A Review of Formal Methods

Robert L. Vienneau

INTRODUCTION

The 1970s 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 the 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, object-oriented design, aspect-oriented design, and model-driven development 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.

A set of core ideas 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 paper 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.

The underlying philosophy of formal methods has not changed drastically in a quarter century. Nevertheless, this approach is a revolutionary paradigm shift from conventional notions about computer programming. Many software engineers may have adopted the new methodologies without fully 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, interfaces—with the user, libraries, and other systems—can be expected to introduce additional complexities. In the traditional view of programming, these details may appear in a design or 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 designer of the development environment 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) sufficient 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 1976)

The contrast between these views is controversial.[1] Advocates of formal methods argue that many have adopted structured programming and top-down development without really understanding the underlying formalism (Mills 1986). A concern with formal methods can produce more rigorous specifications, even if they are expressed in English (Meyer 1985). Design 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 interfaces and error-checking code), and the radical retraining needed for the large population of already existing software engineers.

This paper provides an overview of formal methods from a neutral point of view. It surveys the technical basis for formal methods, while critically noting weaknesses. Polemics are avoided, but enough information is provided to assist the reader to form a knowledgeable judgment on formal methods. Formal methods are slowly beginning to see more widespread use, espe-

---

This paper is based on a paper with a similar name published in Merlin Dorfman and R. H. Thayer (eds.), *Software Engineering,* IEEE Computer Press, Los Alamitos, CA, 1997.

[1]See the discussion engendered in the ACM Forum by DeMillo (1979), Fetzer (1988), Dijkstra (1989), Gries (1991), or Glass (2002).

cially in Europe. Their use is characteristic of organizations with more mature processes, as assessed, for example, by the Capability Maturity Model (CMM) developed by the Software Engineering Institute (Paulk 1995). Formal methods have the potential of leading to further revolutionary changes in practice and have provided the underlying basis for past changes. These reasons make it imperative that software managers and engineers be aware of the increasingly widespread debate over formal methods.

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 1990)

A more narrow definition, however, better conveys the change in practice recommended by advocates of formal methods. The definition offered here is based on Wing's (1990), and has two essential components. First, formal methods involve the 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 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 axiom 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 (for instance, specifications, designs, or 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, that it fulfills the functional components of 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, one for describing a specification and the other for describing 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 include Karl Weierstrass, Gottlob Frege, Giuseppe Peano, and David Hilbert. By the start of the 20th century, a foundation seemed to be in place, but certain strange examples and anomalies caused mathematicians to question the security of their foundations and even their own intuitions on fundamentals. A mechanical method of manipulating symbols was thus invented to investigate these questions. Hilbert challenged his mathematical colleagues to prove that this method was complete and could not be used to derive contradictions. Due to fundamental discoveries of Thoralf Skolem and Leopold Löwenheim,[2] and of Kurt Gödel,[3] mathematicians accepted that Hilbert's program could not be completed. 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 1980).

Formal methods are an adoption of the axiomatic method, as developed by these trends in mathematics, for software engineering (MacKenzie 2001). In fact, Edsger Dijkstra (1989) suggested, somewhat tongue-in-cheek, that computer science be renamed Very Large Scale Application of Logic (VLSAL). 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.

---

[2]The Löwenheim-Skolem theorem states that any model, under certain general conditions, has a countable submodel, thus revealing that a set of axioms cannot uniquely characterize a model, in some sense.

[3]Gödel showed that any formal system encompassing arithmetic contains a true statement that cannot be formally proved within the system by finitary means. A corollary is that such a system cannot be proven consistent within the system.

Use of Formal Methods

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 tools. They are important in ongoing research that may change standard practice, particularly in the areas of specifications and design methodology. They are entwined with life-cycle models that provide an alternative to the waterfall model, namely rapid prototyping, the Cleanroom variant on the spiral model, and "transformational" paradigms such as Model-driven development.

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 1990, Place 1990).

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

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

Formal methods can be used to specify aspects of a system other than functionality. For example formal methods are sometimes applied in practice to ensure software safety[4] and security properties[5] of computer programs. The benefits of proving that unsafe states cannot arise, or that security is assured, can justify the cost of complete formal verifications of the relevant portions of a software system.[6] Formal methods can deal with other areas of concern to software engineers but, other than in research organizations, they have not been much used 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 1990).

Formal methods can include graphical languages. The twelve diagram types[7] defined by the Unified Modeling Language (UML) can be considered as, at least, semiformal languages. Petri nets provide another well-known graphical technique, often used in distributed systems (Peterson 1977). 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 above definition of formal methods is more encompassing than might be first apparent.

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) are frequently 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 (Cohn 1989), such as those based on VHSIC Hardware Description Language (VHDL) descriptions, to model integrated circuits before fabricating them.

Reasoning about a Formal Description

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 si-

[4]Formal proofs are appropriate for Safety Integrity Level 4, the highest of four SIL levels specified in IEC 61508.

[5]Formal models are required for ITSEC 4, 5, and 6, the highest assurance levels under the United Kingdom Information Technology Security Evaluation and Certification Scheme. ISO 15408, an international standard, also specifies assurance levels.

[6]Stepney (1998) reports on two industrial-scale proofs, one for a safety-critical compiler and the other for security properties of a product for electronic transactions.

[7]The class, object, component, and deployment diagrams for system structure; use case, sequence, activity, collaboration, and state chart diagrams for system behavior; and package, subsystem, and model diagrams for model management.

multaneously? Do all legal inputs that 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.

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 principal driver in the development of formal methods. Classic technologies for formal verification include Edsger Dijkstra's "weakest precondition" calculus (Dijkstra 1976, Gries 1981) and Harlan Mills' "functional correctness" approach (Linger 1979).

Tools and Methodology

Developments in supporting tools and methodologies have accompanied new concepts for formalizing software artifacts. For proponents of formal methods, the ultimate end product of software development is not solely a working system. Specifications and demonstrations that the program meets its specification are of equal importance. A proof is very hard to develop after the fact. Consequently, proofs and programs should be developed in parallel, with close interconnections in their 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.

Applying these ideas on large-scale projects is a challenge. Formal specifications seem to scale up 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 (1981) recommends a design methodology incorporating certain heuristics that support more reliable and provable designs. Harlan Mills (Mills 1987 or Dyer 1992, for example) spent considerable effort developing the Cleanroom approach, a life cycle in which formal methods, inspections, and reliability modeling and certification are integrated in a social process for producing software.

Formal methods have also inspired the development of many tools. These tools may bring formal methods into more widespread practice. Interestingly enough, some early advocates of formal methods were not strong believers in tools. Programs to help maintain and automate proofs are an obvious example of such tools. 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.[8]

In some sense, no programmer can avoid formal methods, for every programming language is, by definition, a formal language. Ever since Algol 1960 was introduced, standards defining programming languages have used a formal notation for defining language syntax, namely Backus-Naur Form (BNF). Usually, standards do not formally define the semantics of programming languages, although, in principle, they could. The convention of using natural language descriptions as definitions of language semantics is due to not having yet settled on techniques for defining all constructs in large languages. Nevertheless, formal methods have resulted in one widely agreed upon 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 were always an interest of the Ada community, even before standardization (London 1977, McGettrick 1982, and Preston 1988). More recently, the Java Modeling Language (JML) project provides tools to add annotations to Java source code to document in application programmer's interfaces (APIs) the intended behavior of Java classes (Burdy 2003).

Limitations of Formal Methods

Given the applicability of formal methods throughout the life cycle, and their pervasive possibilities for almost all areas of software engineering, why are they not more widely visible?[9] One issue is pedagogic. Revolutions are not made by conversion, but by the old guard passing away. Recent university graduates will have seen structured programming and object-oriented design from the start of their careers and should be more willing to experiment with formal methods.

On the other hand, it is not the case that the only barrier to the widespread transition of this technology is lack of knowledge on the part of practitioners. Formal methods 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.

---

[8]Discussion of transformational approaches can be found in Agresti (1986); Vienneau (2003); the annual IEEE Conference on Automated Software Engineering, the eighteenth having been held in Montreal, Canada, on 6–10 October 2003; and the September/October 2003 issue of *IEEE Software,* which is focused on model-driven development.

[9]Formal Methods Europe (FME), whose Web page is at http://www.fmeurope.org, maintains a list of applications of formal methods.

The Requirements Problem

The inherent limitations are neatly summarized in the oft-repeated aphorism, "You cannot go from the informal to the formal by formal means." In particular, a formal verification can prove that an implementation satisfies a formal specification, but it cannot prove that a formal specification captures a user's intuitive understanding of a system. In other words, formal methods can be used to verify a system, but not to validate it. The distinction is that validation shows that a product will satisfy its operational mission, whereas verification shows that each step in the development satisfies the requirements imposed by previous steps (Boehm 1981).

The extent of this limitation should not be underemphasized. One influential study (Curtis 1988) found that the three most important problems in software development are:

1. The thin spread of application domain knowledge
2. Changes in and conflicts between requirements
3. 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 suggests, 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, implementers without an exceptional designer's knowledge of the application area commit fewer errors when implementing a formal specification than when relying on hazy knowledge of the application (Goel 1991). These benefits may exist even when the final specification is expressed in English, not a formal language (Meyer 1985). 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 (Zeroual 1991, for example), have developed tools to interactively capture a user's informal understanding and thereby develop a formal specification.

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 more experience and experimentation.

Physical Implementations

The second major gap between the abstractions of formal methods and concrete reality lies in the nature of any 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 useful for humanly readable correctness proofs. For instance, an abstract machine might be assumed to have an infinite memory, whereas every actual machine has some upper limit. Similarly, physical machines cannot implement real numbers, as constructed by mathematicians, whereas proofs are often most simply constructed assuming the existence of mathematically precise real numbers. And proofs may abstract from interrupts used in time-sharing systems. No reason exists in principle why formal methods cannot incorporate some of these limitations. The proofs will be messier and less elegant, and they will be limited to a particular class of architectures.[10]

A limitation in principle, however, exists 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.[11] Memory chips and integrated circuits may contain bugs. No matter how thoroughly an application is formally verified, at some

[10]For example, those computers implementing IEEE 754.

[11]The founders of Computational Logic, Incorporated, a company existing between 1983 and 1997, developed a microprocessor that was formally verified down to the gate level. See http://www.cli.com/.

point one must just accept that an actual physical system satisfies the axioms used in a proof. Explanations must come to an end sometime.

Both critics (Fetzer 1988) and developers (Dijkstra 1976, Linger 1979) of formal methods are quite aware of this limitation, although the critics do not always seem to accept the wording of developers' explicit statements on this point. This limitation does not imply that 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 1983).

Although no prominent advocate of formal methods recommends that 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.[12] From an abstract point of view, the question of what knowledge or degree of rational belief can be provided by testing is the riddle of the 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 correct outputs for some inputs ever lead to a belief that the program is likely to produce the correct outputs 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? The British philosopher David Hume put related questions at the center of his epistemology over two centuries ago.

Two centuries of debate have not brought about 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 is likely to change.

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, as to 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 the most 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 cost-effective 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 (1986) claimed his program function approach applies more easily to large systems than Dijkstra's competing predicate calculus method. Likewise, the languages considered in academic work tend to be simplified, as compared to real-world programming languages.

One scheme for using formal methods on real-world projects is to select a small subset of components for formal treatment,[13] thus finessing the scalability issue. These components might be selected under criteria of safety, security, or criticality. Components particularly amenable to formal modeling might be specifically selected. In this way, the high cost of formal methods is avoided for the entire project, and is only incurred where project requirements justify it.

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 that the implementation satisfies the specification. Rather, one designs implementations and proofs in parallel, with continual interaction. Sometimes, discussions about automated verifiers (see, for example, DeMillo 1979 and Merrill 1983) suggest that the former approach, not the latter, provides an implementation model of formal methods.

Another approach can have more global impacts. Perhaps the waterfall lifecycle should be scrapped. An alternative is to develop formal specifications at the start of the life cycle and then automatically derive the source code for the system. Mainte-

[12]Model-based testing is a methodology in which a formal model of user behavior, typically employing finite state machines, Markov processes, or decision tables, automatically generates test cases. El-Far (2001) provides an overview.

[13]For example, the kernel in an operating system.

nance, enhancements, and modifications are 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 provides the inspiration for many tool developers.

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. 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, specific testing approaches, reliability certifications, 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. Many 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 in undergraduate computer science courses. Even discrete mathematics, a standard course whose place in the university curriculum owes much to the impetus of computer science professional societies, is sometimes 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 education issue[14] is the biggest hurdle to the widespread transition of formal methods.

SPECIFICATION METHODS

Formal methods were originally developed to support verifications, but many projects using formal methods have used them only to establish properties of specifications. Several methods and languages can be used to specify the functionality of computer systems. No single language, of those now available, seems appropriate for all methods, application domains, and aspects of a system. Thus, users of formal specification techniques need to understand the strengths and weaknesses of different methods and languages before deciding on which to adopt. This section briefly describes some 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 1989). Some languages support more than one method, whereas most methods can be used in several specification languages. Some methods are more easily used with certain languages.

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.

Meanings, or interpretations of formulae, are provided by the semantics of a language. A set of objects, known as the language's semantic domain, provides 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 most will find some models 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, in some sense, 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 the mathemat-

[14]http://www.cs.indiana.edu/formal-methods-education/ is an online repository of education materials for formal methods.

ics is that it provides tools for reasoning about specifications. Specifications can then be examined for completeness and consistency.

Specification languages can be classified based on their semantic domains. Three major classes of semantic domains exist (Wing 1990):

1. Abstract data type (ADT) specification languages
2. Process specification languages
3. Programming languages

ADT specification languages can be used to describe algebras. An ADT "defines the formal properties of a data type without defining implementation features"(Vienneau 1991). Z, the Vienna Development Method, and Larch are examples of ADT specification languages. Process specification languages describe state sequences, event sequences, streams, partial orders, and state machines. C. A. R. Hoare's Communicating Sequential Processes (CSP) is a classic process specification language.

Programming languages provide an obvious example of languages with multiple models. Predicate transformers provide one model, functions provide another model, and 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 as both a set of commands for physical machines and as abstract mathematical objects, as provided by these alternative models.

Model-Oriented and Property-Oriented Methods

The distinction between model-oriented and property-oriented methods provides another dimension for classifying formal methods (Avizienis 1990). Model-oriented methods have also been described as constructive or operational (Wing 1990). In a model-oriented 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. The specification technique associated with Harlan Mill's functional correctness approach is an early example of a model-oriented method. 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. A suitable prototyping tool may actually execute it.

Property-oriented methods are also described as definitional (Wing 1990) or declarative (Place 1990). A specification describes 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 property-oriented methods exist, algebraic and axiomatic. In algebraic methods, the properties defining a program are 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 or higher-order logics. Edsger Dijkstra's method of specifying a program's function by preconditions and postconditions is an early example of an axiomatic method.

Use of Specification Methods

In general, formal methods provide for more precise specifications. Misunderstandings and bugs can be discovered earlier in the life cycle. Since the earlier a fault is detected, the cheaper it can be removed, formal specification methods can dramatically improve both productivity and quality. How to best use formal methods in a specific environment will 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 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 model-oriented and property-oriented methods also depends on project-specific details and experience. Generally, programmers are initially more comfortable with model-oriented methods since they are closer to programming. Model-oriented specifications may lead to overspecification. They tend to be larger than property-oriented specifications. Their complexity thus tends to be greater, and relationships among operations tend to be harder to discern.

Property-oriented specifications are generally harder to construct. The appropriate axioms to specify are usually not trivial and consistency and completeness may be difficult to establish. Usually, completeness is more problematic than consistency. Intuition will tend to prevent the specification of obviously 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 1977).

LIFE CYCLES AND TECHNOLOGIES WITH INTEGRATED FORMAL METHODS

To get their full advantages in a cost-effective manner, 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. Two methods of integrating formal methods in software processes can be distinguished: one with heavy use of automated tools and the other with nonmechanical, nonautomated proofs. The following subsections discuss tools for supporting formal methods and the Cleanroom methodology as an exemplar of the nonmechanized use of formal methods. Tools are described for theorem proving, for model checking, and for refinement of specifications into implementations. The division of verification tools into theorem proving and model checking tools is a well-established classification system in formal methods (Clarke 1996).

Verification Systems and Other Automated Tools

An automated verification system provides a means for the user to demonstrate the existence of a formal proof of a software system. Robert S. Boyer and J Strother Moore's Nqthm system, Robert L. Constable and Joseph L. Bates' Proof Refinement Logics (PRL), Robin Milner's Logic for Computable Functions (LCF), and the Larch Prover (Garland 1991) are some examples of early verification systems. A Computational Logic for Applicative Common Lisp ACL2 (Kaufmann 1997) is a more recent system descended from Boyer and Moore's prover. Nuprl (Liu 1999, for example) draws on both PRL and LCF. Variants on Michael Gordon's Higher Order Logic (HOL) provide another important family (see http://www.cl.cam.ac.uk) of recent provers.

The usage of verification systems varies. Some, such as the Larch Prover, allow the user a high level of control. The user supplies intermediate lemmas to be proven. The system is useful, so to speak, for bookkeeping. Others use more automation. The more automated systems often use techniques, such as resolution, that produce nonintuitive results. Verifiers don't necessarily produce a formal proof with each step following from the axioms. They often use decision procedures in restricted domains. Decision procedures show whether or not a proof exists without explicitly constructing the proof.

Another set of tools supports model checking. In model checking, one creates a state transition diagram as a model of a specification. The specification also provides assertions of some properties, which are expressible in temporal logic. Model checking either establishes that the properties are true for the state transition diagram model or provides a counterexample. Model checking tools overcome state explosion problems in practice by the use of symbolic techniques. Symbolic model checking tools do not need to visit every individual state, but can explore sets of states at a time. Binary decision diagrams (BDDs), a generalization of binary decision trees, provide one common method for the internal representation in model checkers of state transition diagrams (Chan 1998). Model checking was first developed for hardware design. It has been applied industrially to large software projects. Properties explored in model checking can address safety (Wang 2004), the correctness of reactive systems (Chandra 2002), and system-level properties of automatically synthesized implementations of specifications of concurrent systems (Deng 2002).

A third set of automated tools support model-driven development, the automatic or semiautomatic transformation of models of a system's specification or architecture into lower-level design and source code. One popular approach to MDD is oriented around models expressed in UML.[15] Such tools do not typically produce proofs. Some ongoing research is attempting to raise the level of formalism associated with UML, including the integrated use of verification systems (Andre 2000).

The Cleanroom as a Life Cycle with Integrated Use of Formal Methods

The Cleanroom methodology integrates nonmechanized formal methods into the life cycle. Cleanroom combines formal methods and structured programming with statistical process control (SPC), the spiral life cycle and incremental releases, inspections, and software reliability modeling. It fosters attitudes such as emphasizing defect prevention over defect removal that are associated with high-quality products in nonsoftware 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. Cleanroom uses incremental releases to support SPC. Specifications developed by the Cleanroom process 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 for a specified time

[15]The Object Management Group (OMG) maintains an online repository of information about UML, including tools supporting MDD. See http://www.omg.org.

- The operational profile for each increment, that is, a model of user behavior of the system
- The reliability model that is applied in system testing to demonstrate reliability

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

Testing does play a very important role in Cleanroom. It serves to verify that reliability goals are attained. Given this orientation, testing is organized differently than in traditional methods. Functional methods, not structural testing methods, are employed. The testing process is deliberately designed to meet the assumptions of the chosen software reliability model. Test cases are statistically generated from the operational profile. Although faults are removed when detected, the testing group's responsibility is reliability measurement and certification, not product improvement.

When testing fails to demonstrate that 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 SPC philosophy tailored for software. Formal methods are embodied in an institutional structure designed to foster a "right the first time" approach. Cleanroom has generated interest and experimentation (Selby 1987, for example) in organizations unaffiliated with Harlan Mills and International Business Machines, the original developers.

CONCLUSIONS

This paper 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 come with costs associated with training and use.

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 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 an extent, when communicating with other programmers. If management provides the appropriate milieu, this group can be expected to foster high-quality attitudes with consequent increases in both productivity and quality.

Technologies that are increasingly widespread today draw on formal methods. Knowledge of formal methods is needed to completely understand these popular technologies and to use them most effectively. Such technologies include:

- Rapid prototyping
- Object-oriented design
- 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. Often, the languages used in prototyping tools involve the same set-theoretical and logical concepts used in formal specification methods more broadly. Abstract data types provide a powerful basis for many classes in object-oriented systems. Furthermore, at least one object-oriented language, Eiffel (Meyer 1988) has assertions, preconditions, postconditions, and invariants built into the language. Thus, formal methods can be usefully combined with object-oriented techniques. Structured

programming is a set of heuristics for producing high-quality code. The historical source for these heuristics lies in formal methods. Adopting formal methods is a natural progression for software development teams currently using structured programming. Inspections throughout the life cycle, following a rigorous methodology (Fagan 1976), have been shown to increase both productivity and quality. Training in formal methods provides inspection-team members with a powerful language to effectively communicate their trains of reasoning.

Further revolutionary advances in software development are likely to draw on formal methods. Many researchers around the world are developing a variety of automated aids for software development and maintenance. Many of these researchers incorporate some level of formal reasoning in their tools. As these tools see more widespread usage, formal methods will be transitioned into wider use.

The full-scale use, transition, and cost-effective use of formal methods is not fully understood. An organization whose leaders can figure out how to effectively integrate formal methods into their software processes will be likely to produce higher-quality software and thereby gain a competitive advantage.

REFERENCES

(Agresti 1986) W. W. Agresti, *New Paradigms for Software Development,* IEEE Computer Society Press.

(Andre 2000) P. Andre, A. Romanczuk, J.-C. Royer, and A. Vasconcelos, "Checking the Consistency of UML Class Diagrams Using Larch Prover," in *Rigorous Object-Oriented Methods 2000,* York, UK, 17 January.

(Avizienis 1990) A. Avizienis and C. S. Wu, "A Comparative Assessment of Formal Specification Techniques," in *Proceedings of the 5th Annual Knowledge-Based Software Assistant Conference.*

(Boehm 1981) B. W. Boehm, *Software Engineering Economics,* Prentice-Hall, Englewood Cliffs, NJ, 1981.

(Burdy 2003) L. Burdy et al., "An Overview of JML Tools and Applications," in *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS'03),* June.

(Chan 1998) W. Chan et al., "Model Checking Large Software Specifications," *IEEE Transactions of Software Engineering, 24,* 7, 498–520, July.

(Chandra 2002) S. Chandra, P. Godefroid, and C. Palm, "Software Model Checking in Practice: An Industrial Case Study," in *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002),* pp. 431–441, 19–25 May.

(Clarke 1996) E. M. Clarke, J. M. Wing, et al., "Formal Methods: State of the Art and Future Directions," *ACM Computing Surveys, 28,* 4, December.

(Cohn 1989) A. Cohn, "The Notion of Proof in Hardware Verification," *Journal of Automated Reasoning, 5,* 127–139.

(Curtis 1988) B. Curtis, H. Krasner, and N. Iscoe, "A Field Study of the Software Design Process for Large Systems," *Communications of ACM, 31,* 11, November.

(DeMillo 1979) R. DeMillo, R. Lipton, and A. Perlis, "Social Processes and Proofs of Theorems and Programs," *Communications of ACM, 22,* 5, May.

(Deng 2002) X. Deng, M. B. Dwyer, J. Hatcliff, and M. Mizuno, "Invariant-based Specification, Synthesis, and Verification of Synchronization in Concurrent Programs," in *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002),* pp. 442–452, 19-25 May.

(Dijkstra 1976) E. W. Dijkstra, *A Discipline of Programming,* Prentice-Hall, Englewood Cliffs, NJ.

(Dijkstra 1989) E. W. Dijkstra, "On the Cruelty of Really Teaching Computer Science," *Communications of ACM, 22,* 5, May.

(Dyer 1992) M. Dyer, *The Cleanroom Approach to Quality Software Development,* Wiley, New York.

(El-Far 2001) I. K. El-Far and J. A. Whittaker, "Model-Based Software Testing," in *Encyclopedia of Software Engineering* (edited by J. J. Marciniak), Wiley.

(Fagan 1976) M. E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal, 15,* 3.

(Fetzer 1988) J. H. Fetzer, "Program Verification: The Very Idea," *Communications of ACM, 31,* 9, September.

(Garland 1991) S. J. Garland and J. V. Guttag, "A Guide to LP, The Larch Prover," Systems Research Center, SRC-082.

(Glass 2002) R. L. Glass, "Proof of Correctness Wars," *Communications of ACM, 45,* 8, August.

(Goel 1991) A. L. Goel and S. N. Sahoo, "Formal Specifications and Reliability: An Experimental Study," in *Proceedings of the International Symposium on Software Reliability Engineering,* IEEE Computer Society Press, 1991.

(Gries 1981) D. Gries, *The Science of Programming,* Springer-Verlag, New York.

(Gries 1991) D. Gries, "On Teaching and Calculation," *Communications of ACM, 34,* 3, March.

(Guttag 1977) J. Guttag, "Abstract Data Types and the Development of Data Structures," *Communications of ACM, 20,* 6, June.

(Hall 1990) A. Hall, "Seven Myths of Formal Methods," *IEEE Software, 7,* 5, 11–19, September.

(IEC 61508) International Electrotechnical Commission, *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems,* 1997.

(IEEE 754) Institute of Electrical and Electronics Engineers, *Standard for Binary Floating-Point Arithmetic,* 1985.

(ISO 15408) International Standards Organization, *Common Criteria for IT Security Evaluations* 1999.

(Kaufmann 1997) M. Kaufmann and J. S. Moore, "An Industrial Strength Theorem Prover for a Logic Based on Common Lisp," *IEEE Transactions on Software Engineering, 203,* 4, 203–213, April.

(Kline 1980) M. Kline, *Mathematics: The Loss of Certainty,* Oxford University Press, 1980.

(Lamport 1989) L. Lamport, "A Simple Approach to Specifying Concurrent Systems," *Communications of ACM, 32,* 1, January.

(Leveson 1990) N. G. Leveson, "Guest Editor's Introduction: Formal Methods in Software Engineering," *IEEE Transactions on Software Engineering, 16,* 9, 929–931, September.

(Linger 1979) R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming: Theory and Practice,* Addison-Wesley, Reading, MA.

(Liu 1999) X. Liu et al., "Building Reliable, High Performance Communication Systems from Components," *Operating Systems Review, 34,* 5, 80–92, December.

(London 1977) R. L. London, "Remarks on the Impact of Program Verification on Language Design," in *Design and Implementation of Programming Languages,* Springer-Verlag, New York.

(McGettrick 1982) A. D. McGettrick, *Program Verification Using Ada,* Cambridge University Press.

(MacKenzie 2001) D. A. MacKenzie, *Mechanizing Proof: Computing, Risk, and Trust,* MIT Press.

(MacKenzie 1998) D. A. MacKenzie, "Computers and Sociology of Mathematical Proof," *Northern Formal Methods Workshop,* Ilkley, September 1998.

(Mellor 2003) S. J. Mellor, A. N. Clark, and T. Futagami, "Model-Driven Development: Guest Editors' Introduction," *IEEE Software, 20,* 5, 14–18, September/October.

(Merrill 1983) G. Merrill, "Proofs, Program Correctness, and Software Engineering," *SIGPLAN Notices, 18,* 12, December.

(Meyer 1985) B. Meyer, "On Formalism in Specifications," *IEEE Software, 2,* 1, 6–26, January.

(Meyer 1988) B. Meyer, *Object-Oriented Software Construction,* Prentice-Hall, Englewood Cliffs, NJ.

(Mills 1986) H. D. Mills, "Structured Programming: Retrospect and Prospect," *IEEE Software, 3,* 6, 58–66, November.

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

(Paulk 1995) M. C. Paulk et al., *The Capability Maturity Model: Guidelines for Improving the Software Process,* Addison-Wesley, 1995.

(Peterson 1977) J. L. Peterson, "Petri Nets," *Computing Surveys, 9,* 3, September.

(Place 1990) P. R. H. Place, W. Wood, and M. Tudball, *Survey of Formal Specification Techniques for Reactive Systems,* Software Engineering Institute, CMU/SEI-90-TR-5, May.

(Preston 1988) D. Preston, K. Nyberg, and R. Mathis, "An Investigation into the Compatibility of Ada and Formal Verification Technology," in *Proceedings of the 6th National Conference on Ada Technology.*

(Selby 1987) R. W. Selby, V. R. Basili, and F. T. Baker, "Cleanroom Software Development: An Empirical Evaluation," *IEEE Transactions on Software Engineering, SE-13,* 9, 1027–1037, September.

(Stepney 1998) S. Stepney, "A Tale of Two Proofs," in *BCS-FACS Third Northern Formal Methods Workshop,* available at http://ewic.bcs.org, Ilkley, UK, September.

(Vienneau 1991) R. L. Vienneau, *An Overview of Object Oriented Design,* Data & Analysis Center for Software.

(Vienneau 2003) R. L. Vienneau and R. Senn, "A State of the Art Report: Software Design Methods," *The ICFAI Journal of Systems Management,* August.

(Wang 2004) F. Wang, K. Schmidt, F. Yu, G.-D. Huang, and B.-Y. Wang, "BDD-Based Safety-Analysis of Concurrent Software with Pointer Data Structures Using Graph Automorphism Symmetry Reduction," *IEEE Transactions on Software Engineering, 30,* 6, 403–417, June.

(Wing 1990) J. M. Wing, "A Specifier's Introduction to Formal Methods," *Computer, 23,* 9, 8–24, September.

(Zeroual 1991) K. Zeroual, "KBRAS: A Knowledge-Based Requirements Acquisition System," in *Proceeding of 6th Annual Knowledge-Based Software Engineering Conference,* IEEE Computer Society Press, 1991.