A Specifier's Introduction to Formal Methods

Jeannette M. Wing

21 May 1990
CMU-CS-90-136

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.

A version of this paper will appear in *IEEE Computer*, September 1990. This TR supersedes CMU-CS-TR-89-200, "What is a Formal Method?"

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. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the National Science Foundation or the U.S. Government.

Keywords: Formal methods, specification languages, software engineering, abstract data types, concurrent and distributed systems.

A Specifier's Introduction to Formal Methods

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

May 21, 1990

1. Introduction

Formal methods used in the development of computer systems are mathematically-based techniques for describing system properties. Formal methods provide frameworks within which people can specify, develop, and verify systems 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 also addresses 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 customer'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 method to be formal, it must have a well-defined mathematical basis; it need not address any pragmatic considerations, but then it 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, 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 encourage the reader to read Meyer's article [78] on the benefits of formal specification and Lamport's article [64] for further remarks on the distinction between a method and a language, and what it means to specify a computer system.

We begin in Section 2 by first defining those elements necessary to give a formal basis to a formal method, and then address in Section 3 pragmatic concerns. In Section 4 we present examples of the application of six different methods. The accompanying survey article in this issue contains a more comprehensive listing of example methods, plus citations. We discuss in Section 5 some of the theoretical bounds of formal methods, and close in Section 6 by listing some general conclusions and directions of current research.

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 [48]. Burstall and Goguen use the term "language" [17] and later the term "institution" [39] for our notion of a formal specification language.

Definition 1 *A* formal specification language *is a triple, <Syn, Sem, Sat>, where Syn and Sem are sets and* $Sat \subseteq Syn \times Sem$, *is a relation between them. Syn is called the language's* syntactic domain*; Sem, its* semantic domain*; Sat, its* satisfies *relation.*

Definition 2 *Given a specification language, <Syn, Sem, Sat>, if Sat(syn, sem), then syn is a* specification *of sem, and sem is a* specificand *of syn.*

Definition 3 *Given a specification language, <Syn, Sem, Sat>, the* specificand set *of a specification syn in Syn is the set of all specificands sem in Sem such that Sat(syn, sem).*

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. 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

We usually define a specification language's syntactic domain in terms of a set of symbols, e.g., constants, variables, and logical connectives, and a set of grammatical rules for combining 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" are used to specify algebras [40, 32], theories [16, 81], and programs [46]. Though specifications written in these different specification languages range over different semantic domains, they often look syntactically similar.

- "Concurrent and distributed systems specification languages" are used to specify state sequences [18, 74, 65], event sequences [52, 74], state and transition sequences [33], streams [13], synchronization trees [79], partial orders [90], and state-machines [73].
- Programming languages are used to specify functions from input to output [96], computations [100], predicate transformers [29], relations [21], and machine instructions [3].

Note that each programming language (with a well-defined formal semantics) is a specification language, but not vice versa because specifications in general do not have to be executable on some machine whereas programs do. By using a more "abstract" specification language, we gain the advantage of not being 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.

Programs, however, are formal objects, susceptible to formal manipulation, e.g., compilation and execution. Thus programmers cannot escape from formal methods; the question is whether they work with informal requirements and (formal) programs, or whether they use additional formalism to assist them.

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. An implementation *prog* is *correct* with respect to a given specification *spec* if *prog* satisfies *spec*. More formally,

Definition 4 *Given a specification language, <Syn, Sem, Sat>, an implementation prog in Sem is* correct *with respect to a given specification spec in Syn if and only if Sat(spec, prog).*

2.3. Satisfies Relation

We often would like to specify different aspects of a single specificand, perhaps using different specification languages. For example, we might want to specify the functional behavior of a collection of program modules as the composition of the functional behaviors of the individual modules. We might additionally 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 "views" of a specificand, we first associate with each specification language a *semantic abstraction function*, which partitions specificands into equivalence classes.

Definition 5 *Given a semantic domain, Sem, a* semantic abstraction function *is a homomorphism, $A: Sem \rightarrow 2^{Sem}$, that maps elements of the semantic domain into equivalence classes.*

For a given specification language, we choose a semantic abstraction function so as to induce an *abstract satisfies relation* between specifications and equivalence classes of specificands. This relation defines a view on specificands.

Definition 6 *Given a specification language, <Syn, Sem, Sat>, and a semantic abstraction function A defined on Sem, an* abstract satisfies relation, *ASat: Syn* $\rightarrow$ 2^{Sem}, *is the induced 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 (denoted by a solid-lined blob), and via ASat2, specification B of syntactic domain Syn2 maps to a different equivalence class of specificands (denoted 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. The behavioral constraint that most formal methods address is a system's required functionality, i.e., mapping from inputs to outputs. 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. Example structural specification languages are module interconnection languages [28]. Structural specifications capture various kinds of

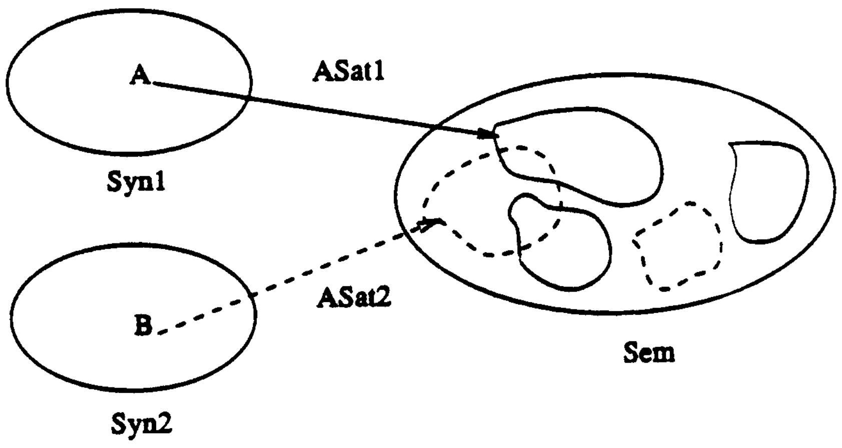

Figure 1: Abstract Satisfies Relations

hierarchical [86] and "uses" relations such as those represented by procedure call graphs, data dependency diagrams, and definition-use chains. 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.

2.4. Properties of Specifications

Each specification language should be defined so that each well-formed specification written in the language is *unambiguous.*

Definition 7 *Given a specification language, <Syn, Sem, Sat>, a specification syn in Syn is* unambiguous *if and only Sat maps syn to exactly one specificand set.*

Informally, 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.*

Definition 8 *Given a specification language, <Syn, Sem, Sat>, a specification syn in Syn is* consistent *(or* satisfiable*) if and only if Sat maps syn to a non-empty specificand set.*

Informally, a specification is consistent 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. Viewing a specification as a set of facts, consistency implies that we cannot derive anything contradictory from the specification. If we were to pose a question based on a consistent specification, we will not get mutually exclusive answers. It is obvious that we want to have consistent specifications, since an inconsistent specification means we have 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 [45]). In practice, we must usually deal with incomplete specifications. Why? Specifiers may intentionally leave some things unspecified, giving the implementor some freedom to choose among different data structures and algorithms. Also, specifiers cannot realistically anticipate all possible scenarios in which a system will be run, and thus, perhaps unwittingly have left some things unspecified. Finally, specifiers develop specifications gradually and iteratively, perhaps in response to changing customer requirements, and hence work 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. Specifiers want 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 is guilty of *implementation bias* [57]. Informally, a specification has implementation bias if it specifies externally unobservable properties of its specificands; hence, it places unnecessary constraints on its specificands. For example, a set specification that keeps track of the insertion order of its elements has implementation bias, e.g., toward an ordered-list representation and against a hash table representation.

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*, mapping a set of well-formed sentences in the specification language to a set of well-formed sentences.

We use this inference system to prove properties from the specification about specificands. Again taking a specification as a set of facts, we derive new facts through the application of the inference rules. When we prove a statement inferrable from these facts, then we prove 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,

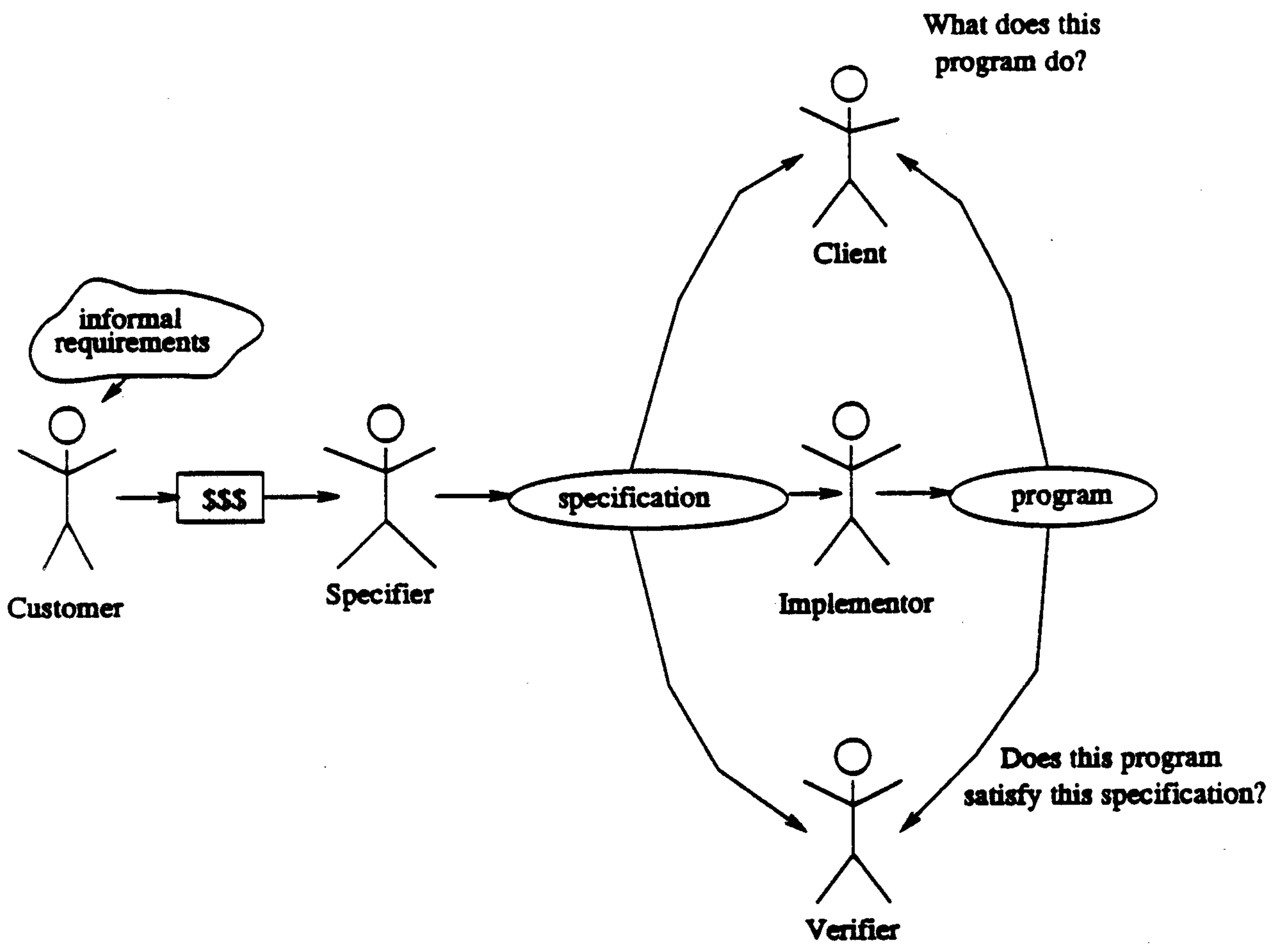

Figure 2: Specification Users

in the form of conjectures, 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 they were able to prove a surprising result from the specification, then perhaps the specification is wrong.

A formal method with an explicitly defined inference system usually has the further advantage that this system can be completely mechanized, e.g., if it has 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. Users

Some users of formal methods are actually going to produce something tangible—formal specifications. However, not everyone need write specifications to benefit from using formal methods. Most people need only read specifications, not develop their own from scratch. Besides specification writers, there are different kinds of specification readers. Figure 2 depicts a scenario in which each stick figure represents a different role in the system development process. A person playing any of these roles is a potential specification user. In practice, one person may play multiple roles, and some role may not be played at all.

Specifiers (in red) write, evaluate, analyze, and refine specifications. They prove that their refinements preserve certain properties and prove properties of specificands through specifications. Specification readers (in blue), besides specifiers, are *customers*, those people who may have hired the specifiers; *implementors*, those people who realize a specification; *clients*, those people who use a specified system, usually without knowledge of how it is implemented; and *verifiers*, those people who prove the correctness of implementations. All these people can benefit from the assistance of *machine tools* (another kind of specification "reader"), some of which might blindly manipulate specifications without regard to their meaning.

One point of tension in many formal methods is that its language may be more suitable to one type of specification user than to others. Most language designers will target their language for least two, e.g., clients and specifiers, or specifiers and implementors. Some specification languages have a lot of syntactic sugar to make specifications more readable by customers. 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.

An advocate of a particular formal method should tell potential users the method's domain of applicability. 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.

A formal method's set of guidelines should identify different types of users the method is targeted for and what the abilities of each should be. To apply some methods properly, users might need to know modern algebra, set theory, and/or predicate logic. To apply some domain-specific methods, users might need to know additional mathematical theories such as digital logic, e.g., if specifying hardware, or probability and statistics, e.g., if specifying system reliability.

3.2. 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 formalizing rather than the end result. Gaining a deeper understanding of the specificand by forcing ourselves to be abstract, yet precise, about the desired properties of a system can sometimes be more rewarding than having the specification document alone.

Let us consider, for each system development phase, some of the uses of formal specifications and example formal methods that support these uses.

Requirements analysis. Applying a formal method helps to clarify a customer's set of informally stated requirements. A specification helps to crystallize the customer'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 customer's responses through the use of a formal specification rather than through an informal one. Both the customer and specifier can pose and answer questions based on the specification to see whether it reflects the customer's intuition and whether the specificand set has the desired set of properties. Systems such as KATE [34] and the Requirements Apprentice [91] address the problem of transforming informal requirements into formal specifications; the Gist explainer [98] addresses the converse problem of translating a formal specification into a restricted subset of English.

System design. Two of the most important activities during design are *decomposition* and *refinement*. The Vienna Definition Method (VDM) [58], Z [97], Larch [46], and Lamport's transition axiom method [64] are formal methods that are especially suitable for system design.

Decomposition is the process of partitioning a system into smaller modules. Specifiers can write specifications to capture precisely the interfaces between these modules. Each interface specification provides the module's client the information needed to use the module without knowledge of its implementation. The interface specification simultaneously provides the module's implementor the information needed to implement the module without knowledge of its clients. 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 clients. 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.

Refinement involves working 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. Each refinement step requires showing that a specification (or program) at one level "satisfies" a higher-level specification. The process of proving satisfaction often generates additional assumptions, called *proof obligations*, that must be discharged for the proof to be valid. A formal method provides the language to state these proof obligations precisely and the framework to carry out the proof itself.

Program refinement dates back to Dijkstra's work on stepwise refinement [30] and predicate transformers [29], and Hoare's work on data representation [53] and abstraction functions [54]. Related work on program transformation [15, 6], program synthesis [75], and inferential programming [95] spawned more recent activity exemplified by the design of languages like Refine [41] and Extended ML [94], and programming environments like CIP-S [8] and the Ergo Support System [66]. Whereas these refinement approaches are based on classical mathematical logic, an alternative approach to program development based on constructive logic [76] gave rise to proof development environments like NuPRL [23] in which programs are proofs, and vice versa.

System verification. Verification is the process of showing a system satisfies its specification. Formal verification is impossible without a formal specification. Though in practice we may never completely verify an entire system, we 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.) Systems such as Gypsy [42], HDM [69], FDM [70], and M-eves [25] evolved as a result of a primary focus on program verification. HOL [43] was originally developed for hardware verification.

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 a module's use 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; the specification must be taken 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.

Only a few formal methods have been developed explicitly for helping the testing process. Three examples are: the DAISTS system [77], used for testing implementations of abstract data types; Kemmerer's symbolic execution tool [62], used to generate and execute test cases from Ina Jo specifications [70]; and the Task Sequencing Language (TSL) Runtime System [93], used to check automatically the execution of Ada tasking statements against TSL specifications.

System documentation. A specification is a description alternative to the implementation of the system itself. 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*. A client 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, developers should do a critical analysis of its functionality and performance once it has been built and tested. Does the system do what the customer wants? Does it do it fast enough? If formal methods were used in its development then they can help system developers formulate and answer these questions. The specification serves as a reference point. In the case that the customer 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 [14] and circuit designs [12]—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 VLSI circuits [37, 20, 82, 38], microprocessors [55, 26], oscilloscopes [27], operating systems kernels [9], distributed databases [24, 22], and secure systems [80]. Most formal methods have not yet been applied to specifying large-scale software and/or hardware systems; most are still inadequate to specify many important 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, e.g., programs, proofs, and test suites, and their structural interrelationships.

The problem of dealing with a specificand's inherent complexity remains, since no magic will make it disappear. System complexity results from internal complexity and/or interface complexity. For example, an optimizing compiler is internally more complex than a non-optimizing one for the same language; yet, in principle, they would both provide the same simple interface to their clients (e.g., "compile *<program_name>*"). By providing a systematic way of thinking and reasoning about specificands, formal methods are precisely what can help humans grapple with both kinds of system complexity.

3.3. Characteristics

A formal method's characteristics, such as whether its language is graphical or whether its underlying logic is first-order, influence the style in which a user applies it. It is not the subject of this paper to give a complete taxonomy of all possible characteristics of a method nor to classify exhaustively all methods according to these characteristics. Instead, we give a partial listing of different characteristics, noting that a method typically reflects a combination of many different ones.

3.3.1. Model- Versus Property-Oriented

Two broad classes of formal methods are called *model-oriented* and *property-oriented*. Using a model-oriented method, a specifier 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. Using a property-oriented method, a specifier 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. A specifier following a property-oriented method tries to state in a specification no more than the necessary minimal constraints on the system's behavior. The fewer the properties specified, the more the possible implementations that will satisfy the specification.

Example model-oriented methods used for specifying the behavior of sequential programs and abstract data types include Parnas's state-machines, [87], Robinson and Roubine's extensions to them with V-, O-, and OV-functions [92], VDM and Z. Example methods used for specifying the behavior of concurrent and distributed systems include Petri Nets [88], Milner's Calculus of Communicating Systems (CCS) [79], Hoare's Communicating Sequential Processes (CSP) [52], Unity [18], I/O automata [73], and TSL. The Raise Project represents more recent work on combining VDM and CSP [83].

Property-oriented methods can be further broken into two categories, sometimes referred to as *axiomatic* and *algebraic*. Axiomatic methods stem from Hoare's work on proofs of correctness of implementations of abstract data types [54], where first-order predicate logic pre- and post-conditions are used for the specification of each operation of the type. Iota [81], Anna [72], and Larch are example specification languages that support an axiomatic method. In an algebraic method, data types and processes are defined to be heterogeneous algebras [10]. 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 [40, 45, 105, 16, 31, 99, 59] including the handling of error values, nondeterminism, and parameterization. The more widely-known specification languages that have evolved from this work are CLEAR [16], OBJ [35], and ACT ONE [32].

Example property-oriented methods used for specifying the behavior of concurrent and distributed systems include extensions to the Hoare-axiom method [84, 5], temporal logic [89, 74, 85], and Lamport's transition axiom method. 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, 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 [49], used to specify state transitions in "reactive" systems. Figure 3 gives a simple example of a statechart that describes the behavior of a one-slot buffer. Rounded rectangles ("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.

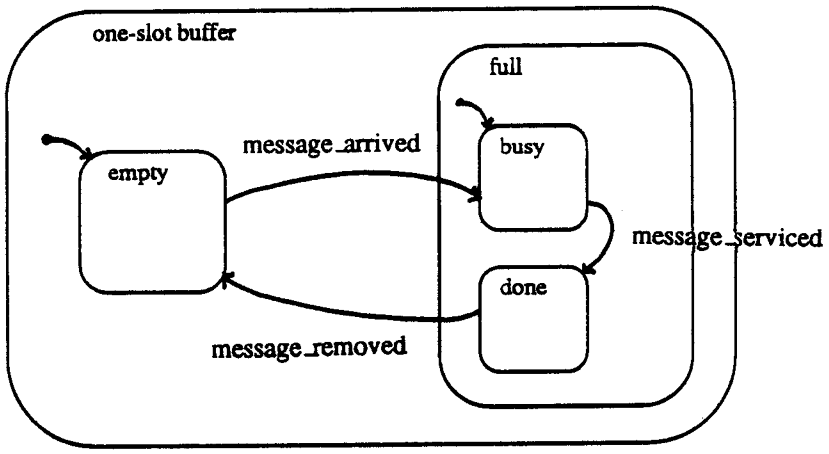

Figure 3: Statechart Specification of a One-Slot Buffer

Harel's higraph notation inspired the design of the Miró visual languages, which are used to specify security constraints [51]. Like statecharts, the Miró languages have a formally defined semantics and tool support.

Many informal methods use visual notations. These 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 [56], HIPO [61], Structured Design [103] and SREM [4] are examples of semi-formal methods that use pictures.

3.3.3. Executable

Some formal methods support *executable* specifications, specifications that can be run on a computer. An executable specification language is by definition more restricted in expressive power than a non-executable language because its functions must be computable and defined over domains with finite representations. 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. Specifiers 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 [50] is a tool that lets users run simulations through the state transition diagram represented by a statechart.

Besides statecharts, executable specification languages include OBJ, an algebraic specification language; Prolog [21], a logic programming language, which when used in a property-oriented style lets specifiers state logical relations on objects; and PAISley [104], a model-oriented 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-Supported

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 [12, 19] are especially useful for specifying and verifying properties of VLSI circuits.

Proof checking tools that let users treat algebraic specifications as rewrite rules include Affirm, Reve [67], the Rewrite Rule Laboratory (RRL) [60], and the Larch Prover [36]. Tools (and their associated specification language) that handle subsets of first-order logic include the Boyer-Moore Theorem [11] Prover (and the Gypsy specification language), FDM (Ina Jo), HDM (SPECIAL), and m-EVES (m-Verdi). Finally, tools that handle subsets of higher-order logics include HOL, LCF [44], and OBJ [38].

4. Some Examples

The purpose of this section is to illustrate a few example methods to give the reader a taste of some of the more well-known or commonly-used ones. We present 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 Lamport's transition axiom method, which are formal methods especially appropriate for specifying properties of concurrent and distributed systems.

Sometimes when specifying the same problem using different methods, the resulting specifications look remarkably similar (e.g., the first three examples); sometimes not (e.g., the last three examples). The similarity or difference is attributable sometimes to the nature and/or simplicity of the specificand and sometimes to the methods themselves. The choice of a method is likely to affect what a specification says and how it is said. A method's guidelines may encourage the specifier to be explicit about some system behaviors, e.g., state changes, and not others, e.g., error handling. Syntactic conventions (e.g., indentation style), special notation (e.g., vertical and horizontal lines), and keywords affect a specification's physical appearance and its readability.

Most proponents of methods that are used primarily to specify behavioral properties of concurrent and distributed systems have historically paid attention to defining carefully the satisfies relation for a given semantic domain. What many of their 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 of concurrent and distributed systems more "user-friendly" specification languages, e.g., LOTOS and RAISE, are just beginning to appear.

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 4 gives a model-oriented specification of a symbol table. We follow the Z notation of Spivey's book [97]. The state of the table is modeled by a partial mapping from keys to values ($X \leftrightarrow Y$ denotes a set of partial mappings from set X to set Y; a partial mapping relates each member of X to at most one member of Y.) 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 four operations on the table, INIT, INSERT, LOOKUP, and DELETE. INIT initializes the symbol table to be empty. INSERT modifies the table by adding a new binding to st, in the case that the key k is not already in the domain of st. LOOKUP requires that the key k be in the domain of the mapping, returns the value to which k is mapped, and does not change the state of the symbol table (st' = st). DELETE also requires that the key k be in the domain of the mapping, and modifies the table by deleting the binding associated with k from st ($\lhd$ is a domain subtraction operator). The proof checker B [2] has been used for proving theorems based on Z specifications.

VDM supports a model-oriented specification style. VDM defines a set of built-in data types, e.g., sets, lists, and mappings, which specifiers use to define other types. The VDM specification in Figure 5 defines a symbol table also in terms of a **mapping** from keys to values. We follow the VDM notation given in Jones's book [57]. The behaviors of the INIT, INSERT, LOOKUP and DELETE operations are the same as specified in the Z specification; however, the pre-conditions, specified in **pre** clauses, are made explicit and separate from the post-conditions, specified in **post** clauses. A pre-condition on an operation is a predicate that must hold in the state upon each invocation of the operation; if it does not hold, then the operation's behavior is unspecified. A post-condition is a predicate that holds in the state upon return. An operation's clients are responsible for satisfying pre-conditions and its implementor is responsible for guaranteeing the post-condition. The fact that LOOKUP does not modify the symbol table (hence $st' = st$), but INSERT and DELETE do, is specified 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.

Larch is a property-oriented method that combines both axiomatic and algebraic specifications into a "two-tiered" specification [101]. The axiomatic component specifies state-dependent behavior, e.g., side effects and exceptional termination, of programs; the algebraic component specifies state-independent properties of data accessed by programs. Figure 6 gives a Larch specification of the symbol table example. We follow the Larch notation given in [47].

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 its pre-and post-conditions. The **modifies** clause lists those objects whose value may possibly change as a result of executing the operation. Hence, lookup is

$ST \mathrel{\hat{=}} KEY \nrightarrow VAL$

INIT

$st' : ST$

$st' = \{ \}$

INSERT

$st, st' : ST$

$k : KEY$

$v : VAL$

$k \notin dom(st) \land$

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

LOOKUP

$st, st' : ST$

$k : KEY$

$v' : VAL$

$k \in dom(st) \land$

$v' = st(k) \land$

$st' = st$

DELETE

$st, st' : ST$

$k : KEY$

$k \in dom(st) \land$

$st' = \{ k \} \lhd st$

Figure 4: Z Specification of a Symbol Table

$ST =$ **map** *Key* **to** *Val*

INIT( )

ext **wr** $st : ST$

post $st' = \{\}$

INSERT($k : Key, v : Val$)

ext **wr** $st : ST$

pre $k \notin$ **dom** st

post $st' = st \cup \{k \mapsto v\}$

LOOKUP($k : Key$) $v : Val$

ext **rd** $st : ST$

pre $k \in$ **dom** st

post $v' = st(k)$

DELETE($k : Key$)

ext **wr** $st : ST$

pre $k \in$ **dom** st

post $st' = \{k\} \mathbin{\lhd\!\!\!-} st$

Figure 5: 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)
        requires ~ isin(s, k)
        modifies (s)
        ensures s' = add(s, k, v)
    lookup = proc (s: symbol_table, k: key) returns (v: val)
        requires isin(s, k)
        ensures 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 (find, 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
        converts (rem, find, isin) exempting (rem(emp), find(emp))
    end SymTab
```

Figure 6: Larch Specification of a Symbol Table

not allowed to change the state of its symbol table argument, but insert and delete are. One difference, not shown in the example, between Larch and VDM (and Larch and Z), is that if the target programming language supports exception handling, the interfaces would specify whether and under what conditions an operation signals exceptions. For example, we could remove insert's **requires** clause and, instead, use a special **signals** clause in its post-condition to specify that a signal should be raised in the case that the key k is already in the symbol table.

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 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 are 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 **requires** and **signals** clauses in the interface specification deal with these "error values". The **converts** and **exempting** clauses together provide a way to state 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 $\cup$ and $\lessdot$ used in the Z and VDM specifications. 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 [47] 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 finite mappings, partial orders, sets, and sequences.

4.2. Concurrency: Temporal logic, CSP, Transition axioms

As mentioned in Section 2.2, many formal methods for specifying the behavior of concurrent and distributed systems differ because of their choice in semantic domain. To be more concrete here, we will model a system's behavior as a set of linear sequences of states and associated events, where sometimes we will focus on just the states and sometimes on just the events. An alternative approach, used by CCS and EMC, is to model a system's behavior as a set of trees of states and associated events. When a specification is interpreted with respect to sets of sequences, it is common to separate properties of concurrent and distributed systems into two general categories, *safety* and *liveness* [65]. Safety properties ("nothing bad ever happens") include functional correctness and liveness properties ("something good eventually happens") include termination.

Temporal logic is a property-oriented method for specifying properties of concurrent and distributed systems. For a given temporal logic inference system, special ***modal operators*** are used to state concisely assertions about system behavior. Specifiers use these operators to refer to past, current, and future states (or events). There is no one standard temporal logic inference system nor one standard set of operators. Modal operators commonly used are $\Box$, $\Diamond$, and $\bigcirc$. Informally, when interpreted with respect to a sequence of states, $\Box P$ says in ***all*** future states the state predicate P holds, $\Diamond P$ says in ***some*** future state P will hold, and $\bigcirc P$ says in the ***next*** state P will hold. For example, $P \Rightarrow \Diamond Q$ says that if P holds in the current state then eventually Q will hold. Temporal logic notation tends to be terse and a temporal logic specification is simply an unstructured set of predicates all of which must be satisfied by a given implementation.

Figure 7 gives a temporal logic specification of the behavior of an unbounded buffer in an asynchronous environment. The example is adapted from [63] using the temporal logic system in [89], which has twelve different modal operators. The formula are interpreted with respect to sequences of events. 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,

$$\langle \mathit{right}!m \rangle \Rightarrow \diamondminus \langle \mathit{left}!m \rangle$$

states that any message transmitted to the right channel ($\langle \mathit{right}!m \rangle$) must have been previously placed on the left channel ($\diamondminus \langle \mathit{left}!m \rangle$) The second predicate,

$$(\langle \mathit{right}!m \rangle \wedge \circledequal \diamondminus \langle \mathit{right}!m' \rangle) \Rightarrow \diamondminus(\langle \mathit{left}!m \rangle \wedge \circledequal \diamondminus \langle \mathit{left}!m' \rangle)$$

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 ($\circledequal \diamondminus \langle \mathit{right}!m' \rangle$), then there must have been a preceding (the second $\diamondminus$) event of placing m on the input channel ($\langle \mathit{left}!m \rangle$), and moreover, an even earlier event that placed m' on the input channel ahead of m ($\circledequal \diamondminus \langle \mathit{left}!m' \rangle$). The third predicate,

$$(\langle \mathit{left}!m \rangle \wedge \circledequal \diamondminus \langle \mathit{left}!m' \rangle \Rightarrow (m \neq m')$$

states that all messages are unique: For each message m currently placed on the input channel and for each previously placed message m' ($\circledequal \diamondminus \langle \mathit{left}!m' \rangle$), m and m' are not equal. This property is not a property of the buffer, but an assumption on the environment. This assumption is essential to the validity of the 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 predicate,

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

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

$$(\langle right!m \rangle \wedge ⊜ \diamondminus \langle right!m' \rangle) \Rightarrow \;\diamondminus (\langle left!m \rangle \wedge ⊜ \diamondminus \langle left!m' \rangle) \tag{2}$$

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

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

Figure 7: Temporal Logic Specification of an Unbounded Buffer

states a liveness property: each incoming message will eventually be transmitted.

CSP uses a model-oriented method for specifying concurrent processes and a property-oriented method for 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 8 gives a CSP specification of an unbounded buffer (adapted from [52]). BUFFER itself is specified to be a process P that acts as an unbounded buffer. The recursive definition of P is divided into two clauses to handle the empty and non-empty cases. The first clause,

$$P_{<>} = left?m \rightarrow P_{<m>}$$

says that if the buffer is empty, in the event that there is a message m on the $left$ channel ($left?m$), it will input it. In CSP, if x is an event and P is a process, the notation $x \rightarrow P$ denotes a process that first engages in the event x and then behaves exactly as described by P. The second clause,

$$P_{<m>\hat{\ }s} = (left?n \rightarrow P_{<m>\hat{\ }s\hat{\ }<n>} \mid right!m \rightarrow P_s)$$

says that 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. CSP uses $s\hat{\ }t$ to denote the concatenation of sequence s to sequence t. It uses $\mid$ to denote choice: If x and y are distinct events, $x \rightarrow P | y \rightarrow Q$ describes a process that initially engages in either x or y; after this first event, subsequent behavior is described by P if the first event was x and Q if the first event was y.

Within CSP's formalism, BUFFER is a CSP program; we can state and prove properties about the traces it denotes. Using algebraic laws on traces we can formally verify that a given CSP program satisfies a specification on traces. The last line in Figure 8 states that BUFFER 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. CSP uses the notation $s \leq t$ to denote that the

$BUFFER = P_{<>}$

$$\text{where } P_{<>} = left?m \rightarrow P_{<m>}$$

$$\text{and } P_{<m>\frown s} = (left?n \rightarrow P_{<m>\frown s\frown <n>} \mid right!m \rightarrow P_s)$$

$$\text{BUFFER } \textbf{sat} \ (right \leq left) \wedge (\textbf{ if } right = left \textbf{ then } left \notin ref \textbf{ else } right \notin ref)$$

Figure 8: CSP Program and Specification of an Unbounded Buffer

sequence s is a prefix of sequence t. 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 the same property as stated in the temporal logic specification's fourth predicate.

B, previously mentioned for proving theorems from Z specifications, has also been used to prove properties of CSP specifications [102]. Occam is a programming language derivative of CSP that has been implemented and used on Transputers [71].

Lamport's transition axiom method combines an axiomatic method for describing the behavior of individual operations with temporal logic assertions for specifying safety and liveness properties. In the buffer example of Figure 9 (adapted from [65]) we use Lamport's original notation, although he introduced two other notations in a more recent description of his method [64].

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. For this example, we assume that invocations of different operations can be active concurrently, but at most one invocation of a given operation can be active at once. The predicates $at(OP), in(OP)$, and $after(OP)$ state whether control is at the point of calling the operation OP, within the execution of 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 $*$ denotes appending an element to a sequence.) 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 temporal logic 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

module BUFFER **with subroutines** PUT, GET

state functions:

$buffer$: **sequence of** $message$

$parg$: $message$ **or** $NULL$

$gval$: $message$ **or** $NULL$

initial conditions:

$| buffer | = 0$

safety properties

1. (a) $at(\text{PUT}) \Rightarrow parg = \text{PUT.PAR}$

   (b) $after(\text{PUT}) \Rightarrow parg = NULL$

2. (a) $at(\text{GET}) \Rightarrow gval = NULL$

   (b) $after(\text{GET}) \Rightarrow \text{GET.PAR} = gval$

3. **allowed changes to** $buffer$

   $parg$ **when** $in(\text{PUT})$

   $gval$ **when** $in(\text{GET})$

   (a) $\alpha[\text{BUFFER}]{:}in(\text{PUT}) \wedge parg \neq NULL \rightarrow$

   $parg' = NULL \wedge buffer' = buffer * parg$

   (b) $\alpha[\text{BUFFER}]{:}in(\text{GET}) \wedge gval = NULL \wedge | buffer | > 0 \rightarrow$

   $gval' \neq NULL \wedge buffer = gval' * buffer'$

liveness properties

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

Figure 9: Transition Axiom Specification of an Unbounded Buffer

must eventually return.

Unlike the temporal logic and CSP examples, but like the Z, VDM, and Larch examples, the last example uses keywords and distinct clauses for highlighting a model of state (**state functions**), state initialization (**initial conditions**), and state changes (**allowed changes to**). Again, unlike the temporal logic and CSP examples, it uses similar notational conveniences to highlight synchronization conditions (the *enabling predicates* to the left-hand-side of $\rightarrow$) and safety and liveness constraints on the processes' behaviors. Hence, this last example shows a combination of linguistic features borrowed from formal methods used to specify sequential programs and others used to specify concurrent ones.

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.

The first boundary is crossed in the codification of the customer's informally stated requirements. Figure 10 illustrates this mapping where the cloud symbolizes the customer's informal requirements and the oval symbolizes a formal specification of them. This mapping from 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 customer, and revise it as a result of the customer's feedback. The formal specification is always only a mathematical representation of the customer'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 customers interact. Specifiers can help customers clarify their fuzzy, perhaps contradictory, notions; customers can help specifiers debug their specifications. The existence of this boundary should not be surprising because human beings use formal methods.

The second boundary is crossed in the mapping from the real world to some abstract representation of it. Figure 11 illustrates this mapping where the cloud symbolizes the real world and the oval symbolizes an abstract model 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. This boundary is not unique to formal methods, or computer science in general; it is ubiquitous in all fields of engineering and applied mathematics [7, 64].

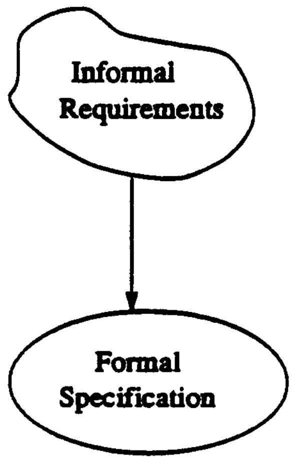

Figure 10: Mapping Informal Requirements to a Formal Specification

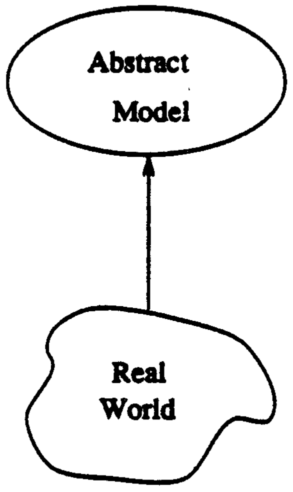

Figure 11: Mapping the Real World to an Abstract Model

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* [64]. 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 [33] 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 clients 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.

A specifier often makes implicit assumptions about a system's environment when specifying something 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 language's 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.

However, when specifying a large, complex, software and/or hardware system, the specifier should take special care to make explicit as many assumptions about the environment as possible. Unfortunately, too often when specifying a large system, specifiers forget to state explicitly the circumstances under which the system is expected to behave properly.

In reality, it is impossible to model formally many environmental aspects such as unpredictable or unanticipated events, human error, and natural catastrophes (lightning, hurricanes, earthquakes). Hazard analysis, as a complementary technique to formal methods, can be used to identify safety-critical components of a system [68]. Formal methods can then be used to describe and reason about these components, where reasoning holds only for those system input parameters that are 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 a specifier 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 developers get the requirements stated "correctly," the more likely they get the design "right," and of course the more precisely they can argue the correctness of the implementation.

In conclusion, 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. In an effort to push against some of the current pragmatic bounds (in contrast to the two theoretical bounds discussed in the previous section), the formal methods community is actively pursuing the following goals:

- 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 so that systems and their components can be reused based on information captured in their specification. General libraries like the Larch Handbook [47] and the Z Mathematical Toolkit [97], and domain-specific ones like that for oscilloscopes [27] are recent examples.
- To integrate formal methods with the entire system development effort, e.g., to provide a formal way to record design rationale in the system development process;
- To demonstrate that existing techniques scale up to handle real-world problems and to scale up the techniques themselves;
- To educate and train more people in the use of formal methods.

Acknowledgments

I thank Susan Gerhart for giving me this opportunity to express my thoughts about formal methods. I am grateful to 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 system development. I thank all those who attended Formal Methods '89 in Halifax, Nova Scotia for helpful feedback and discussion. I would especially like to credit Jim Horning for suggesting this paper's title, a subject of much controversy, and Joseph Goguen who has suggested that formal methods can be given a formal characterization in terms of institutions [39] where semantic abstraction functions are institution morphisms. Finally, I thank Mark Ardis, Dan Craigen, Susan Gerhart, Joseph Goguen, Bob Harper, Jim Horning, Leslie Lamport, and David Parnas for their critical comments on an earlier draft of this paper.

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] A. Aho and J. Ullman. *Principles of Compiler Design*. Addison=Wesley, 1977.

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

[5] K.R. Apt, N. Francez, and W.P. de Roever. A proof system for Communicating Sequential Processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–385, July 1980.

[6] R. Balzer. Transformational implementation: an example. *IEEE TSE*, 7(1):3–14, January 1981.

[7] J. Barwise. Mathematical proofs of computer correctness. *Notices of AMS*, September 1989.

[8] Bauer et al. *The Munich Project CIP, Volume 1: The Wide Spectrum Language CIP-L*. Volume 183 of *Lecture Notes in Computer Science*, Springer-Verlag, 1985.

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

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

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

[12] 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.

[13] M. Broy. A fixed point to applicative multiprogramming. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 565–623, Reidel Publishing Company, 1982.

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

[15] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[16] 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.

[17] R.M. Burstall and J.A. Goguen. The semantics of Clear, a specification language. In *Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332, Springer-Verlag, 1980. Lecture Notes in Computer Science 86.

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

[19] 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.

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

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

[22] B.P. Collins, J.E. Nicholls, and I.H. Sorensen. *Introducing Formal Methods: the CICS Experience with Z*. Technical Report TR 12.260, IBM, United Kingdom Laboratories, Hursley, 1987.

[23] R. Constable et al. *Implementing Mathematics with the NuPRL Proof Development Environment*. Prentice-Hall, 1986.

[24] IBM Corporation. *Customer Information Control System/Virtual Storage, Introduction to Program Logic*. IBM Corporation, sc33-0067-1 edition, June 1978.

[25] D. Craigen, S. Kromodimoeljo, I. Meisels, A. Neilson, B. Pase, and M. Saaltink. M-eves: a tool for verifying software. In *Proceedings of the 10th International Conference on Software Engineering*, pages 324–333, Singapore, April 1988.

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

[27] N. Delisle and D. Garlan. Formally specifying electronic instruments. In *Proc. Fifth Int'l Workshop on Software Specification and Design*, pages 242–248, Pittsburgh, 1989.

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

[29] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

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

[31] H.-D. Ehrich. Extensions and implementations of abstract data type specifications. In *Mathematical Foundations of Computer Science 1978 Proceedings*, pages 155–164, Springer-Verlag, Poland, 1978. Lecture Notes in Computer Science 64.

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

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

[34] S. Fickas. Automating the analysis process: an example. In *Proceedings of the Fourth International Workshop on Software Specification and Design*, pages 79–86, April 1987.

[35] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proceedings of ACM POPL*, pages 52–66, 1985.

[36] 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.

[37] 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.

[38] J.A. Goguen. *OBJ as a Theorem Prover with Applications to Hardware Verification*. Technical Report SRI-CSL-88-4R2, Stanford Research Institute, Menlo Park, CA, August 1988.

[39] J.A. Goguen and R.M. Burstall. Introducing institutions. In *Proceedings of Logics of Programming Workshop*, pages 221–255, Springer-Verlag, 1983. Lecture Notes in Computer Science 164.

[40] 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.

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

[42] 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.

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

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

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

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

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

[48] 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.

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

[50] 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.

[51] 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.

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

[53] C.A.R. Hoare. *Notes on Data Structuring*, pages 83–174. Academic Press, 1972.

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

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

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

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

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

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

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

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

[62] 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.

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

[64] L. Lamport. A simple approach to specifying concurrent systems. *CACM*, 32(1):32–45, January 1989.

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

[66] P. Lee, F. Pfenning, G. Rollins, and W. Scherlis. The Ergo Support System: an integrated set of tools for prototyping integrated environments. In *Proc. Third ACM SIGSOFT Symposium on Software Development Environments*, Boston, MA, November 1988.

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

[68] N.G. Leveson. Software safety: what, why, and how. *ACM Computing Surveys*, 18(2):125–163, June 1986.

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

[70] 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.

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

[72] 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.

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

[74] 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.

[75] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *TOPLAS*, 2(1):90–121, January 1980.

[76] P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, North-Holland, Amsterdam, 1973.

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

[78] B. Meyer. On formalism in specifications. *IEEE Software*, January 1985.

[79] A.J.R.G. Milner. *A Calculus of Communicating Systems.* Volume 92 of *Lecture Notes in Computer Science*, Springer-Verlag, 1980.

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

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

[82] P. Narendran and J. Stillman. Formal verification of the sobel image processing chip. In G. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 92–127, Springer-Verlag, 1989.

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

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

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

[86] D.L. Parnas. *On a 'Buzzword': Hierarchical Structure*, pages 336–339. North-Holland Publishing Company, 1974.

[87] D.L. Parnas. A technique for software module specification with examples. *CACM*, 15(5):330–336, May 1972.

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

[89] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In W.-P. de Roever and G. Rozenberg, editors, *Current Trends in Concurrency: Overviews and Tutorials*, pages 510–584, Springer-Verlag, 1986. Lecture Notes in Computer Science 224.

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

[91] C. Rich, R.C. Waters, and H.B. Reubenstein. Toward a requirements apprentice. In *Proceedings of the Fourth International Workshop on Software Specification and Design*, pages 79–86, April 1987.

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

[93] D.S. Rosenblum and D.C. Luckham. Testing the correctness of tasking supervisors with TSL specifications. In *Proc. ACM SIGSOFT '89 3rd Symposium on Software Testing, Avanlysis, and Verification (TAV-3)*, pages 187–196, Key West, FL, 1989.

[94] D. Sannella and A. Tarlecki. Program specification and development in standard ml. In *Proceedings of the Symposium on Principles of Programming Languages*, 1985.

[95] W.L. Scherlis and D. Scott. First steps toward inferential programming. In *Proceedings of IFIPS '83*, Paris, 1983.

[96] 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.

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

[98] W. Swartout. The Gist behavior explainer. In *Proc. American Association Artificial Intelligence Conf.*, pages 402–407, August 1983.

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

[100] P. Wegner. The Vienna Definition Language. *Computing Surveys*, 4(1):5–63, 1972.

[101] J.M. Wing. Writing Larch interface language specifications. *ACM TOPLAS*, 1–24, January 1987.

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

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

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

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