Reprinted from
Encyclopedia of Software Engineering - 2 Volume Set
ISBN #1-54004-8

FORMAL METHODS

Formal methods used in developing computer systems are mathematically based techniques for describing system properties. Such 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 precisely defining 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 svstem'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 ambiguity, incompleteness, and inconsistency in a system. When used early in the system development process, they can reveal design flaws that otherwise might be discovered only during costly testing and debugging phases. When 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 lacking such considerations would render it useless. Hence, a formal method should possess a set of guidelines, or a *style sheet,* that tells the user the circumstances under which the method can and should be applied as well as how it can be applied most effectively.

One tangible product of applying a formal method is a formal specification. A specification serves as a contract, a valuable piece of documentation, and a means of communication among a client, a specifier, and an implementer. Because of their mathematical basis, formal specifications are more precise and usually more concise than informal ones.

Since a formal method is a method and 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, providing standard syntax analysis tools for formal specifications would be appropriate. 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.

For more on the benefits of formal specification, see Meyer (1985). For more on the distinction between a method and a language, and what specifying a computer system means, see Lamport (1989).

What Is a Specification Language?

A formal specification language provides a formal method's mathematical basis. I borrowed the terms and definitions from Guttag, Horning, and Wing (1982). Burstall and Goguen have used the term *language* and more recently the term *institution* for the notion of a formal specification language.

> *Definition: 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; *and Sat, its* satisfies relation.
> *Definition: 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: 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).*

Less formally, a formal specification language provides a notation (its syntactic domain), a universe of objects (its semantic domain), and a precise rule that defines 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.

Backus-Naur form 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.

Syntactic Domains. We usually define a specification language's syntactic domain in terms of a set of symbols (for example, 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, $\forall x P(x) \Rightarrow tQ(x)$, would be well-formed in predicate logic, but this one, $\forall x \Rightarrow P(x) \Rightarrow Q(x)$, would not 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.

Semantic Domains. Specification languages differ most in their choice of semantic domain. The following are some examples:

- Abstract-data-type specification languages are used to specify algebras, theories, and programs. Though specifications written in these languages range over different semantic domains, they often look syn-tactically similar.
- Concurrent and distributed systems specification languages are used to specify state sequences, event sequences, state and transition sequences, streams, synchronization trees, partial orders, and state machines.
- Programming languages are used to specify functions from input to output, computations, predicate transformers, relations, and machine instructions.

Each programming language (with a well-defined formal semantics) is a specification language, but the reverse is not true, 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 (for example, 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 during requirements specification.

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: 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).*

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

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

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

Definition: Given a specification language, <Syn, Sem, Sat>, and a semantic abstraction function, *A, defined on Sem, an* abstract satisfies relation, *ASat:Syn→2^{Sem}, is the induced relation such that*

$$\forall spec \varepsilon Syn.prog \varepsilon Sem \times [Sat(spec.prog) = ASat(spec.A(prog))]$$

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. Having several specification languages with different semantic abstraction functions for a single semantic domain can be useful. This encourages and supports complementary specifications of different aspects of a system.

For example, in Figure 1, a single semantic domain, Sem, is 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 proce-

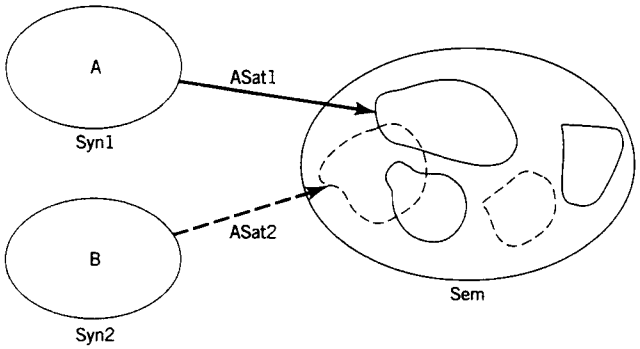

Figure 1. Single semantic domain and a set of equivalent classes.

dures that call functions on a user-defined enumeration type E. Then, a procedure that sorts 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. Behavioral specifications describe only constraints on the observable behavior of specificands. The behavioral constraint that most formal methods address is a system's required functionality (that is, 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, 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 on time.

Structural Specifications. Structural specifications describe constraints on the internal composition of specificands. Example structural specification languages are module interconnection languages. Structural specifications capture various kinds of hierarchical 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.

Properties of Specifications. Each specification language should be defined so each well-formed specification is unambiguous.

Definition: Given a specification language, <Syn, Sem, Sat>, a specification syn in Syn is unambiguous *if and only if 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 incorporating a natural language (like English) is not formal because natural languages are inherently ambiguous. It also means that a visual specification language that permits multiple interpretations of a box and/or arrow is ill-defined, and hence not formal.

Another desirable property of specifications is consistency.

Definition: 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 there is some implementation that will satisfy the specification. If you view a specification as a set of facts, consistency implies that you cannot derive anything contradictory from the specification.

Were you to pose a question based on a consistent specification you would not get mutually exclusive answers. Obviously we want consistent specifications. An inconsistent specification, which negates on one occasion what it asserts on another, means you have no knowledge at all.

Specifications need not be complete in the sense used in mathematical logic, though certain relative-completeness properties might be desirable (for example, sufficient completeness of an algebraic specification).

In practice, you must usually deal with incomplete specifications. Why? Specifiers may intentionally leave some things unspecified, giving the implementer 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 with unfinished products more often than finished ones.

A delicate balance exists between saying just enough and saying too much in a specification. Specifiers want to say enough so that implementers 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 implementer. A specification that overspecifies is guilty of implementation bias (Jones, 1980).

Informally, a specification has implementation bias if it specifies externally unobservable properties of its specificands; 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 toward an ordered-list representation and against a hash table representation.

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 you prove a statement inferable from these facts, you 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 a system's behavior without having to execute or even build it. It gives users a way to state questions, 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 increases user confidence in the specification's validity. If users are 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 (for example, 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.

Pragmatics

Certain pragmatic concerns exist about formal methods, their users, their uses, and their characteristics.

Users. Some users of formal methods are actually going to produce something tangible: formal specifications. However, most people need only read specifications, not develop their own from scratch. Besides specification writers, there are several kinds of specification readers.

In Figure 2, 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 write, evaluate, analyze, and refine spec-ifications. They prove that their refinements preserve certain properties and prove properties of specificands through specifications. Specification readers, besides specifiers, are *customers,* those people who may have hired the specifiers; *implementers,* 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.

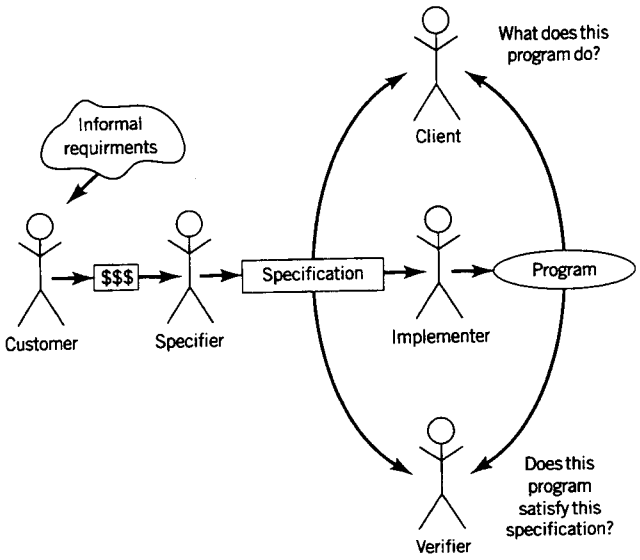

Figure 2. Specification users.

One point of tension in many formal methods is that their languages may be more suitable to one type of specification user than to others. Most language designers will target their language for at least two types of users (for example, clients and specifiers or specifiers and implementers). Some specification languages contain a lot of syntactic "sugar" to make specifications more readable by customers. Some contain 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 math-ematical 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 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 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 the capabilities each should have. 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—for example, digital logic, if specifying hardware, or probability and statistics, if specifying system reliability.

Uses. You can apply formal methods in all phases of system development. Such applications should not be considered a separate activity, but rather an integral one. The greatest benefit in applying a formal method often comes from the process of formalizing rather than from the end result. Gaining a deeper understanding of the specificand by forcing yourself to be abstract yet precise about desired system properties can be more rewarding than having the specification document alone.

Consider, for each system development phase, some uses of formal specifications and the formal methods that support them.

Requirements Analysis. Applying a formal method helps clarify a customer's set of informally stated requirements. A specification helps crystallize the customer's vague ideas and reveals contradictions, ambiguities, and incom-pleteness in the requirements. A specifier has a better chance of asking pertinent questions and evaluating customer responses through the use of a formal, rather than informal, specification. 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 and the Requirements Apprentice address the problem of trans-forming informal requirements into formal specifications; the Gist explainer 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 Development Method (VDM), Z, Larch, and Lamport's transition axiom method 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. At the same time, it provides the module's implementer 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 a place for recording design decisions; moreover, any intentional incompleteness can be captured succinctly 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 a representation type for an abstract data type. Each refinement step requires showing that a specification (or program) at one level satisfies a higher level specification.

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.

Program refinement dates back to Dijkstra's work on stepwise refinement and predicate transformers and Hoare's work on data representation and abstraction functions. Related work on program transformation, program synthesis, and inferential programming have spawned the design of languages like Refine and Extended ML, and programming environments like CIP-S and the Ergo Support System. These refinement approaches are based on classical mathematical logic. An alternative approach to program development based on constructive logic gave rise to proof development environments like Nuprl in which programs are proofs and vice versa.

System Verification. Verification is the process of showing that a system satisfies its specification. Formal verification is impossible without a formal specification. Although you may never completely verify an entire system, you can certainly verify smaller, critical pieces. The trickiest part is in explicitly stating the assumptions about the environment in which each critical piece is placed. Systems such as Gypsy, the Hierarchical Development Method (HDM), the Formal Development Method (FDM), and m-EVES (Environment for Verifying and Evaluating Software) evolved as a result of a primary focus on program verification. Higher Order Logic (HOL) was originally developed for hardware verification.

System Validation. Formal methods can aid in system testing and debugging. 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 for 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 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 testing. Three examples are the Data Abstraction, Implementation, Specification, and Testing System, used to test implementations of abstract data types; Kemmerer's symbolic execution tool, used to venerate and execute test cases from Ina Jo specifications; and the Task Sequencing Language Runtime System, used to automatically check the execution of Ada tasking statements against TSL specifications.

System Documentation. A specification is a description alternative to system implementation. It serves as a communication medium between a client and a specifier, between a specifier and an implementer, and among members of an implementation team. In reply to the question "What does it do?" no answer is more exasperating than "Run it and see." One of the primary intended uses of formal methods is to capture the "what" in a formal specification 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. 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, they can help system developers formulate and answer these questions. The specification serves as a reference point. If the customer is unhappy but the system meets the spec-ification, the specification can be changed and the system changed accordingly.

Indeed, much recent work in the application of formal methods to nontrivial examples has been in specifying a system already built, running, and used. Some of these exercises revealed bugs in published algorithms and circuit designs, serious bugs that had gone undiscovered for years. As expected, most revealed unstated assumptions, inconsistencies, and unintentional incompleteness in the system.

Medium-sized systems that have been specified formally include VLSI circuits, microprocessors, oscil-loscopes, operating systems kernels, distributed databases, and secure systems. Most formal methods have not yet been applied to specifying large-scale software or hardware systems: most are still inadequate to specify many important behavioral constraints beyond functionality, for example, fault-tolerance and real-time performance.

This problem of scale exists in two 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 (such as programs, proofs, and test suites) and their structural interrelationships.

The problem of dealing with a specificand's inherent complexity remains. System complexity results from internal complexity and/or interface complexity. For example, an optimizing compiler is internally more complex than a nonoptimizing one for the same language, yet, in principle, they both provide the same simple interface to their clients (for example, "compile *pro-gram_name*"). By providing a systematic way to think and reason about specificands, formal methods can help people grapple with both kinds of system complexity.

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. This article is not intended to give a complete taxonomy of all possible characteristics of a method nor to classify exhaustively all methods according to these characteristics. Instead, a partial listing of characteristics is given; note that a method typically reflects a combination of many different ones.

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 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.

Model-oriented methods for specifying the behavior of sequential programs and abstract data types include Parnas' state-machines, Robinson and Roubine's extensions to them with V-, O-, and OV-functions, VDM, and Z. Methods for specifying the behavior of concurrent and distributed systems include Petri nets, Milner's Calculus of Communicating Systems, Hoare's Communicating Sequential Processes, Unity, I/O automata, and TSL. The Raise Project represents more recent work on combining VDM and CSP.

Property-oriented methods can be 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, where first-order predicate logic preconditions and post-conditions are used for the specification of each operation of the type. Iota, OBJ, Anna, 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. This approach uses axioms to specify properties of system, but the axioms are restricted to equations. Much work has been done on the algebraic specification of abstract data types, including the handling of error values, nondeterminism, and parameterization. The more widely known specification languages that have evolved from this work are Clear and Act One (Algebraic Specification Techniques for Correct and Trusted Software Systems).

Property-oriented methods for specifying the behavior of concurrent and distributed systems include extensions to the Hoare-axiom method, temporal logic, and Lamport's transition axiom method. The Language of Temporal Ordering of Specifications (LOTOS) specification language represents more recent work on the combination of Act One and CCS (with some CSP influence).

Visual Languages. Visual methods include any whose language contains graphical elements in their syntactic domains. 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 state charts based on higraphs, used to specify state transitions in reactive systems. Figure 3 gives a simple example of a state chart 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. If a message arrives and is 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 state charts 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 out of a system and its subsystems.

Harel's higraph notation inspired the design of the Miró visual languages, which specify security constraints. Like

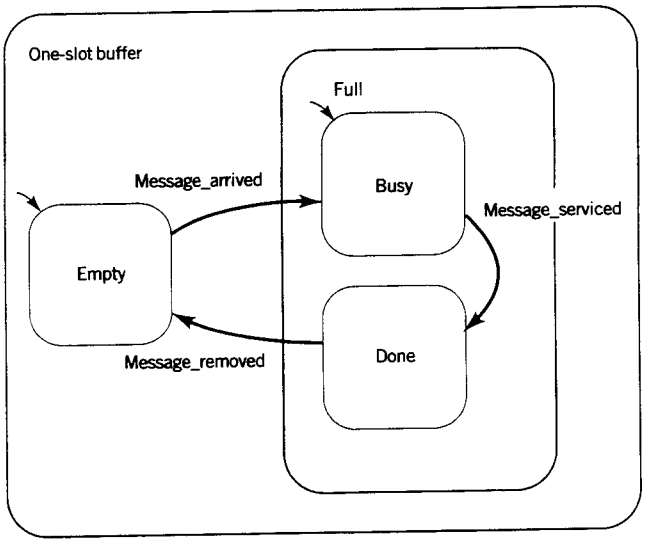

Figure 3. State chart specification of a one-slot buffer.

state charts, the Miró languages have a formally defined semantics and tool support.

Many informal methods use visual notations. These methods allow 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, Hierarchy-Input-Processing-Output (HIPO), structured design, and software requirements engineering methodology are examples of semiformal methods that use pictures.

Executable. Some formal methods support executable specifications, specifications that can run on a computer. An executable specification language is by definition more restricted in expressive power than a nonexecutable 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 specification through symbolic execution of the specification. For example, the Statemate tool lets users run simulations through the state transition diagram represented by a state chart.

Besides state charts, executable specification languages include OBJ; Prolog, a logic programming language that when used in a property-oriented style lets specifiers state logical relations on objects; and Paisley, a model-oriented language, based on a model of event sequences and used to specify functional and timing behavioral constraints for asynchronous parallel processes.

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 or state transition of the system. Tools such as Extended Model Checker (EMC) 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, the Rewrite Rule Laboratory, and the Larch Prover. Tools (and their associated specification languages) that handle subsets of first-order logic include the Boyer-Moore Theorem 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, and OBJ.

Some Examples

This section illustrates six well-known or commonly used formal methods, half applied to one simple example and the other half applied to another example. All six methods have been used to specify much more complex systems.

Sometimes, when specifying the same problem using different methods, the resulting specifications look remarkably similar (as in the first three examples), and sometimes they do not (as in the last three). The similarity or difference is sometimes attributable to the nature 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 (for example, state changes) and not others (for example, error handling). Syntactic conventions (such as indentation style), special notation (vertical and horizontal lines), and keywords affect a specification's physical appearance and its readability.

Most proponents of methods used primarily to specify behavioral properties of concurrent and distributed systems have carefully defined the satisfies relation for a given semantic domain. Many of their methods lack the niceties —the syntactic sugar and software support tools—that formal methods for sequential systems provide. For some theories or models of concurrent and distributed systems, more user-friendly specification languages (LOTOS and Raise) are beginning to appear.

Abstract Data Types: Z, VDM, Larch. Z, a formal method based on set theory, can be used in both model-oriented and property-oriented styles. Figure 4 gives a model-oriented specification of a symbol table, following the Z notation of Spivey (1988). The state of the table is modeled by a partial mapping from keys to values ($X \nrightarrow Y$ 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. I will use the same convention in the VDM and Larch specifications.

The table contains four operations: INIT, INSERT, LOOKUP, and DELETE. INIT initializes the symbol *st* to

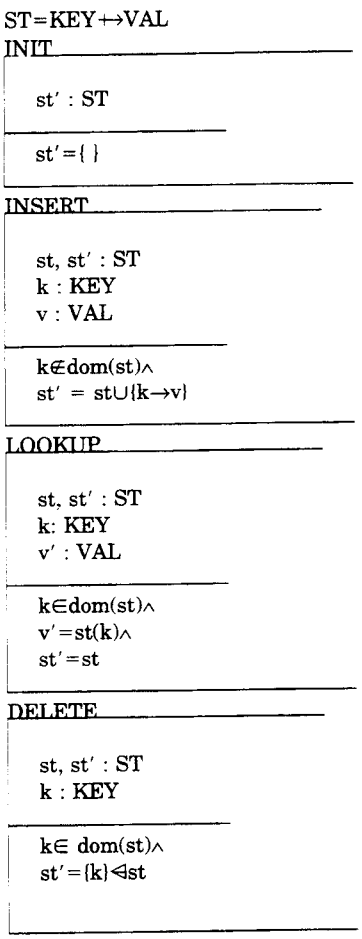

Figure 4. Z specification of a symbol table.

be empty. INSERT modifies the table by adding a new binding to *st,* in the case 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* (⩤ is a domain subtraction operator). The proof checker *B* has been used to prove theorems based on Z specifications.

VDM supports a model-oriented specification style and defines a set of built-in data types (such as 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. I follow the VDM notation given in Jones (1986). The behavior of the INIT, INSERT, LOOKUP, and DELETE operations are the same as specified in the Z specification. However, the preconditions, specified in pre clauses are made explicit and separate from the postconditions, specified in post clauses.

A precondition on an operation is a predicate that must hold in the state on each invocation of the operation, if it does not hold, the operation's behavior is unspecified. A

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

INIT( )
ext wr *st*: *ST*
post *st'* = { }

INSERT (*k*:*Key*,*v* : *Val*)
ext wr *st* : *ST*
pre *k* ∉ **dom** *st*
post *st'* = *st*∪ {*k* ↦ *v*}

LOOKUP(*k* : *KEY*)*v*: *Val*
ext rd *st* : *ST*
pre *k* ∈ **dom** *st*
post *v'* = *st*(*k*)

DELETE(*k* : *Key*)
ext wr *st* : *ST*
pre *k* ∈ **dom** *st*
post *st'* = {*k*}⩤ *st*

Figure 5. VDM specification of a symbol table.

postcondition is a predicate that holds in the state upon return. An operation's clients are responsible for satisfying preconditions, and its implementer is responsible for guaranteeing the postcondition.

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 (Guttag, Horning, and Wing, 1985b). The axiomatic component specifies state-dependent behavior (for example, side effects and exceptional termination) of programs. The algebraic component specifies state-independent properties of data accessed by programs. Figure 6 shows a Larch specification of the symbol table example. The Larch notation given in Guttag, Horning, and Wing (1985a) is used.

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 postconditions. 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.

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 postcondition 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

```
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) V isin(s,k1)

implies
   converts (rem,find,isin)exempting(rem(emp),find(emp))
end SymTab
```

Figure 6. Larch specification of a symbol table.

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 the symbol table does not matter, that is, insertion is commutative.

The **exempting** clause documents the absence of right 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 postconditions of the interface specification; they serve the same role as the built-in symbols like ∪, and ⩤ used in the Z and VDM specifications.

Unlike Z and VDM, Larch does not come with any special built-in notation nor with 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 the exact meaning suitable for the specificand set. Exactly those properties of a data type being specified need 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 I modeled symbol tables in Z and VDM in terms of finite mappings, I did not need to state explicitly that insertion is commutative. This is a property of mappings—the commutative

property came for free. The Larch handbook (Guttag, Horning, and Wing, 1985a) serves as a compromise between the two extremes in that it provides a library of traits that define many general and commonly used concepts (for example, properties of finite mappings, partial orders, sets, and sequences).

Concurrency: Temporal Logic, CSP, Transition Axioms. As mentioned before, many formal methods for specifying the behavior of concurrent and distributed systems differ because of their choice in semantic domain. Some focus on just the states, some on just the events, and some on both. To be more concrete here, I will model a system's behavior as a set of linear sequences of states and associated 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, separating properties of concurrent and distributed systems into two general categories, safety and liveness, is common. 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 concisely state 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 that in all future states, the state predicate P holds. $\Diamond P$ says that in some future state, P will hold; and $\bigcirc P$ says that in the next state P will hold. For example, $P \rightarrow \Diamond Q$ says that if P holds in the current state, Q will eventually 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 presents a temporal logic specification of the behavior of an unbounded buffer in an asynchronous environment. The example is adapted from Koymans (Koymans, Vytopil, and de Roever, 1983), using the temporal logic system in Pnueli (1986), which has 12 modal operators. The formulas are interpreted with respect to sequences of events. A buffer has a left input channel and a right output channel. The expression $<c!m>$ denotes the event of placing message m on channel c. The first predicate.

$$<\text{right}!m> \Rightarrow \Diamond\text{-} <\text{left}!\text{m}>$$

$$<\text{right}!m> =>\Diamond\text{-}<\text{left}!m> \quad (1)$$

$$(<\text{right}!m> \wedge \ominus\Diamond <\text{right}!m'>) =>\Diamond\text{-}(<\text{left}!m> \wedge \ominus\Diamond\text{-}<\text{left}!m'>) \quad (2)$$

$$(<\text{left}!m> \wedge \ominus\Diamond\text{-}<\text{left}!m'> \Rightarrow (\text{m} \neq m') \quad (3)$$

$$(<\text{left}!m>) =>\Diamond\text{-}(<\text{right}!m>) \quad (4)$$

Figure 7. Temporal logic specification of an unbounded buffer.

states that any message transmitted to the right channel ($<\text{right}!m>$) must have been previously placed on the left channel ($\Diamond <\text{left}!m>$). The second predicate,

$$(<\text{right}!\text{m}> \wedge \ominus\Diamond\text{-}(\text{right}!m'>) \Rightarrow \Diamond\text{-}(<\text{left}!m> \wedge \ominus\Diamond\text{-}<\text{left}!m>)$$

states that messages are transmitted first in, first out. If message m placed on the output channel is preceded by some other message m' also on the output channel ($\ominus\Diamond\text{-}<\text{right}!m'>$), there must have been a preceding (the second $\Diamond\text{-}$) event of placing m on the input channel ($<\text{left}!m>$) and, moreover, an even earlier event that placed m' on the input channel ahead of m($\ominus\Diamond\text{-}<\text{left}!m'>$). The third predicate,

$$(<\text{left}!m> \wedge \ominus\Diamond\text{-}<\text{left}!m'>) \Rightarrow (m \neq m')$$

states that all messages are unique. For each message m currently placed on the input channel and for each previously placed message m'($\ominus\Diamond\text{-}<\text{left}!m'>$), m and m' are not equal. This property is not a property of the buffer, but an assumption of 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,

$$(<\text{left}!\text{m}>) \Rightarrow \Diamond(<\text{right}!\text{m}>)$$

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 the event of sending output message m on 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 (Hoare, 1985). 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,

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

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,

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

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

$BUFFER = P_{\langle\rangle}$
where $P_{\langle\rangle} = \text{left}?m \rightarrow P_{\langle m\rangle}$
and $P_{\langle m\rangle \hat{} s} = (\text{left}?n \rightarrow P_{\langle m\rangle \hat{} s \hat{} \langle n\rangle} | \text{right}!m \rightarrow P_s)$

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

Figure 8. CSP program and specification of an unbounded buffer.

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 | 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 by Q, if the first event was y.

In CSP's formalism, BUFFER is a CSP program; you can state and prove properties about the traces it denotes. Using algebraic laws on traces, you 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 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 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 refuse 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. Occam is a programming language derivative of CSP that has been implemented and used on Transputers.

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 (Lamport, 1983). I use his original notation, although Lamport introduced two other notations in a more recent description of his method (Lamport, 1989).

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 $\rightsquigarrow$ 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.

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.

Bounds of Formal Methods

Between the Ideal and Real Worlds. Formal methods are based on mathematics but are not entirely mathematical. Formal methods users must acknowledge the two important boundaries between the mathematical world and the real world.

Users cross the first boundary in codifying 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.

At all times, the formal specification is only a mathematical representation of the customer's requirements. On 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.

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

liveness properties

4. *in*(PUT)$\wedge|buffer|<min \rightsquigarrow$*after*(PUT)
5. *in*(GET)$\wedge|buffer|>0 \rightsquigarrow$*after*(GET)

Figure 9. Transition axiom specification of an unbounded buffer.

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 people 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.

Assumptions about the Environment. Another kind of boundary is often neglected, even by experienced specifiers. It's 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 you can formally model the system (in terms of a specification language's semantic domain), then, if you can formally model the environment, you can formally characterize the interface between a system and its environment. Most formal methods leave the environment's specification (formal or otherwise) outside the system's

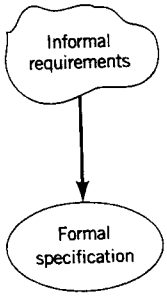

Figure 10. Mapping informal requirements for a formal specification.

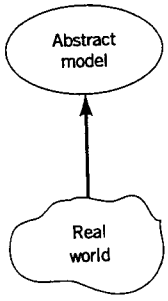

Figure 11. Mapping the real world to an abstract model.

specification. An exception is the Gist language 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:

$$\text{Environment} => \text{System}$$

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, the specifier 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 the environment is expected to satisfy, not a property the buffer is expected to satisfy nor a constraint 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 programming language's invocation protocol. A procedure's specification will typically omit explicit mention of the language's parameter passing mechanism, or, for a compile-time type-checked language, that the argument types 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 doing the appropriate type checking.

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

In reality, it is impossible to formally model 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 identify a system's safety-critical components. 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.

CONCLUSION

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 radically from one another. Within some well-defined mathematical framework, they let system developments couch their ideas precisely. The more rigor applied in system development, the more likely developers are to state requirements correctly and to 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, though not all, deficiencies in a set of informally stated requirements, detect discrepancies between a specification and an implementation, and find errors in existing programs and systems;
- Specify medium-sized and nontrivial problems, especially the functional behavior of sequential programs, abstract data types, and hardware; and
- Provide 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 covered in the previous section), the formal methods community is actively pursuing the following goals:

- Specifying nonfunctional behavior such as reliability, safety, real time, performance, and human factors;
- Combining different methods, such as a domain-specific one with a more general one, or an informal one with a formal one;
- Building 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;
- Building specification libraries so systems and their components can be reused based on information captured in their specification (general libraries, like the Larch handbook (Guttag, Horning, and Wing, 1985a) and the Z mathematical toolkit (Spivey, 1988), and domain-specific ones like that for oscilloscopes, are recent examples);
- Integrating formal methods with the entire system development effort, for example, to provide a formal way to record design rationale in the system development process;
- Demonstrating that existing techniques scale up to handle real-world problems and to scale up the techniques themselves; and
- Educating and training more people in the use of formal methods.

BIBLIOGRAPHY

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), 53–68(1982).

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

J. V. Guttag, J. J. Horning, and J. M. Wing, "The Larch Family of Specification Languages," *IEEE Software* **2**(5), 24–36(1985b).

C. A. R. Hoare, *Communicating Sequential Processes,* Prentice Hall International, Hempstead, UK, 1985.

C. B. Jones, *Software Development: A Rigorous Approach,* Prentice Hall International, Hempstead, UK, 1980.

C. B. Jones, *Systematic Software Development Using VDM,* Prentice Hall International, Hempstead, UK, 1986.

R. Koymans, J. Bytopil, and W. P. de Roever, "Real-Time Programming and Asynchronous Message Passing," *Proceedings of the Second ACM Symposium Principles Distributed Programming,* 1983, pp. 187–197.

L. Lamport, "Specifying Concurrent Program Modules," *ACM Transactions Programming Languages and Systems* **5**(2), 190–222 (1983).

L. Lamport, "A Simple Approach to Specifying Concurrent Systems," *Communications of the ACM* **32**(1) 32–45, (Jan. 1989).

B. Meyer, "On Formalism in Specification," *IEEE Software,* 6–26 (1985).

A. Pnueli, "Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends," in *Current Trends in Concurrency: Overviews and Tutorials,* W.-P. de Roever and G. Rozenberg, eds., lecture notes in *Computer Science* **224,** Springer-Verlag, New York, 1986, pp. 510–584.

J. M. Spivey, *Introducting Z: A Specification Language and its Formal Semantics,* Cambridge University Press, New York, 1988.

JEANNETTE M. WING
Massachusetts Institute of Technology