate the desired reliability goal is met, the design process is altered. The level of formality may be increased, or more inspections may be planned. Testing and incremental builds are combined to provide feedback into the development process under a Statistical Process Control philosophy as tailored for software. Formal methods are embodied in an institutional structure designed to foster a "right the first time" approach. The Cleanroom methodology draws upon evolving concepts of the best practice in software, including formal methods. The Cleanroom approach is beginning to generate interest and experimentation in organizations unassociated with Harlan Mills and International Business Machines [Selby 87].

4.2 Technologies Supported by Formal Methods

Researchers are drawing on formal methods in developing tools and techniques that may not be state of the practice for several years. Lifecycle paradigms that rely on automatically transforming specifications to executable code are necessarily formal. Many software development tools, whether standalone or integrated into a common environment, draw on formal methods. Consequently, as software development becomes more tool intensive, formal methods will be more heavily used. Inasmuch as these formal methods are embodied in the tools, tool users may not be fully aware of the embedded formalism.

Tool users who are trained in formal methods will be able to wield some of these tools more effectively. Formal methods, through their use in tools, have the promise of being able to transform the software development lifecycle from a labor-intensive error-prone process to a capital-intensive high quality process.

Emerging technologies that are increasingly widespread today also draw on formal methods. A knowledge of formal methods is needed to completely understand these popular technologies and to use them most effectively. These technologies include:

- Rapid prototyping
- Object Oriented Design (OOD)
- Structured programming
- Formal inspections.

Rapid prototyping depends on the ability to quickly construct prototypes of a system to explore their ability to satisfy user needs. Using executable specifications to describe a system at a high level is a typical approach. The tool that compiles the specification fills in the details. Specifications constructed under a rapid prototyping methodology, if executable, are by definition in a formal language. Often the languages used in prototyping tools involve the same set theoretical and logical concepts used in formal specification methods not intended for prototyping.

OOD is another increasingly well-known technology that is based on formal methods. Abstract Data Types provide a powerful basis for many classes in Object Oriented systems. Furthermore, at least one pure object oriented language, Eiffel, has assertions, preconditions, postconditions, and loop invariants built into the language to a certain extent. Simple boolean expressions are checked during execution of an Eiffel program, but not all assertions, such as those with existential and universal quantifiers, can be expressed in the language [Meyer 88]. Thus, formal methods can be usefully combined with object oriented techniques.

The connection between formal methods and structured programming is very close. Structured programming is a set of heuristics for producing high quality code. Only a limited set of constructs should be used. Programs should be developed in a top-down fashion. The historical source for these heuristics lies in formal methods. Programs developed with these precepts will be capable of being rigorously proven correct. Consequently, they will also be capable of being understood intuitively and nonrigorously. Structured programming cannot be completely understood without understanding the rigorous mathematical techniques associated with formal methods. Adopting formal methods is a natural progression for software development teams who employ structured programming techniques.

Inspections throughout the lifecycle have been shown to increase both productivity and quality. A rigorous methodology has been defined for inspections [Fagan 76]. Those participating in inspections play specified roles: moderator, author, coder, tester, etc. Inspections should be organized to include representatives from specified departments (for example, Quality Assurance) within a software organization. Fault data is collected during inspections and analyzed to ensure the development process is under control. Inspections rely on the ability of individuals to reason about software products and to convince others of the correctness of their reasoning. Training in formal methods provides inspection team members with a powerful language to communicate their trains of reasoning. Formal and semi-formal verifications can lead to more effective inspections. The Cleanroom methodology demonstrates the potential synergy between formal methods and inspections.

4.3 Summary

Formal methods promise to yield benefits in quality and productivity. They provide an exciting paradigm for understanding software and its development, as well as a set of techniques for use by software engineers. Over the last twenty years, researchers have drawn on formal methods to develop certain software technologies that are currently becoming increasingly popular and are dramatically altering software development and maintenance. Further revolutionary advances based on formal methods are highly likely considering research currently in the pipeline.

Many organizations have experience with the use of formal methods on a small scale. Formal methods are typically used in organizations attaining a Level 3 rating and above on the Software Engineering Institute's process maturity framework. Increasingly, recently trained software engineers have had some exposure to formal methods. Nevertheless, their full scale use and transition is not fully understood. An organization that can figure out how to effectively integrate formal methods into their current process will be able to gain a competitive advantage.

Table of Contents NEXT

DoD DACS Home Page Technical Reports

5.0 References

[Agresti 86] William W. Agresti, *New Paradigms for Software Development,* IEEE Computer Society, 1986.

[Aho 86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, 1986.

[Avizienis 90] Algirdas Avizienis and Chi-Sharn Wu, "A Comparative Assessment of Formal Specification Techniques," *Proceedings of the 5th Annual Knowledge-Based Software Assistant Conference*, Liverpool, NY, September 24-28, 1990.

[Baber 91] Robert L. Baber, *Error-Free Software: Know-how and Know-why of Program Correctness,* John Wiley & Sons, 1991.

[Backus 78] John Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM,* Volume 21, Number 8, August 1978.

[Boehm 81] Barry W. Boehm, *Software Engineering Economics,* Prentice- Hall, Inc., 1981.

[Curtis 88] Bill Curtis, Herb Krasner, and Neil Iscoe, "A Field Study of the Software Design Process for Large Systems," *Communications of the ACM,* Volume 31, Number 11, November 1988.

[DeMillo 79] R. DeMillo, R. Lipton, and A. Perlis, "Social Processes and Proofs of Theorems and Programs," *Communications of the ACM,* Volume 22, Number 5, May 1979.

[DeRemer 76] Frank DeRemer and Hans H. Kron, "Programming-in-the-Large Versus Programming-in-the-Small," *IEEE Transactions on Software Engineering*, Volume SE-2, Number 2, June 1976.

[Dijkstra 76] Edsger W. Dijkstra, *A Discipline of Programming,* Prentice Hall, 1976.

[Dijkstra 89] Edsger W. Dijkstra, "On the Cruelty of Really Teaching Computer Science," *Communications of the ACM,* Volume 32, Number 12, December 1989.

[Dyer 92] Michael Dyer, *The Cleanroom Approach to Quality Software Development,* John Wiley & Sons, 1992.

[Fagan 76] M. E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal,* Volume 15, Number 3, 1976.

[Fetzer 88] James H. Fetzer, "Program Verification: The Very Idea," *Communications of the ACM,* Volume 31, Number 9, September 1988.

[Goel 91] Amrit L. Goel and Swarupa N. Sahoo, "Formal Specifications and Reliability: An Experimental Study," *Proceedings of the International Symposium on Software Reliability Engineering,* Austin, TX, May 1991.

[Gries 81] David Gries, *The Science of Programming,* Spring-Verlag, 1981.

[Gries 91] David Gries, "On Teaching and Calculation," *Communications of the ACM,* Volume 34, Number 3, March 1991.

[Guttag 77] John Guttag, "Abstract Data Types and the Development of Data Structures," *Communications of the ACM,* Volume 20, Number 6, June 1977.

[Hall 90] Anthony Hall, "Seven Myths of Formal Methods," *IEEE Software,* September 1990.

[Hoare 85] C. A. R. Hoare, *Communicating Sequential Processes,* Prentice-Hall International, 1985.

[Hoare 87] C. A. R. Hoare, "Laws of Programming," *Communications of the ACM,* Volume 30, Number 8, August 1987.

[Humphrey 88] Watts S. Humphrey, "Characterizing the Software Process: A Maturity Framework," *IEEE Software,* Volume 5, Number 2, March 1988.

[KBSE 92] *Proceedings of the Seventh Knowledge-Based Software Engineering Conference,* McLean, VA, September 20-23, 1992.

[Kline 80] Morris Kline, *Mathematics: The Loss of Certainty,* Oxford University Press, 1980.

[Lamport 89] Leslie Lamport, "A Simple Approach to Specifying Concurrent Systems," *Communications of the ACM,* Volume 32, Number 1, January 1989.

[Leveson 90] N. G. Leveson, "Guest Editor's Introduction: Formal Methods in Software Engineering," *IEEE Transactions in Software Engineering,* Volume 16, Number 9, September 1990.

[Linger 79] R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming: Theory and Practice,* Addison-Wesley Publishing Company, 1979.

[London 77] Ralph L. London, "Remarks on the Impact of Program Verification on Language Design," in *Design and Implementation of Programming Languages,* Springer-Verlag, 1977.

[Lyons 77] John Lyons, *Noam Chomsky,* Penguin Books, Revised Edition 1977.

[McGettrick 82] Andrew D. McGettrick, *Program Verification using Ada,* Cambridge University Press, 1982.

[Merrill 83] Gary Merrill, "Proofs, Program Correctness, and Software Engineering," *SIGPLAN Notices,* Volume 18, Number 12, December 1983.

[Meyer 85] Bertrand Meyer, "On Formalism in Specifications," *IEEE Software,* Volume 2, Number 1, January 1985.

[Meyer 88] Bertrand Meyer, *Object-Oriented Software Construction,* Prentice-Hall, 1988.

[Mills 86] Harlan D. Mills, "Structured Programming: Retrospect and Prospect," *IEEE Software,* Volume 3, Number 6, November 1986.

[Mills 87] Harlan D. Mills, Michael Dyer, and Richard C. Linger, "Cleanroom Software Engineering," *IEEE Software,* September 1987.

[Peterson 77] James L. Peterson, "Petri Nets," *Computing Surveys*, Volume 9, Number 3, September 1977.

[Place 90] Patrick R. H. Place, William Wood, and Mike Tudball, *Survey of Formal Specification Techniques for Reactive Systems,* Software Engineering Institute, CMU/SEI-90-TR-5, May 1990.

[Preston 88] David Preston, Karl Nyberg, and Robert Mathis, "An Investigation into the Compatibility of Ada and Formal Verification Technology," *Proceedings of the Sixth National Conference in Ada Technology*, Arlington, VA, 1988.

[Selby 87] Richard W. Selby, Victor R. Basili, and F. Terry Baker, "Cleanroom Software Development: An Empirical Evaluation," *IEEE Transactions on Software Engineering,* Volume SE-13, Number 9, September 1987.

[Spivey 88] J. M. Spivey, *Understanding Z: A Specification Language and its Formal Semantics*, Cambridge University Press, 1988.

[Stolyar 70] Abram Aronovich Stolyar, *Introduction to Elementary Mathematical Logic,* Dover Publications, 1970.

[Suppes 72] Patrick Suppes, *Axiomatic Set Theory,* Dover Publications, 1972.

[Terwilliger 92] Robert B. Terwilliger, "Simulating the Gries/Dijkstra Design Process," *Proceedings of the Seventh Knowledge-Based Software Engineering Conference*, McLean, VA, September 20-23, 1992.

[Vienneau 91] Robert Vienneau, *An Overview of Object Oriented Design*, Data & Analysis Center for Software, April 30, 1991.

[Wing 90] Jeannette M. Wing, "A Specifier's Introduction to Formal Methods," *Computer*, Volume 23, Number 9, September 1990.

[Zeroual 91] K. Zeroual, "KBRAS: A Knowledge-Based Requirements Acquisition System," *Proceedings of the 6th Annual Knowledge-Based Software Engineering Conference, Syracuse*, NY, September 22-25, 1991.

DoD DACS Home Page

Technical Reports

Table of Contents

A Review of Formal Methods

Abstract:

Technical reports addressing a variety of software engineering topics are available from the DACS. Our newest report, A Review of Formal Methods, examines a topic of increasing interest, formal methods. Formal methods provide a foundation for many of the techniques that have changed the face of software development over the last two decades. Structured programming, top down development, abstract data types, rapid prototyping, object oriented design, and formal inspections all draw on formal methods.

Formal methods imply a distinctive view of software development. According to this view, the software engineers task is to produce several models or descriptions of a system for an abstract machine, with accompanying proofs that models at lower levels of abstraction correctly implement higher-level models. The job of the hardware engineer, language designer, and compiler writer is to provide a realization suitable for executing the applications developers abstract model, but it is not the application programmers job to twist his logic to fit inadequate physical automata. Only this idealized design process can ensure high levels of quality, not testing. Testing can demonstrate the presence of faults, not their absence.

A formal method in software is a method that provides a formal language for describing a software artifact (for example, specifications, designs, or source code) such that formal proofs are possible, in principle, about properties of the artifact so expressed. Formal methods involve the essential use of a formal language, that is, a set of strings over some well- defined alphabet. Rules are given for distinguishing strings in the language from those outside. Formal methods in software support formal reasoning about formulae in the language. Formal proofs are based on a set of axioms, which are statements postulated to be true, and inference rules. A proof consists of a sequence of well-defined formulae in the language in which each formula is either an axiom or derivable by an inference rule from previous formulae in the sequence. The last axiom in the sequence is said to be proven. To be practically useful, formal method should provide heuristics and guidelines for developing elegant specifications and for developing implementations and proofs in parallel. The report overviews the technical basis for formal methods, while noting weaknesses.

Formal methods in software draw upon some advanced mathematics. Mathematical topics of interest include formal logic, set theory, formal languages, and automata. Formal methods support precise and rigorous specifications of those aspects of a computer system capable of being expressed in the available formal languages. Since defining what a system should do, and understanding the implications of these decisions, are the most troublesome problems in software engineering, this use of formal methods has major benefits. In fact, practitioners frequently use formal methods solely for recording precise specifications, not for formal verifications.

Two formal verification styles are presented in this report, as well as an overview of specification methods. One verification approach is based on demonstrating assertions about the variables within a program at specific points in the program. Important advocates of this approach include Edsger Dijkstra, David Gries, and C.A.R. Hoare. The other style was developed by Harlan Mills and is based on showing the equivalence of a program and a mathematical function.

Formal methods can provide more precise specifications, better internal communications, an ability to

verify designs before execution, and higher quality and productivity. Limitations include an inability to formally demonstrate that a specification captures an users intuitive understanding of a system, the need for training, and difficulties in integrating formal methods into the software development process with the capability of varying the level of formality.

Ordering Information:
A Review of Formal Methods, which includes an 81 page annotated bibliography, is available for **$30** from the DACS.

It may be ordered on-line or by contacting the DACS at:
Phone: (315) 334-4905.
E-mail: cust-liasn@dacs.dtic.mil
