What is a Formal Method?

Jeannette M. Wing

10 November 1989
CMU-CS-89-200 Z

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

A formal method is a mathematically-based technique used in Computer Science to describe properties of hardware and/or software systems. It provides a framework within which large, complex systems may be specified, developed, and verified in a systematic rather than ad hoc manner. A method is formal if it has a sound mathematical basis, typically given by a formal specification language. A formal method is only a method, rather than an isolated mathematical entity in itself, because of a number of pragmatic considerations: who uses it, what it is used for, when it is used, and how it is used. This paper elaborates on what makes up a formal method and compares six different well-known formal methods, three used to specify abstract data types and three used to specify properties of concurrent and distributed systems.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory Under Contract No. F33615-87-C-1499. Additional support was provided in part by the National Science Foundation under grant CCR-8620027.

What is a Formal Method?

Jeannette M. Wing
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

November 10, 1989

1. Introduction

A formal method is a mathematically-based technique used in Computer Science to describe properties of hardware and/or software systems. A formal method provides a framework within which large, complex systems may be specified, developed, and verified in a systematic rather than ad hoc manner.

A method is formal if it has a sound mathematical basis, typically given by a *formal specification language*. This basis provides the means of defining precisely notions like consistency and completeness, and more relevantly, specification, implementation, and correctness. It provides the means of proving that a specification is realizable, proving that a system has been implemented correctly, and proving properties of a system without necessarily running it to determine its behavior.

A formal method is only a method, rather than an isolated mathematical entity in itself, because of a number of pragmatic considerations: who uses it, what it is used for, when it is used, and how it is used. Most commonly, system designers use formal methods to specify a system's desired behavioral and structural properties. However, anyone involved in any stage of system development can make use of formal methods. They can be used in the initial statement of a client's requirements, through system design, implementation, testing, debugging, maintenance, verification and evaluation. Formal methods are used to reveal ambiguities, incompleteness, and inconsistencies in a system. If used early in the system development process, they can reveal design flaws that otherwise would only possibly be discovered in the costly testing and debugging phases. If used later, they can help determine the correctness of a system implementation and the equivalence of different implementations.

For a formal method to be formal, it must have a well-defined mathematical basis; it need not address any pragmatic considerations, but one that does not would be a pretty useless method. Hence, a formal method should come with a set of guidelines, or "style sheet," that tells users under what circumstances the method can and should be applied and how most effectively to apply it.

One tangible product of the application of a formal method is a (formal) specification. A specification serves as a "contract" between the client and the implementor. A specification serves as a valuable piece of documentation and as a means of communication between clients, specifiers, designers, and implementors. Because of the mathematical basis of a formal method, formal specifications are more precise, and usually more concise, than informal ones.

Since a formal method is a method, not just a computer program or language, it may or may not have tool support. If the syntax of a formal method's specification language is made explicit then it would be straightforward to provide standard syntax analysis tools for formal specifications. If the language's semantics are sufficiently restricted, varying degrees of semantic analysis can be performed with machine aids as well. Thus, formal specifications have the additional advantage over informal ones of being amenable to machine analysis and manipulation.

We begin by first defining those elements necessary to give a formal basis to a formal method and then address pragmatic concerns. We then draw examples using a few of the more well-known or commonly-used methods. See the accompanying survey article in this issue for a more comprehensive listing of example methods, plus citations.

2. What is a Specification Language?

A formal specification language provides a formal method's mathematical basis. We borrow the following terms and definitions from Guttag, Horning and Wing [33].

A *formal specification language* consists of two sets, Syn and Sem, and a relation, Sat, between them. The first set is called its *syntactic domain*; the second, its *semantic domain*; the relation is called *satisfies*. For a given specification language, if Sat(syn, sem), then syn is a *specification* of sem, and sem is a *specificand* of syn. The *specificand set* of a specification is the set of all its specificands.

Somewhat less formally, a formal specification language provides a notation (its syntactic domain), a universe of objects (its semantic domain), and a precise rule defining which objects satisfy each specification. A specification is a sentence written in terms of the elements of the syntactic domain. It denotes a specificand set, a subset of the semantic domain. A specificand is an object satisfying a specification. The satisfies relation provides the *meaning*, or interpretation, for the syntactic elements.

BNF is an example of a simple formal specification language, with a set of grammars as its syntactic domain and a set of strings as its semantic domain. Every string is a specificand of each grammar that generates it. Every specificand set is a formal language.

In principle, a formal method is based on some well-defined formal specification language; in practice, however, this language may not have been explicitly given. One could say that the more explicit the specification language's definition, the more "well-defined" the formal method.

Formal methods differ because their specification languages have different syntactic and/or semantic domains. Even if they have identical syntactic and semantic domains, they may have different satisfies relations.

2.1. Syntactic Domains

A specification language's syntactic domain is usually defined in terms of a set of symbols, e.g., constants, variables, and logical connectives, and a set of grammatical rules for concatenating these symbols into *well-formed* sentences. For example, using standard notation for universal quantification ($\forall$) and logical implication ($\Rightarrow$), let x be a logical variable and P and Q be predicate symbols. Then this sentence, $\forall x.P(x) \Rightarrow Q(x)$, would be well-formed in predicate logic, but not this one, $\forall x. \Rightarrow P(x) \Rightarrow Q(x)$, because $\Rightarrow$ is a binary logical connective.

A syntactic domain need not be restricted to text; graphical elements such as boxes, circles, lines, arrows, and icons can be given a formal semantics just as precisely as textual ones. A well-formedness condition on such a *visual* specification might be that all arrows start and stop at boxes.

2.2. Semantic Domains

Specification languages differ most in their choice of semantic domain. Some examples:

- "Abstract data type specification languages," have been used to specify algebras [25,20], theories [11,61], and programs [31]. Though specifications written in these different specification languages range over different semantic domains, they often look syntactically similar.

- "Concurrent and distributed systems specification languages," have been used to specify state sequences [12,56], event sequences [37,56], state and transition sequences [21], streams [9], synchronization trees [58], partial orders [68], and state-machines [55,48].
- Programming languages are used to specify functions from input to output [70], computations, predicate transformers [18], relations [15], and machine instructions.

Note that each programming language is a specification language, but not vice versa because specifications in general do not have to be executable on some machine whereas programs do. The advantage gained in using a more "abstract" specification language is that one is not restricted to expressing only computable functions. It is perfectly reasonable in a specification to express notions like "For all x in set A, there exists a y in set B such that property P holds of x and y" where A and B might be infinite sets.

When a specification language's semantic domain is over programs or systems of programs, the term *implements* is used for the satisfies relation, and the term *implementation* is used for a specificand in Sem. One says that an implementation *prog* is *correct* with respect to a given specification *spec* if *prog* satisfies *spec*.

2.3. Satisfies Relation

One often would like to specify different aspects of a single specificand, perhaps using different specification languages. For example, one might want to specify the functional behavior of a collection of program modules as the composition of the functional behaviors of the individual modules. One might also want to specify a structural relationship between the modules such as what set of modules each module directly invokes.

In order to accommodate these different possible "views" of a specificand, one can associate with each specification language a homomorphism, A, called a *semantic abstraction function*, that maps elements of the semantic domain into equivalence classes. One can choose the semantic abstraction function so that there exists an induced relation ASat, called an *abstract satisfies relation*, such that

$$\forall spec \in Syn, prog \in Sem[Sat(spec, prog) = ASat(spec, A(prog))]$$

Different semantic abstraction functions make it possible to describe multiple views of the same equivalence class of systems, or similarly, impose different kinds of constraints on these systems. It can be useful to have several specification languages with different semantic abstraction functions for a single semantic domain. This encourages and supports complementary specifications of different aspects of a system.

For example, in Figure 1 there is a single semantic domain, Sem, on the right. One semantic abstraction function partitions specificands in Sem into a set of equivalence classes, three of which are drawn as blobs in solid lines. Another partitions specificands into a different set of equivalence classes, two of which are drawn as blobs in dashed lines. Via the abstract satisfies relation ASat1, specification A of syntactic domain Syn1 maps to one equivalence class of specificands (defined by a solid-lined blob), and via ASat2, specification B of syntactic domain Syn2 maps to a different equivalence class of specificands (defined by a dashed-line blob). Note the overlap between the solid-lined and dashed-lined blobs. To be concrete, suppose Sem is a library of Ada program modules. Imagine that A specifies (perhaps through a predicate in first-order logic) all procedures that sort arrays, and B specifies (perhaps through a call graph) all procedures that call functions on a user-defined enumeration type *E*. Then a procedure that sorted arrays of *E*'s might be in the intersection of ASat1(A) and ASat2(B).

Two broad classes of semantic abstraction functions are those that abstract preserving each system's *behavior* and those that abstract preserving each system's *structure*. In the example above, A specifies a behavioral aspect of the Ada program modules, but B describes a structural aspect.

Behavioral specifications describe constraints only on the observable behavior of specificands. A system's required functionality, i.e., mapping from inputs to outputs, is the behavioral constraint addressed by most formal

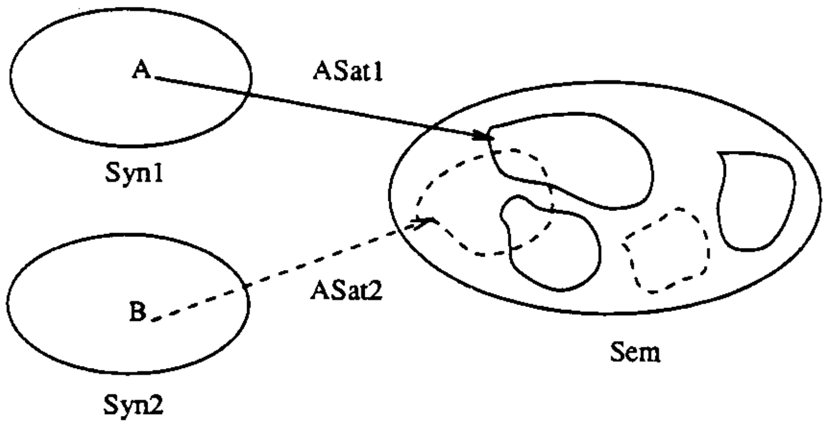

Figure 1: Abstract Satisfies Relations

methods. Current research in formal methods addresses other behavioral aspects such as fault-tolerance, safety, security, response time and space efficiency. Often some of these behavioral aspects, such as security, are included as part of, rather than separate from, a system's functionality. If the overall correctness of a system is defined so that it must satisfy more than one behavioral constraint, then a system that satisfies one but not another would be incorrect. For example, if functionality and response time were the constraints of interest, a system producing correct answers past deadlines would be just as unacceptable as a system producing incorrect answers in time.

Structural specifications describe constraints on the internal composition of specificands. Module interconnection languages [17] are examples of structural specification languages that capture "uses" relations such as those that procedure call and data flow graphs represent. Systems that satisfy the same structural constraints do not necessarily satisfy the same behavioral constraints. Moreover, the structure of a specification need not bear any direct relationship to the structure of its specificands.

For the most part, we will confine our discussion in this paper to behavioral specifications of specificands that are programs or systems of programs.

2.4. Properties of Specifications

Each specification language should be defined so that each well-formed specification written in the language is unambiguous. A specification is *unambiguous* if and only if it has exactly one meaning. This key property of formal specifications means that any specification language based on or that incorporates a natural language (like English) is not formal since natural languages are inherently ambiguous. It also means that a visual specification language that permits for multiple interpretations of a box and/or arrow is ill-defined, and hence not formal.

Another desirable property of specifications is consistency. A specification is *consistent* (or *satisfiable*) if and only if its specificand set is non-empty. In terms of programs, consistency is important because it means that there exists some implementation that will satisfy the specification. If one views a specification as a set of facts, consistency implies that one cannot derive anything contradictory from the specification. If one were to pose a question based on a consistent specification, one will not get mutually exclusive answers. It is obvious that one wants to have consistent specifications, since an inconsistent specification means one has no knowledge at all, as the specification negates on one occasion what it asserts on another.

Specifications need not be "complete" in the sense used in mathematical logic, though certain "relative-completeness" properties might be desirable (e.g., *sufficient-completeness* of an algebraic specification [30]). In practice, one will not write or deal with "complete" specifications. Why not? One may intentionally leave some things unspecified, giving the implementor some freedom to choose among different data structures and algorithms. Also, one cannot realistically anticipate all possible scenarios in which a system will be run, and thus, perhaps unwittingly have left some things unspecified. Finally, one typically develops specifications gradually, and hence works more often with unfinished products, rather than finished ones.

There is a delicate balance between saying just enough and saying too much in a specification. One wants to say enough so that implementors do not choose unacceptable implementations. Specifiers are responsible for not making oversights; any incompleteness in the specification should be an intentional incompleteness. On the other hand, saying too much may leave little design freedom for the implementor. A specification that overspecifies will be accused of "implementation bias" [41].

2.5. Proving Properties of Specificands

Most formal methods are defined in terms of a specification language that has a well-defined *logical inference system*. A logical inference system defines a *consequence relation*, typically given in terms of a set of *inference rules*, that maps a set of well-formed sentences in the specification language to a set of well-formed sentences.

One uses this inference system to prove properties about specificands through the specification. Again taking a specification as a set of facts, one derives new facts through the application of the inference rules. When one proves a statement inferrable from these facts, then one proves a property that a specificand satisfying the specification will have–a property not explicitly stated in the specification. An inference system gives users of formal methods a way to "predict" the behavior of a system without having to execute or even build it. It gives users a way to state questions, in the form of theorems, about a system cast in terms of just the specification itself. Users can then answer these questions in terms of a formal proof constructed through a formal derivation process. The inference system gives users a way to increase their confidence in the validity of the specification itself since if one were able to prove a surprising result from the specification, then perhaps the specification is wrong.

Formal methods with explicitly defined inference systems usually have the further advantage that these systems can be completely mechanized, e.g., if they have a finite set of finite rules. Theorem provers and proof checkers are example tools that assist users with the tedium of deriving and managing formal proofs.

3. Pragmatics

3.1. Uses

Formal methods can be applied in all phases of system development. Such application ought not to be considered as a separate activity, but rather as an integral one. The greatest benefit gained in applying a formal method is often in the process of specifying rather than the end result. Gaining a deeper understanding of the specificand by forcing oneself to be abstract, yet precise, about the desired properties of a system can be more rewarding than in having a one hundred-page specification document.

Let us consider more specifically uses of formal methods in the different phases of system development:

Requirements analysis. Applying a formal method helps to clarify a client's set of informally stated requirements. A specification helps to crystallize the client's vague ideas and helps to reveal contradictions, ambiguities, and incompleteness in the requirements. A specifier has a better chance of asking pertinent questions and evaluating the client's responses through the use of a formal specification rather than through an informal one. Both the client and specifier can pose and answer questions based on the specification to see whether it reflects the client's intuition and whether the specificand set has the desired set of properties.

System design. During design, one decomposes a system into smaller modules. Formal methods can aid in the process of design by capturing precisely the interfaces between these modules. Each interface specification provides the module's user all the information needed to use the module without knowledge of its implementation. The interface specification simultaneously provides the module's implementor all the information needed to implement the module without knowledge of its users. Thus, as long as the interface remains the same, the implementation of the module can be replaced, perhaps by a more efficient one, at some later time without affecting its users. The interface provides the place for the designer to record design decisions; moreover, any intentional incompleteness can be succinctly captured as a parameter in the interface.

System refinement. During refinement, one works at different levels of abstraction, perhaps refining a single module at one level to be a collection of modules at a lower level or choosing for an abstract data type its representation type. This process generates proof obligations for showing that the refinement "satisfies" the higher-level specification. A formal method provides a way to state and discharge these obligations. Formal methods, such as Refine [26], that incorporate program transformation, derivation, and verification techniques are applicable during system refinement.

System validation. Formal methods can be used to help in testing and debugging systems. Specifications alone can be used to generate test cases for black-box testing. Specifications that explicitly state assumptions on the module's use clearly identify test cases for boundary conditions. Specifications along with implementations can be used to do other kinds of testing analysis such as path testing, unit testing, and integration testing. Testing based solely on an analysis of the implementation is not sufficient; one must take the specification into account. For example, a test set may be complete for doing a path analysis of an implementation, but may not reveal missing paths that the specification would otherwise suggest. The success of unit and integration testing depends on the precision of the specifications of the individual modules. Testing tools that take formal specifications as input include test case generators and symbolic execution tools [57,46].

System verification. Formal verification is impossible without a formal specification. Though one may never completely verify an entire system, one can certainly verify smaller critical pieces of a system. The trickiest part is in stating explicitly the assumptions about the environment in which each critical piece is placed. (Section 5 elaborates on this point.)

System documentation. A specification serves as an alternative description of the system. It serves as a communication medium–between a client and a specifier, between a specifier and an implementor, and among members of an implementation team. Nothing is more exasperating to hear in reply to the question "What does it do?" than the answer "Run it and see." One of the primary intended uses of formal methods is to capture in a formal specification the *what* rather than the *how*. One can then read the specification, rather than read the implementation or worse, execute the system, to find out the system's behavior.

System analysis and evaluation. In order to learn from the experience of building a system, one should do a critical analysis of its functionality and performance once it has been built and tested. Does the system do what the client wants? Does it do it fast enough? If formal methods were used in its development then they can help system evaluators formulate and answer these questions. The specification serves as a reference point. In the case that the client is unhappy, but the system meets the specification, then the specification can be changed and the system changed accordingly.

Indeed much recent work in the application of formal methods to non-trivial examples has been in specifying a system already built, running, and used, rather than in specifying one yet to be built. Some of these exercises revealed bugs in published algorithms [10] and circuit designs [8]–serious bugs that had gone undiscovered for years. Most revealed, as expected, unstated assumptions, inconsistencies, and unintentional incompleteness in the system.

Example medium-sized systems that have been specified formally include hardware [23,14], microprocessors

[39,16], operating systems kernels [5], and secure systems [59]. Most formal methods have not yet been applied to specifying large-scaled software and/or hardware systems; most are still inappropriate for specifying a wide range of behavioral constraints beyond functionality, e.g., fault-tolerance and real-time.

This problem of scale exists in two different, often confused, dimensions: size of the specification and complexity of the specificands. Tools can help address specification size since managing large specifications is just like managing other large documents. The problem of dealing with a specificand's inherent complexity remains, since no magic will make it disappear. However, by providing a systematic way of thinking and reasoning about specificands, formal methods are precisely what can help humans grapple with such complexity.

3.2. Users

The audience of a specification includes two not necessarily disjoint classes of users: writers and readers. *Specifiers* write, evaluate, analyze, and refine specifications. They prove that their refinements preserve certain properties and prove properties of specificands through specifications.

Most people need only read specifications, not develop their own from scratch. There are many classes of readers. Besides specifiers, there are *clients*, those people who may have hired the specifiers; *implementors*, those people who realize a specification; *verifiers*, those people who prove (by hand or with machine assistance) the correctness of implementations; and finally, *machine tools*, some of which might blindly manipulate specifications without regard to their meaning.

One point of tension in many formal methods is that the language of the method may be accessible to one class of users and not another. Most languages will try to be accessible to at least two, e.g., clients and specifiers, or specifiers and implementors. Some specification languages have a lot of syntactic sugar to make the specifications more readable by clients. Some have a minimal amount because the intent of the method is to do formal proofs by machines or because the meaning of a rich set of cryptic mathematical notation is assumed.

It is important that users are told the domain of applicability of a given method. For example, a formal method might be applicable for describing sequential programs, but not parallel ones; or, for describing message-passing distributed systems, but not transaction-based distributed databases. Without knowing the proper domain of applicability, a user may unknowingly inappropriately apply a formal method to an inapplicable domain.

Part of a formal method's pragmatic concerns is to identify what different classes of users the method is targeted for and what the abilities of each should be. Some methods expect that the users know modern algebra, set theory, and/or predicate logic in order to be applied correctly. Domain-specific methods may require knowledge of additional mathematical theories such as digital logic, e.g., if specifying hardware, or probability and statistics, e.g., if specifying system reliability.

3.3. Style

The aspect that inherently makes a formal method a method is the matter of style or how one uses a method. It is not the subject of this paper to characterize completely all different styles of specification nor to classify exhaustively all methods according to these styles. Instead, we give a partial listing of different styles, where one method can justifiably be classified under more than one style. In the next section, we give examples of methods that illustrate some of these styles.

3.3.1. Model- Versus Property-Oriented

Two broad classes of formal methods are the so-called *model-oriented* and *property-oriented*. In the model-oriented style, one defines a system's behavior directly by constructing a model of the system in terms of mathematical structures such as tuples, relations, functions, sets, and sequences. In the property-oriented style, one defines the system's behavior indirectly by stating a set of properties, usually in the form of a set of axioms, that the system must satisfy. What often matters in a property-oriented specification is not what one explicitly says, but what one does not say, thereby permitting for a larger number of possible implementations.

Example methods supporting a model-oriented style used in the domain of sequential programs and abstract data types include Parnas's state-machines, [65], Robinson and Roubine's extensions to them with V-, O-, and OV-functions [69], VDM [42] and Z [71]. Example methods used in the domain of concurrent and distributed systems include Petri Nets [66], Milner's Calculus of Communicating Systems (CCS) [58], Hoare's Communicating Sequential Processes (CSP) [37], Unity [12], I/O automata [55], and TSL [53]. The Raise Project represents more recent work on combining VDM and CSP [62].

The property-oriented style can be further broken into two categories, sometimes referred to as *axiomatic* and *algebraic*. The axiomatic style stems from Hoare's work on proofs of correctness of implementations of abstract data types [38], where first-order predicate logic pre- and post-conditions are used for the specification of each operation of the type. Iota [61] and Anna [54] are example specification languages that support an axiomatic style. In the algebraic style, objects such as data types and processes are defined to be heterogeneous algebras [6]. This approach uses axioms to specify properties of systems, but the axioms are restricted to equations. Much work has been done on the algebraic specification of abstract data types [25,30,76,11,19,72,43] including the handling of error values, nondeterminism, and parameterization. The more widely-known specification languages that have resulted from this work are CLEAR [11], OBJ [24], ACT ONE [20], and Larch [31].

Examples of property-oriented methods for specifying concurrent and distributed systems include extensions to Hoare-axioms [63,4], temporal logic [67,56,64], and Lamport's state functions [48]. The LOTOS specification language [1] represents more recent work on the combination of ACT ONE and CCS (with some CSP influence).

3.3.2. Visual Languages

Visual methods include any whose language contains graphical elements in its syntactic domain. The most prominent visual method is Petri Nets [66], and its many variations, used most typically to specify the behavior of concurrent systems.

More recent visual language work includes Harel's statecharts based on higraphs [34], used to specify state transitions in "reactive" systems. Figure 2 gives a simple example of a statechart that describes the behavior of a one-slot buffer. Roundtangles represent states in a state machine and arrows represent state transitions. Initially, the one-slot buffer is empty; in the event that a message arrives and gets put in the buffer, the buffer becomes full; when the message has been serviced and removed from the buffer, its state changes back to being empty. The example shows one of the more notable features of statecharts that distinguish them from "flat" state-transition diagrams: A roundtangle can represent a hierarchy of states (and in general, an arrow can represent a set of state transitions), thereby letting users "zoom-in" and "zoom-out" of a system and its subsystems.

Harel's higraph notation inspired the design of the Miró visual languages, which are used to specify security constraints [36]. Like statecharts, the Miró languages have a formally defined semantics and tool support.

Some methods use visual notation but are not formal. These informal or semi-formal methods allow for the construction of ambiguous specifications, perhaps because English text is attached to the graphical elements or because multiple interpretations of a graphical element (usually different meanings for an arrow) are possible. Many popular software and system design methods such as Jackson's method [40], HIPO [45], Structured Design [74] and SREM [3] are examples of semi-formal methods that use pictures.

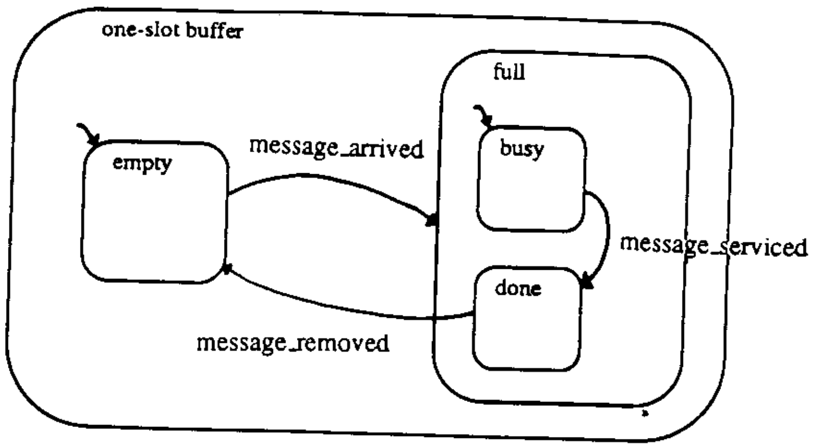

Figure 2: Statechart Specification of a One-Slot Buffer

3.3.3. Executable

Another class of methods falls under the term *executable*. Specifications written using these methods can be run on a computer and thus, by definition are more restricted in expressive power than a non-executable specification. As long as users realize that the specification may suffer from implementation bias, executable specifications can play an important role in the system development process. One can use them to gain immediate feedback about the specification itself, to do rapid prototyping (the specification serves as a prototype of the system), and to test a specificand through symbolic execution of the specification. For example, Statemate [35] is a tool that lets users run simulations through the state transition diagram represented by a statechart.

Besides statecharts, executable specification languages include OBJ [24], an algebraic specification language; Prolog [15], a logic programming language, which supports a property-oriented specification style where properties are stated as logical relations on objects; and PAISley [75], a model-oriented specification language, based on a model of event sequences and used to specify functional and timing behavioral constraints for asynchronous parallel processes.

3.3.4. Tool-Based

Some formal methods evolved from the semantic-analysis tools that were built to manipulate specifications and programs. Model checking tools let users construct a finite-state model of the system and then show a property holds of each state and/or state transition of the system. These tools such as EMC [8,13] are especially useful for specifying and verifying properties of hardware circuits.

Proof checking tools that let users treat algebraic specifications as rewrite rules include Affirm [60], Reve [49], the Rewrite Rule Laboratory (RRL) [44], and the Larch Prover [22]. Tools (and their associated specification language) that handle subsets of first-order logic include the Boyer-Moore Theorem Prover (and the Gypsy specification language) [7,27], FDM (and Ina Jo) [51], and HDM (and SPECIAL), [50]. Finally, tools that handle subsets of higher-order logics include HOL [28] and LCF [29].

4. Some Examples

We present and compare six different formal methods, three applied to one simple example, and three applied to another. All six methods have been used to specify much more complex systems. First, we will specify a symbol table data type using Z, VDM, and Larch, which are formal methods especially appropriate for specifying sequential programs modules like procedures, functions, classes, and packages. Then we will specify the behavior of an unbounded buffer process using temporal logic, CSP, and state functions, which are formal methods especially appropriate for specifying properties of concurrent and distributed systems.

4.1. Abstract Data Types: Z, VDM, Larch

Z is formal method based on set theory. Though Z can be used in both model-oriented and property-oriented styles, Figure 3 gives a model-oriented specification of a symbol table. The state of the table is "modeled" by a partial mapping from keys to values. The initial state of the mapping, and hence the symbol table, is empty. By convention, unprimed variables in Z stand for the state before an operation is performed and primed variables for the state afterwards; we will use the same convention in the VDM and Larch specifications. There are three operations on the table, UPDATE, LOOKUP, and DELETE. UPDATE modifies the table by either adding a new binding to the mapping st, or, if the key is already in the mapping st, overriding the value assigned to that key ($\oplus$ is an overriding operator). LOOKUP requires that the key being looked up be in the domain of the mapping, returns the value to which a given key is mapped, and does not change the state of the symbol table (st' = st). DELETE also requires that the key being looked up be in the domain of the mapping, and modifies the table by deleting the binding associated with the key k from st ($\lhd$ is a domain subtraction operator). The machine tool B [2] has been used for proving theorems based on Z specifications.

The Vienna Definition Method (VDM) supports a model-oriented specification style. VDM defines a set of built-in data types, e.g., sets, lists, and mappings, from which specifiers define other types. The VDM specification in Figure 4 defines a symbol table also in terms of a mapping from keys to values. The INIT function initializes the symbol table to be empty. The INSERT function, which adds a new binding to the mapping, is different from the UPDATE function in the Z specification since the pre-condition requires that the key to be inserted not already be in the symbol table; only new bindings are inserted and old ones are never overridden. LOOKUP's and DELETE's pre-conditions are the same as in the Z specification, but made more explicit in the VDM specification. Their effects, as stated in the post-conditions are also similar. Different notation is used for the mapping difference operator ($\backslash$). The fact that LOOKUP does not modify the symbol table (hence $st' = st$), but INSERT and DELETE do, is annotated by using **rd** (for "read only" access) instead of **wr** (for "write and read" access) in the declaration of the external state variables accessed by each operation.

The Larch Family of Specification Languages uses a property-oriented style, combining both axiomatic and algebraic styles into a "two-tiered" specification [31]. The axiomatic component specifies state-dependent behavior, e.g., side effects, of programs; the algebraic component specifies state-independent properties of data accessed by programs. Figure 5 gives a Larch specification of the symbol table example.

The first piece of the Larch specification, called an *interface* specification, looks similar to the Z and VDM specifications. For each operation, the **requires** and **ensures** clauses specify the pre-and post-conditions for the symbol table's operations. Instead of placing a pre-condition on the insert and lookup operations as was done in the Z and VDM specifications, a **signals** clause is added. Hence, insert may terminate by either inserting a new key-value binding or by signaling to the invoker that the key to be inserted is already in the symbol table. Similar remarks hold for the lookup operation. It is assumed that the target programming language supports signals; if not, then the specification could be modified to have explicit pre-conditions as in the delete operation. The **modifies** clause lists those objects whose value may possibly change as a result of executing the operation. Hence, lookup is not allowed to change the state of its symbol table argument, but insert and delete are.

The second piece of the Larch specification, called a *trait*, looks like an algebraic specification. It contains a set of function symbol declarations and a set of equations that define the meaning of the function symbols. The

$ST \mathrel{\widehat{=}} KEY \pfun VAL$

$st_{INIT} \mathrel{\widehat{=}} \{\}$

UPDATE

$st, st' : KEY \pfun VAL$
$k : KEY$
$v : VAL$

$st' = st \oplus \{k \mapsto v\}$

LOOKUP

$st, st' : KEY \pfun VAL$
$k : KEY$
$v : VAL$

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

DELETE

$st, st' : KEY \pfun VAL$
$k : KEY$

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

Figure 3: Z Specification of a Symbol Table

```
State:: ST :Mkv
    Mkv = map Key to Val

INIT()
ext    ST :wr Mkv
post   $st' = []$

INSERT(K: Key, V: Val)
ext    ST :wr Mkv
pre    $k \notin \mathbf{dom}\ st$
post   $st' = st \cup [k \mapsto v]$

LOOKUP(K: Key) V: Val
ext    ST :rd Mkv
pre    $k \in \mathbf{dom}\ st$
post   $v' = st(k)$

DELETE(K: Key)
ext    ST :wr Mkv
pre    $k \in \mathbf{dom}\ st$
post   $st' = st \setminus \{k\}$
```

Figure 4: VDM Specification of a Symbol Table

```
symbol_table is data type based on S from SymTab
    init = proc () returns (s: symbol_table)
        ensures s' = emp ∧ new (s)
    insert = proc (s: symbol_table, k: key, v: val) signals (already_in)
        modifies (s)
        ensures if isin(s, k) then signal already_in else s' = add(s, k, v)
    lookup = proc (s: symbol_table, k: key) returns (v: val) signals (not_in)
        ensures if ¬isin(s, k) then signal not_in else v' = find(s, k)
    delete = proc (s: symbol_table, k: key)
        requires isin(s, k)
        modifies (s)
        ensures s' = rem(s, k)
    end symbol_table


SymTab: trait
    introduces
        emp: → S
        add: S, K, V → S
        rem: S, K → S
        find: S, K → V
        isin: S, K → Bool
    asserts
     S generated by (emp, add)
     S partitioned by (isin)
     for all (s: S, k, k1: K, v: V)
        rem(add(s, k, v), k1) == if k = k1 then s else add(rem(s, k1), k, v)
        find(add(s, k, v), k1) == if k = k1 then v else find(s, k1)
        isin(emp, k) == false
        isin(add(s, k, v), k1) == (k = k1) ∨ isin(s, k1)
    implies
        exempting (rem(emp), find(emp))
    end SymTab
```

Figure 5: Larch Specification of a Symbol Table

equations determine an equivalence relation on sorted terms. Objects of the symbol_table data type specified in the interface specification range over values denoted by the terms of sort S. The **generated by** clause states that all symbol table values can be represented by terms composed solely of the two function symbols, emp and add. This clause defines an inductive rule of inference and is useful for proving properties about all symbol table values. The **partitioned by** clause adds more equivalences between terms. Intuitively it states that two terms can be considered equal if they cannot be distinguished by any of the functions listed in the clause. In the example, we could use this property to show that order of insertion of distinct key-value pairs in a symbol table does not matter, i.e., insertion is commutative. The **exempting** clause documents the absence of right-hand sides of equations for rem(emp) and find(emp); the pre-conditions and **signals** clauses in the interface specification deal with these "error values". The **exempting** clause together with the **generated by** clause is used to show that this algebraic specification is sufficiently-complete.

Syntax analyzers exist for Larch traits and interfaces. The Larch Prover has been used to perform semantic analysis on Larch traits.

The user-defined function symbols in a Larch trait are exactly those used in the pre- and post-conditions of the interface specification; they serve the same role as the built-in symbols like $\oplus$ in the Z specification or $\setminus$ in the VDM specification. Unlike Z and VDM, Larch does not come with any special built-in notation nor any built-in types. The advantage is that the user does not have to learn any special vocabulary for those concepts and is free to introduce whatever symbols he or she desires, giving them exactly the meaning suitable for the specificand set. Exactly and only those properties of a data type being specified need to be stated explicitly and satisfied by an implementation. The disadvantage is that the user may often need to provide a large set of user-defined symbols, as well as the equations that define their meaning. Since we modeled symbol tables in Z and VDM in terms of finite mappings, we did not need to state explicitly that insertion is commutative since this is a property of mappings, i.e., this property came "for free." The Larch Handbook [32] serves as a compromise between the two extremes: it provides a library of traits that define many general and commonly-used concepts, e.g., properties of partial orders, sets and sequences.

Larch is property-oriented, not model-oriented, because nothing in a Larch specification specifies a particular model or class of models for the data type being defined. The implementor is free to represent symbol tables as binary trees, hash tables, sets of pairs, or finite mappings, just as long as the implementation satisfies the more abstract properties of symbol tables. Of course, the implementor has the same design freedom if starting from the VDM or Z specification, but he or she must prove the behavioral equivalence between the representation type chosen and the model used in the VDM or Z specification.

4.2. Concurrency: Temporal logic, CSP, State-Functions

When specifying properties of concurrent and distributed systems, properties of interest fall into two general categories, *safety* and *liveness* [48]. Safety properties ("nothing bad ever happens") include functional correctness and liveness properties ("something good eventually happens") include termination.

Temporal logic is a property-oriented style for specifying safety and liveness properties. For a given temporal logic inference system, *modal operators* such as $\Box$ and $\Diamond$ are used to state concisely predicates about state (or event) sequences and allow one to talk about past, current, and future states (or events). Temporal logic notation tends to be terse and a temporal logic specification is simply a unstructured set of predicates all of which must be satisfied by a given implementation. A temporal logic specification of the behavior of an unbounded buffer in an asynchronous environment is given in Figure 6 (adapted from [47] and [67]). A buffer has a left input channel and a right output channel. The expression $\langle c!m \rangle$ denotes the event of placing message m on channel c. The first predicate states that any message transmitted to the right must have been previously placed on the left channel. The second predicate states that messages are transmitted in a first-in-first-out fashion: If a message m currently placed on the output channel is preceded by some other message m' (also on the output channel), then there must have been a preceding event of placing m on the input channel, and moreover, an even earlier event that placed m' on the input channel ahead of m. The third predicate states that all messages are unique. This property is not a property of the buffer, but an assumption on the environment. This assumption is essential to the validity of the

$$\langle right!m\rangle \Rightarrow \Diamond\langle left!m\rangle \tag{1}$$

$$(\langle right!m\rangle \wedge \ominus\Diamond\langle right!m'\rangle) \Rightarrow (\langle left!m\rangle \wedge \ominus\Diamond\langle left!m'\rangle) \tag{2}$$

$$(\langle left!m\rangle \wedge \ominus\Diamond\langle left!m'\rangle) \Rightarrow (m \neq m') \tag{3}$$

$$(\langle left!m\rangle) \Rightarrow \Diamond(\langle right!m\rangle) \tag{4}$$

Figure 6: Temporal Logic Specification of an Unbounded Buffer

$$\begin{aligned} &BUFFER = P_{<>} \\ &\qquad \text{where } P_{<>} = left?m \rightarrow P_{<m>} \\ &\qquad\quad \text{and } P_{<m>^\frown s} = (left?n \rightarrow P_{<m>^\frown s^\frown <n>} \mid right?m \rightarrow P) \end{aligned}$$

BUFFER **sat** $(right \leq left) \wedge$ (**if** $right = left$ **then** $left \notin ref$ **else** $right \notin ref$)

Figure 7: CSP Program and Specification of an Unbounded Buffer

specification. Without it, a buffer that outputs duplicate copies of its input would be considered correct. Whereas the first three predicates state safety properties of the system (and its environment), the fourth states a liveness property: each incoming message will eventually be transmitted.

CSP supports a model-oriented style of specifying concurrent processes and a property-oriented style of stating and proving properties about the model. CSP is based on model of *traces*, or event sequences, and assumes that processes communicate by sending messages across channels. Processes synchronize on events so that the event of sending an output message *m* on a named channel *c* is synchronized with the event of simultaneously receiving an input message on *c*. Figure 7 gives a CSP specification of an unbounded buffer (adapted from [37]). BUFFER itself is specified to be a process that acts as an unbounded buffer. If the buffer is empty and in the event that there is a message *m* on the *left* channel (*left?m*), it will input it. If the buffer is non-empty, then either the buffer will input another message *n* from the *left* channel, appending it to the end of the buffer, or output the first message in the buffer to the *right* channel.

Within CSP's underlying formalism, one can treat BUFFER as a CSP "program," and then state and prove properties about the traces it denotes. Using algebraic laws on traces one can formally verify that a given CSP "program" satisfies a specification on traces. In the example, BUFFER is the CSP "program" that describes a set of traces each of which satisfies the predicate given on the right-hand-side of **sat**. The predicate's first conjunct says that the sequence of (output) messages on the right channel is a prefix of the sequence of (input) messages on the left channel. The prefix property of sequences guarantees that only messages sent from the left will be delivered to the right, only once, and in the same order. The second conjunct says that the process never stops: it cannot *ref*use to communicate on either the right or left channel. This implies that input messages will eventually be delivered, which is similar to the fourth predicate in the temporal logic specification.

The aforementioned tool, B, used for proving theorems from Z specifications, has also been used to prove properties about CSP specifications [73]. Occam is a programming language derivative of CSP that has been implemented and used on transputers [52].

module BUFFER **with subroutines** PUT, GET

state functions:

buffer : **sequence of** *message*
parg : *message* **or** *NULL*
gval : *message* **or** *NULL*

initial conditions:

$|\ \mathit{buffer}\ | = 0$

safety properties

1. (a) $\mathit{at}(\mathrm{PUT}) \Rightarrow \mathit{parg} = \mathrm{PUT.PAR}$
   (b) $\mathit{after}(\mathrm{PUT}) \Rightarrow \mathit{parg} = \mathit{NULL}$
2. (a) $\mathit{at}(\mathrm{GET}) \Rightarrow \mathit{gval} = \mathit{NULL}$
   (b) $\mathit{after}(\mathrm{GET}) \Rightarrow \mathrm{GET.PAR} = \mathit{gval}$
3. **allowed changes to** *buffer*
   parg **when** *in*(PUT)
   gval **when** *in*(GET)
   (a) $\alpha[\mathrm{BUFFER}]{:}\mathit{in}(\mathrm{PUT}) \wedge \mathit{parg} \neq \mathit{NULL} \rightarrow$
   $\mathit{parg}' = \mathit{NULL} \wedge \mathit{buffer}' = \mathit{buffer} * \mathit{parg}$
   (b) $\alpha[\mathrm{BUFFER}]{:}\mathit{in}(\mathrm{GET}) \wedge \mathit{gval} = \mathit{NULL} \wedge |\ \mathit{buffer}\ | > 0 \rightarrow$
   $\mathit{gval}' \neq \mathit{NULL} \wedge \mathit{buffer} = \mathit{gval}' * \mathit{buffer}'$

liveness properties

4. $\mathit{in}(\mathrm{PUT}) \wedge |\ \mathit{buffer}\ | < \mathit{min} \leadsto \mathit{after}(\mathrm{PUT})$
5. $\mathit{in}(\mathrm{GET}) \wedge |\ \mathit{buffer}\ | > 0 \leadsto \mathit{after}(\mathrm{GET})$

Figure 8: State-function Specification of an Unbounded Buffer

Lamport's state function approach to specifying concurrent program modules is illustrated in Figure 8 (adapted from [48]). It combines an axiomatic-style specification for describing the behavior of individual operations with temporal logic assertions for specifying safety and liveness properties. For a module's set of operations, invocations of different operations can be active concurrently, but at most one invocation of a given operation can be active at once.

In the example, the functions, *buffer*, *parg*, and *gval* define the state of the buffer, which has two operations, PUT and GET, and an initial size of 0. The predicates $\mathit{at}(OP)$, $\mathit{in}(OP)$, and $\mathit{after}(OP)$ state whether control is at the point of calling the operation *OP*, within the execution of the *OP*, or at the point of return from *OP*. The first pair of safety properties states that the value of the state function *parg* is equal to the input parameter to PUT at the time of call and equal to *NULL* upon return. The second pair states similar properties for GET. The third pair of properties indicates how the state functions change as a result of executing PUT and GET: If control is in PUT, *buffer* gets updated by appending the non-*NULL* message to its end; if control is in GET and the buffer is non-empty, *buffer* gets updated by removing its first message, which is GET's return value *gval*. The fourth and fifth properties are liveness properties requiring that PUT return whenever there are fewer than *min* messages in the buffer and that GET return whenever the buffer is non-empty. (The operator $\leadsto$ stands for "leads to.") These requirements ensure that progress is made: that once control is within the PUT (or GET) operation, control will reach its corresponding return point. The fifth implies that messages received (through PUT) are eventually transmitted (through GET) since if control is in GET, it must eventually return.

Most methods that focus on the specification of behavioral properties of concurrent and distributed systems pay attention to only the formal specification language part of a formal method and not at all to the pragmatics. In fact, most pay attention to defining carefully the satisfies relation for a given semantic domain. What many of these methods lack are the niceties that formal methods for sequential systems provide: the syntactic sugar and software support tools. For certain specific theories or models for concurrent and distributed systems more "user-friendly" specification languages, e.g., LOTOS [1] and RAISE [62], are just beginning to appear.

5. Bounds of Formal Methods

5.1. Between the Ideal and the Real Worlds

Formal methods are based on mathematics but are not entirely mathematical in nature. There are two important boundaries between the mathematical world and the real world that users of formal methods must acknowledge. These boundaries are not unique to formal methods, but are common to all engineering disciplines by virtue of the fact that formal methods involve human users attempting to model the physical, real world.

The first boundary is crossed in the codification of the client's informally stated requirements. This mapping between informal to formal is typically achieved through an iterative process not subject to proof. A specifier might write an initial specification, discuss its implications with the client, and revise it as a result of the client's feedback. The formal specification is always only a mathematical representation of the client's requirements. On the one hand, any inconsistencies in the requirements would be faithfully preserved in the specifier's mapping. On the other, the specifier might incorrectly interpret the requirements and formally characterize the misinterpretation. For these reasons, it is important that specifiers and clients interact. Specifiers can help clients clarify their fuzzy, perhaps contradictory, notions; clients can help specifiers debug their specifications.

The second boundary is crossed in the mapping from the real world to some abstract representation of it. The formal specification language encodes this abstraction. For example, a formal specification might describe properties of real arithmetic, abstracting away from the fact that not all real numbers can be represented in a computer. The formal specification is only a mathematical approximation of the real world. The existence of this boundary should not be surprising since it is ubiquitous in all fields of engineering and applied mathematics.

5.2. Assumptions About the Environment

There is another kind of boundary that is often neglected by even experienced specifiers: the boundary between a real system and its *environment*. A system does not run in isolation; its behavior is affected by input from the external world which in turn consumes the system's output.

Given that we can formally model the system (in terms of a specification language's semantic domain) then if we can formally model the environment, we can formally characterize the interface between a system and its environment. Most formal methods leave the specification (formal or otherwise) of the environment outside of the specification of the system. An exception is the Gist language [21] used to specify *closed-systems*. In theory, a "complete" Gist specification includes not only a description of the system's behavior, but also of its users and other environmental factors like hardware.

A system's behavior as captured in its specification is conditional on the environment's behavior:

$$\mathit{Environment} \Rightarrow \mathit{System}$$

This implication says that if the environment satisfies some precondition, *Environment*, then the system will behave as specified in *System*. If the environment fails to satisfy the precondition, then the system is free to behave in any way. *Environment* is a set of assumptions. Whereas a system specifier places constraints on the system's behavior, he or she cannot place constraints on the environment, but can only make assumptions about its behavior. For example, in the temporal logic specification of the unbounded buffer, the assumption that messages are unique is an obligation expected of the environment to satisfy, not a property expected of the buffer to satisfy nor a constraint that the system specifier can place on the environment.

Implicitly assuming something about a system's environment is typically done when specifying something small like a procedure in a programming language because the environment is usually fixed or at least well-defined. A procedure's environment is defined in terms of the invocation protocol of the programming language. A procedure's specification will typically omit explicit mention of what the parameter passing mechanism is, or, for a compile-time type-checked language, that the types of the arguments are correct. The specifier presumably knows the details of the programming language's parameter passing mechanism, and assumes the programmer will compile the procedure, thereby do the appropriate type-checking.

Such (implicit) assumptions are not so grave an omission for specifications of program modules like procedures, functions, classes, and packages because one assumes the programming language's semantics is well-defined and that the language is correctly implemented. However, when what one is specifying is a large, complex, software and/or hardware system, one should take special care to make explicit as many assumptions about the environment as possible. Unfortunately, too often when one specifies a large system, one forgets to state explicitly the circumstances under which one expects the system to behave properly. Leave something unsaid now and you pay for it later.

In reality, one may not be able to model formally many environmental aspects such as natural castastrophes (lightning, hurricanes, earthquakes) or to anticipate completely all possible environmental behavior. Formal methods provide a means for systematically and precisely specifying what cases are covered and reasoning about how each case is handled; such reasoning holds only for those system input parameters that are made explicit. Unpredictable events and human error are difficult to incorporate as parameters in a formal model. At best, one might assume that the probability of a natural castastrophe is low or assume that no faults occur, in which case these assumptions should be made explicit.

6. Conclusions and Future Work

In a strict mathematical sense, formal methods differ greatly from one another. Not only does notation vary, but the choice of the semantic domain and definition of the satisfies relation both make a tremendous difference between what one can easily and concisely express in one method versus another. An idiom in one language might translate into a long list of unstructured statements in another or might not even have a counterpart.

But in a more practical sense, formal methods do not differ so radically from one another. Within some well-defined mathematical framework, they let system developers couch their ideas in a precise manner. The more rigor applied in system development, the more likely one gets the requirements stated "correctly," the more likely one gets the design "right," and of course the more precisely one can argue the correctness of the implementation.

One can conservatively, but justifiably conclude that existing formal methods can be used:

- To identify many, but not all, deficiencies in a set of informally stated requirements, to discover discrepancies between a specification and an implementation, and to find errors in existing programs and systems;
- To specify "medium-sized" and "non-trivial" problems, especially the functional behavior of sequential programs, abstract data types, and hardware.
- To gain a deeper understanding of the behavior of large, complex systems.

Many challenges remain. Researchers in the formal methods community are actively pursuing the following goals, spanning theory and engineering:

- To specify non-functional behavior such as reliability, safety, real-time, performance, and human factors;
- To combine different methods such as a domain-specific one with a more general one, or an informal one with a formal one;

- To build more usable and more robust tools, in particular tools to manage large specifications and tools to perform more complicated semantic analysis of specifications more efficiently, perhaps by exploiting parallel architectures and parallel algorithms;
- To build specification libraries like the Larch Handbook [32] so that systems and their components can be reused based on information captured in their specification;
- To demonstrate that existing techniques scale up to handle real-world problems and to scale up the techniques themselves;
- To integrate formal methods with the entire system development effort.

Acknowledgments

I thank Susan Gerhart for giving me this opportunity to express my thoughts about formal methods. I thank John Guttag and Jim Horning for being the ones who first introduced me to formal specifications. They have been instrumental in shaping my opinions about the role formal methods can and should play in the development of large, complex systems. I thank all those who attended Formal Methods '89 in Halifax, Nova Scotia for helpful feedback and discussion. I would especially like to thank Joseph Goguen who has suggested that one way to give a formal characterization of what constitutes a formal method would be in terms of institutions.

References

[1] DIS 8807. *Information Systems Processing–Open Systems Interconnection–LOTOS*. Technical Report, International Standards Organization, 1987.

[2] J.-R. Abrial. *B User Manual*. Technical Report, Programming Research Group, Oxford University, 1988.

[3] M. Alford. Srem at the age of eight: the distributed computing design system. *Computer*, 36–46, April 1985.

[4] K.R. Apt, N. Francez, and W.P. DeRoever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–385, July 1980.

[5] W.R. Bevier. *A Verified Operating System Kernel*. Technical Report 11, Computational Logic, Inc., March 1987.

[6] G. Birkhoff and J.D. Lipson. Heterogeneous algebras. *Journal of Combinatorial Theory*, 8:115–133, 1970.

[7] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, 1979. ACM monograph series.

[8] M.C. Browne, E.M. Clarke, and D. Dill. Checking the correctness of sequential circuits. In *Proc 1985 IEEE Int. Conf. Comput. Design*, pages 545–548, 1985.

[9] M. Broy. *A Fixed Point to Applicative Multiprogramming*, pages 565–623. Reidel Publishing Company, 1982.

[10] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. In *Proc. of Symp. on Operating Systems*, 1989.

[11] R.M. Burstall and J.A. Goguen. Putting theories together to make specifications. In *Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058, August 1977. Invited paper.

[12] K.M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, 1988.

[13] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2):244–263, 1986.

[14] E.M. Clarke and O. Grumberg. Research on automatic verification of finite-state concurrent systems. *Ann. Rev. Comput. Sci.*, 2:269–290, 1987.

[15] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 1984.

[16] W.J. Cullyer. Implementing safety-critical systems: the viper microprocessor. In *VLSI Specification, Verification and Synthesis*, Kluwer, 1987.

[17] F. DeRemer and H.H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Trans. on Soft. Eng.*, June 1976.

[18] E.W. Dijkstra. *Notes of Structured Programming*, pages 1–81. Academic Press, 1972.

[19] H.-D. Ehrich. *Extensions and Implementations of Abstract Data Type Specifications*, pages 155–164. Volume 64 of *Lecture Notes in Computer Science*, Springer-Verlag, Poland, 1978.

[20] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*. Springer-Verlag, Berlin, 1985.

[21] M. Feather. Language support for the specification and development of composite systems. *ACM Trans. on Prog. Lang.*, 9(2):198–234, April 1987.

[22] S.J. Garland and J.V. Guttag. Inductive methods for reasoning about abstract data types. In *Proceedings of the 15th Symposium on Principles of Programming Languages*, pages 219–228, January 1988.

[23] S.J. Garland, J.V. Guttag, and J. Staunstrup. Verification of vlsi circuits using lp. In *Proceedings of the IFIP WG 10.2, The Fusion of Hardware Design and Verification*, North-Holland, 1988.

[24] J.A. Goguen and J.J. Tardo. An introduction to obj: a language for writing and testing formal algebraic program specifications. In *Proceedings of the Conference on Specifications of Reliable Software*, pages 170–189, Boston, MA, 1979.

[25] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright. Abstract data types as initial algebras and correctness of data representations. In *Proceedings from the Conference of Computer Graphics, Pattern Recognition and Data Structures*, pages 89–93, ACM, May 1975.

[26] A. T. Goldberg. Knowledge-based programmin: a survey of program design and construction techniques. *IEEE Trans. Software Eng.*, 12(7):752–768, 1986.

[27] D.I. Good, R.L. London, and W.W. Bledsoe. An interactive program verification system. *IEEE Transactions on Software Engineering*, 1(1):59–67, 1979.

[28] M. Gordon. Hol: a proof generating system for higher-order logic. In *VLSI Specification, Verification and Synthesis*, Kluwer, 1987.

[29] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF*. Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, 1979.

[30] J.V. Guttag. *The Specification and Application to Programming of Abstract Data Types*. PhD thesis, University of Toronto, Toronto, Canada, September 1975.

[31] J.V. Guttag, J.J. Horning, and J.M. Wing. The larch family of specification languages. *IEEE Software*, 2(5):24–36, September 1985.

[32] J.V. Guttag, J.J. Horning, and J.M. Wing. *Larch in Five Easy Pieces*. Technical Report 5, DEC Systems Research Center, July 1985.

[33] J.V. Guttag, J.J. Horning, and J.M. Wing. Some remarks on putting formal specifications to productive use. *Science of Computer Programming*, 2(1), October 1982.

[34] D. Harel. On visual formalisms. *CACM*, 31(5):514–530, 1988.

[35] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring. Statemate: a working environment for the development of complex reactive systems. In *Proc. 10th IEEE Int'l Conf. on Software Engineering*, April 1988.

[36] A. Heydon, M. Maimone, J.D. Tygar, J.M. Wing, and A. Moormann Zaremski. Constraining pictures with pictures. In *Proceedings of IFIPS '89*, San Francisco, August 1989.

[37] C.A.R. Hoare. *Communicating sequential processes.* Prentice-Hall International, 1985.

[38] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(1):271–281, 1972.

[39] W.A. Hunt. *The Mechanical Verification of a Microprocessor Design.* Technical Report 6, Computational Logic, Inc., 1987.

[40] M.A. Jackson. *Principles of Program Design.* Academic Press, London, 1975.

[41] C.B. Jones. *Software Development: A Rigorous Approach.* Prentice-Hall International, 1980.

[42] C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, 1986.

[43] S. Kamin. Final data types and their specification. *ACM Transactions on Programming Languages and Systems*, 5(1):97–121, January 1983.

[44] D. Kapur and D. Musser. Proof by consistency. *Artificial Intelligence*, 31:125–157, 1987.

[45] H. Katzan. *Systems Design and Documentation: An Introduction to the HIPO Method.* Van Nostrand Reinhold, New York, 1976.

[46] R.A. Kemmerer and S.T. Eckmann. *A User's Manual for the UNISEX System.* Technical Report, Dept. of Computer Science, UCSB, Santa Barbara, CA, December 1983.

[47] R. Koymans, J Vytopil, and W.P. DeRoever. Real time programming and asynchronous message passing. In *2nd ACM Symp. on Principles of Distributed Programming*, pages 187–197, 1983.

[48] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[49] P. Lescanne. Computer experiments with the reve term rewriting system gnerator. In *Proceedings of the 10th Symposium on Principles of Programming Languages*, pages 99–108, Austin, Texas, January 1983.

[50] K.N. Levitt, L. Robinson, and B.A. Silverberg. *The HDM Handbook.* Technical Report Volumes 1-3, SRI International, Menlo Park, CA, 1979.

[51] R. Locasso, J. Scheid, D.V. Schorre, and P.R. Eggert. *The Ina Jo Reference Manual.* Technical Report TM-(L)-6021/001/000, System Development Corporation, Santa Monica, CA, 1980.

[52] INMOS Ltd. *Occam Programming Manual.* Prentice-Hall International, 1984.

[53] D.C. Luckham, D.P. Helmbold, D.L. Bryan, and M.A. Haberler. *Task Sequencing Language for Specifying Distributed Ada Systems*, pages 444–463. Springer-Verlag, 1987.

[54] D.C. Luckham and F.W. von Henke. An overview of anna, a specification language for ada. *IEEE Software*, 2(2):9–23, March 1985.

[55] N. Lynch and M. Tuttle. *Hierarchical Correctness Proofs for Distributed Algorithms.* Technical Report, MIT Laboratory for Computer Science, Cambridge, MA, April 1987.

[56] Z. Manna and A. Pnueli. *Verification of concurrent Programs, Part I: The Temporal Framework.* Technical Report STAN-CS-81-836, Dept. of Computer Science, Stanford University, June 1981.

[57] P.R. McMullin and J.D. Gannon. Combining testing with formal specifications: a case study. *IEEE Trans. on Soft. Eng.*, 9(3), May 1983.

[58] A.J.R.G. Milner. A calculus of communicating systems. In *Lecture Notes in Computer Science 92*, Springer-Verlag, 1980.

[59] A.P. Moore. Investigating formal specification and verification techniques for comsec software security. In *Proceedings of the 1988 National Computer Security Conference*, October 1988.

[60] D. Musser. Abstract data type specification in the affirm system. *IEEE Transactions on Software Engineering*, 6(1):24–32, January 1980.

[61] R. Nakajima, M. Honda, and H. Nakahara. Hierarchical program specification and verification– a many-sorted logical approach. *Acta Informatica*, 14:135–155, 1980.

[62] M. Nielsen, K. Havelund, K.R. Wagner, and C. George. The raise language, method and tools. *Formal Aspects of Computing*, 1:85–114, 1989.

[63] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.

[64] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[65] D.L. Parnas. A technique for software module specification with examples. *CACM*, 15:330–336, May 1972.

[66] J.L. Peterson. Petri nets. *Computing Surveys*, 9(3), September 1977.

[67] A. Pnueli. *Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends*, pages 510–584. Springer-Verlag, 1986.

[68] V. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, February 1986.

[69] L. Robinson and O. Roubine. *SPECIAL - A Specification and Assertion Language*. Technical Report CSL-46, Stanford Research Institute, Menlo Park, Ca., January 1977.

[70] D. Scott. Toward a mathematical semantics for computer languages. In J. Fox, editor, *Proc. Symposium on Computers and Automata*, Polytechnic Institute of Brooklyn Press, 1971.

[71] J.M. Spivey. *Introducing Z: a Specification Language and its Formal Semantics*. Cambridge University Press, 1988.

[72] M. Wand. Final algebra semantics and data type extensions. *Journal of Computer and System Sciences*, 19(1):27–44, August 1979.

[73] J.C.P. Woodcock. Transaction processing primitives and csp. *IBM Journal of Research and Development*, 31(5):535–45, 1987.

[74] E. Yourdon and L.L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Programs and Systems Design*. Yourdon Press, New York, 1978.

[75] P. Zave. An operational approach to requirements specification for embedded systems. *IEEE Trans. Software Eng.*, 8(3):250–269, May 1972.

[76] S.N. Zilles. Abstract specifications for data types. IBM Research Laboratory, San Jose, CA, 1975.