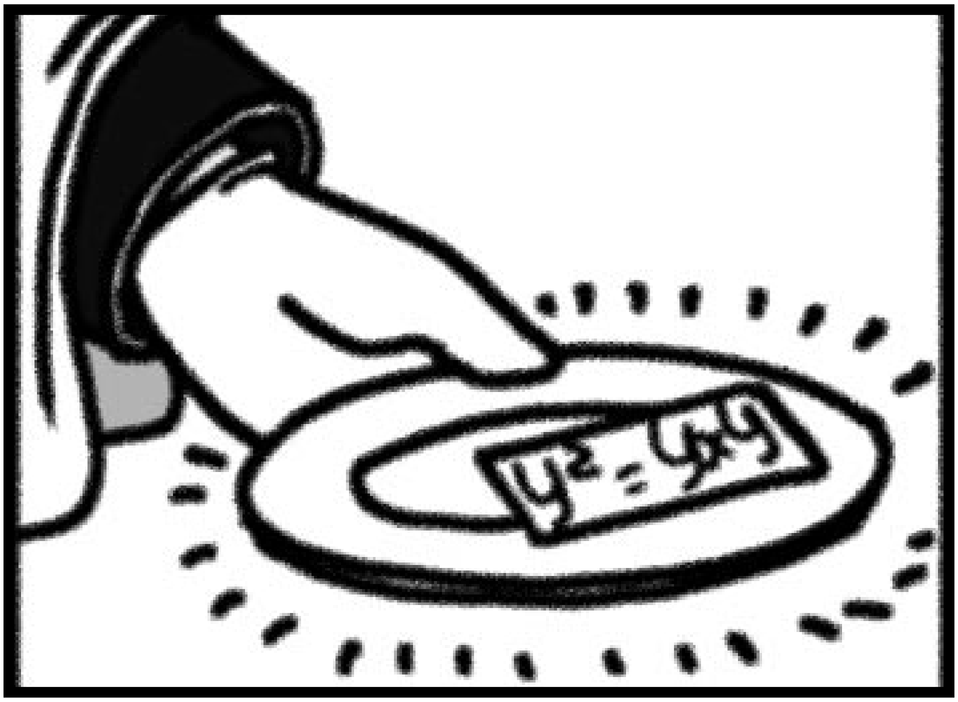
Formal Methods: Promises and Problems

Successfully applying formal methods to software development promises to move us closer to a true engineering discipline. The authors offer suggestions for overcoming the problems that have hindered the use of formal methods thus far.



LUQI
Naval Postgraduate School

JOSEPH A. GOGUEN
University of California
at San Diego

Today's fast-moving technology demands ever quicker and more reliable ways to develop software systems that meet user needs. Although industry spends billions of dollars each year developing software, many software systems fail to satisfy their users. Moreover, many systems once thought adequate no longer are, while others are never finished or never used. The September 1994 issue of *Scientific American* gives some sobering examples and concludes that "despite 50 years of progress, the software industry remains years—perhaps decades—short of the mature engineering discipline needed to meet the demands of an information-age society."[1] Software development failures have reached staggering proportions: an estimated $81 billion was spent on canceled software projects in 1995 and an estimated $100 billion in 1996.[2]

0740-7459/97/$10.00 © 1997 IEEE

Many computer scientists have suggested that formal methods can play a significant role in improving this situation. Although these methods have achieved impressive successes, they have also produced disappointments.

> Requirements for large and complex systems are nearly always problematic initially and evolve throughout the life cycle.

Formal methods do not yet effectively handle large and complex system development, although they can make a contribution. We know that requirements for large and complex systems are nearly always problematic initially and that they evolve continually throughout the life cycle. Thus, any method you use to implement requirements should be flexible and robust, so that it can easily accommodate the inevitable and often continuous stream of changes. We suggest that you can more effectively use formal methods by

♦ putting more emphasis on formal models and on domain-specific formal methods;

♦ using formal models as a basis for computer support of software evolution;

♦ using large-grain software composition methods, rather than small-grain statement-oriented programming methods; and

♦ taking better account of the system development context by tracing objects and relationships back to requirements.

FORMALIZATION

Webster's Dictionary defines *formal* as definite, orderly, and methodical; defines *method* as a regular, orderly, and definite procedure; and defines *model* as a preliminary representation that serves as a plan from which the final and usually larger object is to be constructed. Thus, to be formal does not necessarily require the use of formal logic, or even mathematics. But in computer science, the phrase "formal methods" has acquired a narrower meaning, referring specifically to the use of a formal notation to represent system models during program development. An even narrower sense refers to the formalization of a method for system development. Typically, you first write a specification in a formal notation, then refine it step by step into code. Correctness of the refinement steps guarantees that the code satisfies the specification. In some methods, developers can check correctness of the refinement steps using a theorem prover for the method's underlying formal logic, but other methods remain manual because it is difficult to automate the notation used. To better understand the issues and myths related to the practical usefulness of formal methods, consult "Seven More Myths of Formal Methods,"[3] and for an appraisal of their recent industrial applications, see "An International Survey of Industrial Applications of Formal Methods."[4]

Logical foundation. The prototypical formal notation is first-order logic. This notation has been extensively studied and has inference rule sets known to be sound and complete for a convenient class of models. Unfortunately, mechanical theorem provers for first-order logic can be difficult to work with.

More powerful logical systems can capture additional levels of meaning, but their theorem provers can be even harder to work with. For example, second-order logic can express security requirements for computer systems, but it does not have a sound and complete inference rule set.

Context. Experience shows that many of the most vexing problems in software development arise because any computer system is situated in a particular social context. Moreover, much of the information needed to design a system is embedded in the worlds of users and managers, and is extracted through interaction with these people. This information is informal and highly dependent on its social context for interpretation. On the other hand, we define the programming languages and other representations used to construct computer-based systems using formal syntactic and semantic rules. Both the formal, context-insensitive, and the informal, socially situated aspects of information are crucial for success. These two aspects have been called "the dry" and "the wet," and their reconciliation claimed to be the essence of requirements engineering.[5]

The dry and the wet. That we can make sense out of social life suggests it is somewhat orderly enough to be at least partly formalizable. But it is difficult to formalize domains that have many ad hoc special cases or contain much tacit knowledge or are subject to change. Formalization is more successful on narrow and orderly domains, such as sporting events, which have long traditions, regulating bodies, rule books, referees, and so on. For example, it would be more difficult to formalize a children's game than a regatta, and more difficult still to formalize human political behavior.

There are *degrees* of formalization, ranging from the very formal *dry* to the very informal *wet*. In the driest formalizations, the metalanguage is also formalized, and an object-level model is given as a formal theory in the metalanguage. In less fully formalized models, the metalanguage may be simply a natural language, or a somewhat stylized dialect. There can be rules at both the object and the meta levels. Rules at the object level are part of the model, while rules at the meta level define the language used for formalization. For a given application, it can be a serious error to formalize more than is appropriate to the particular situation.

Formalization is useful only to the extent that it helps meet concrete goals. For example, it would only make it harder to bake cookies if the recipe were expressed in a fully formalized language. There are many similar examples in requirements engineering. Good formalizations do not usually arise top-down from desires, but rather are based on extensive experience and intuition with the domain being formalized and the intended development process.

Formal methods generally address some large class of systems, such as information systems, or even all possible systems, whereas formal models are often tailored to a specific application domain. Experience suggests that using mechanically processable formal models in building and integrating tools can yield systems that increase automation and decrease inconsistency, and thus produce software faster, cheaper, and more reliably. For example, attribute grammars are a formally processable notation that can be useful in this way. Experience also suggests using an evolutionary development process that involves rapid prototyping, such as that supported by the computer-aided prototyping system (CAPS).[6] Such an approach contrasts with formal methods that call for mathematical rigor throughout the development process, usually by using a formal notation with a precise mathematical semantics in connection with a step-by-step refinement process. We believe it makes more sense to provide computer support for software evolution by formalizing the activities of the supporting tools rather than those of the software engineers.

In software engineering, you cannot validate results purely by proving theorems. On the contrary, you must measure the value of a contribution by its impact on practical software development and ultimately on customer satisfaction. But formalization still plays a fundamental role in software engineering, because you must have a formal (in the broad sense) model of a domain before you can design effective software for that domain. That is, problem formalization is an essential part of requirements capture.

In this respect, software engineering differs from other engineering disciplines. For example, in electrical engineering, the formalization of the problem domain is already done, and the practicing engineer need only apply it. The lack of such formalization makes software engineering more difficult than other engineering disciplines, which makes it less developed and less effectively practiced than its cousins.

Unfortunately, like many things in computer science, formal methods have been oversold. Formal methods, notations, and tools do not yet adequately support the development of large and complex systems. In general, practitioners consider formal methods useful for proving that programs satisfy certain mathematical properties, but such methods are also often considered too expensive to be practical. This view ignores evidence that appropriate and correctly used formal methods can reduce time to market, provide better documentation, improve communication, facilitate maintenance, and organize activities throughout the life cycle. Factors that influence the cost-effectiveness of formal methods include the consequences of software failure, the type of formal method to be applied, the availability of automated support for the formal method, and the skill level of available personnel.

SOFTWARE EVOLUTION

Traditionally, many in industry have viewed software evolution as occurring only after the completion of initial development. For example, L.J. Arthur defines software evolution as consisting of “the activities required to keep a software system operational and responsive after it is accepted and placed into production.”[7] This is synonymous with maintenance, but avoids that word’s negative connotations. According to Lawrence Bernstein (formerly of AT&T), evolution emphasizes the dynamic aspect of software development.[8]

Here, we consider software evolution to include all the activities that change a software system, as well as the relationships among those activities. In this case, evolution is not just another name for maintenance, because it occurs throughout the life cycle. Evolution encompasses activities ranging from adjusting requirements to updating working systems, including responses to requirements changes,

The lack of formalization makes software engineering more difficult than other engineering disciplines.

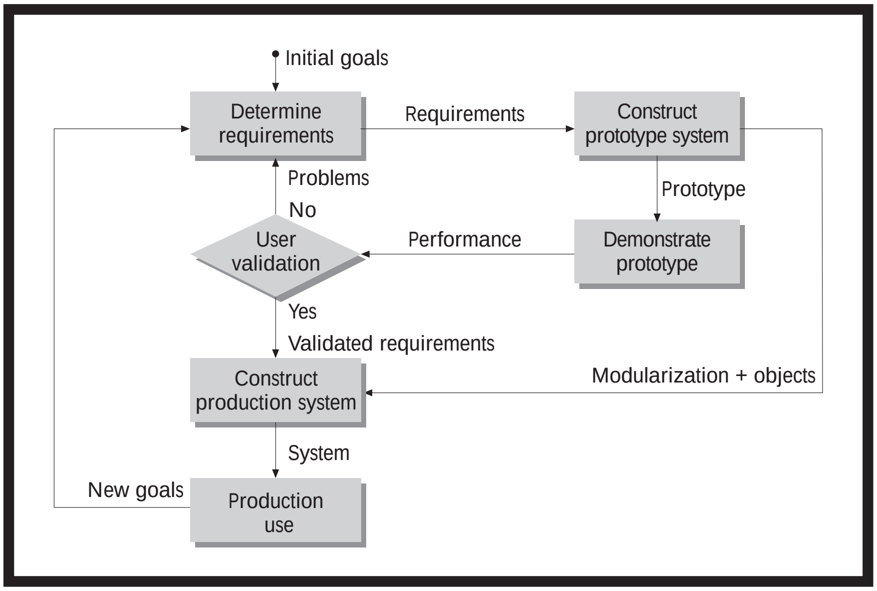


Figure 1. *An iterative prototyping life cycle. Once constructed, the prototype is demonstrated to check its actual behavior against its expected behavior. This helps identify problems and can lead to a redefinition of requirements.*

improvements to performance and clarity, bug repair, version and configuration control, documentation, testing, code generation, and the overall organization of the development process.

The term "evolution" focuses attention on change. Change is inevitable and unending during software development, because it is so difficult to get a system right before it has been tried by actual users under actual operating conditions. Not only do the code and design change, the requirements and the needs that drive the requirements change as well. This occurs partly because users and analysts get a better understanding of what they really need when they see the software operating, but also because the system context changes: laws and regulations change, the competition changes, workers' expectations and habits change, management structure changes, organizational goals change, and so on.

Flexibility through prototyping. Change motivates the use of *iterative* life cycle processes, and in particular, *prototyping:* the process of quickly building and evaluating a series of concrete, executable models of selected aspects of a proposed system. In prototyping, evolution activities are interleaved with development, and continue even after delivery of the system's initial version.[6,9]

This contrasts with traditional life cycles, such as the waterfall model, which assume that requirements can be correctly determined at the beginning of a project. Generally, project staff develop an overall system architecture from the requirements, write specifications and code for individual components, then test and debug the system. Maintenance appears in or after the final phase of testing and debugging.

Figure 1 shows an iterative prototyping life cycle. The user and designer work together to define the requirements for the envisioned system. The designer constructs a prototype at the specification level. Demonstrations of the prototype let the user evaluate the prototype's actual behavior against its expected behavior, identify problems, and work with the designer to redefine requirements. This process continues until the prototype successfully captures the critical aspects of the envisioned system. The designer then uses the validated requirements as a basis for the production software. In this way, software systems can be delivered incrementally and requirements analysis can continue throughout the system's lifetime. Incremental delivery gives users early experience with the software, leading to new goals, triggering further iterations, and extending the advantages of prototyping to the production environment.

Evolution and formal methods. Given the inevitability of change and iteration, formal methods should be more useful in supporting evolution than in their traditional role of verifying that code meets certain fixed requirements. Possible contributions to software evolution include computer-aided design completion, program transformation, dependency maintenance (among needs, requirements, design information, documentation, code, and so on), code generation (for certain limited purposes), and merging changes to programs. These contributions become even more valuable when many programmers work concurrently on a large and complex system.

The difficulties of software evolution often extend beyond the purely technical: Social, political, and cultural factors can be significant and in many projects will dominate development costs.

Nevertheless, formal model-based tools can help maintain a software development project's integrity in many ways, such as scheduling project tasks, monitoring deadlines, tracing reasons for objects and changes, and maintaining dependency relations

A HYPERGRAPH MODEL FOR SOFTWARE EVOLUTION

Because evolution plays a fundamental role in software development, we must understand it better, formalize key aspects, and build suitable tools based on the resulting formal models. This endeavor is still at an early stage. Here we briefly describe a formal model that helps develop tools to manage both the activities in a software development project and the products that those activities produce.

This model simplifies and clarifies previous CAPS models by incorporating some of their features into a more abstract mathematical structure. We intend that the software evolution data model presented here be easier to modify, extend, and understand than earlier models; this should also make it easier to implement. The model represents the evolution history and future plans for software development as a *hypergraph*. Hypergraphs generalize the usual notion of a directed graph by allowing *hyperedges*, which may have multiple output nodes and multiple input nodes. The following mathematical concepts are used in this software evolution model:

Definition 1. A (directed) hypergraph is a tuple (N,E,I,O) where

N is a set of *nodes*,

E is a set of *hyperedges* (sometimes simply called edges),

$I : E \rightarrow 2^N$ is a function giving the set of *inputs* of each hyperedge, and

$O : E \rightarrow 2^N$ is a function giving the set of *outputs* of each hyperedge.

A path p from a node n to a node n' is a sequence $e_1...e_k$ of k>0 edges and a sequence $n_1...n_{k+1}$ of nodes such that $n_i \in I(e_i)$ and $n_{i+1} \in O(e_i)$ for $i = 1,...,k$, where $n = n_1$ and $n' = n_{k+1}$. A hypergraph H is *acyclic* if there is no path from any node in H to itself.

A set N' of nodes is reachable from a set R of nodes if there is a path to each $n' \in N'$ from some $n \in R$. A hypergraph H is reachable from a set R of its nodes if its set N of nodes is reachable from R. A *root* of H is a node from which H is reachable. A *leaf* of H is a node from which no other node is reachable.

If $H = (N,E,I,O)$ is a hypergraph, then its opposite, denoted H^{op}, is the hypergraph (N,E,O,I). We say that H is coreachable from N' if H^{op} is reachable from N'. A hyperpath in a hypergraph $H = (N,E,I,O)$ from $D \subseteq N$ to $T \subseteq N$ is a minimal hypergraph contained in H, whose node set contains D and T, and that is reachable from D and coreachable from T; we call D and T the input and output sets of the hyperpath, respectively.

In the hypergraph software evolution data model each node represents a software component, which is an immutable version of a software object. Edges record dependencies among various versions of software objects in the system and represent the evolution steps (development activities that create the output objects of the edge). These software objects can be of many different kinds, including problem reports, change requests, reactions to prototype demonstrations, requirements, specifications, manuals, test data, design documents, and many other kinds of object besides program code. The dependencies represent the essence of the derivation history, as well as plans for future evolution.

Definition 2. An evolutionary hypergraph is a hypergraph $H = (N,E,I,O)$ together with functions $L_N : N \rightarrow C$ and $L_E : E \rightarrow A$ such that the following assumptions are satisfied:

N and E are disjoint subsets of a set U whose elements are called *unique identifiers*;

if $O(e) \cap O(e') \neq \varnothing$ then $e = e'$; we call this the *identifiability* condition;

H is acyclic; and

A={s, d} · A (that is, each element of A has the form (S, a′) or (d, a′), where a′ ∈ A′.

An edge labeled "s" is called a step and one labeled "d" is called a decomposition.

The elements of N are identifiers for software components, the elements of E are identifiers for evolution steps, and I and O give the inputs and outputs of each evolution step. The function L_N labels each node with component attributes from the set C, including the corresponding version of the software object, and the function L_E labels each edge with step attributes from the set A, including the current status of the step. The notion of component used here includes components in the usual sense as well as systems built by combining subcomponents, test cases, bug reports, and other kinds of software objects. Decomposition edges include the part_of relation in earlier versions of this model.

The first condition says that the node and edge identifiers are distinct. The second says that the output sets of different evolution steps are disjoint; this implies that each step is uniquely identifiable by any component that it produces, so that the producing step can be considered an attribute of a component. The third condition implies that the process of software evolution never brings us back to a component we have already built; this simply means that we never reuse a unique identifier for a component. However, it is certainly possible that a later version of a component is equal to an earlier one, in the sense that $L(n) = L(n')$ where $n \neq n'$ and n' depends on n, in a sense made precise by the following:

A node n' depends on a node n if there is a path from n to n'. Similarly, a node n depends on a step s if there is a path to n involving s. A step s' depends on a step s if there is a path involving both s and s' with s earlier in the path than s'. We may say that a component c' depends on a component c if there is a path from n to n' such that $c = L(n)$ and $c' = L(n')$.

The model developed so far does not include the idea that some evolution steps may be composites of other, lower level steps. To model this, we introduce a hierarchical structure on the hyperedges in a hypergraph. This also has the advantage of permitting overviews of the evolution history at various levels of detail.

Definition 3. An *(edge) hierarchical hypergraph* is an acyclic graph with nodes labeled by hypergraphs, such that: the graph has just one leaf and one root; each of its edges corresponds to an edge expansion of a single hyperedge in its source hypergraph, the result of which is the hypergraph in its target; and the result of the composite expansions along

Continued on page 78

Continued from page 77

any two paths between the same two nodes are equal. A *hierarchical evolutionary hypergraph* is a hierarchical hypergraph whose nodes are labeled by evolutionary hypergraphs and whose edges are labeled by the step that is expanded.

The intuition behind this definition is that the root node hypergraph is the most abstract top-level view of the system's evolution history and structure, while the leaf node is the fully expanded form. The nodes of the root hypergraph are different versions of the entire software product, while the nodes of the leaf hypergraph include the versions of the atomic software objects that constitute the software product. All of the steps in the leaf hypergraph are atomic. An edge expansion in a hypergraph replaces a hyperedge by a hypergraph containing the original nodes; we leave this technically complex notion informal and illustrate it in the following example. We do not intend that the evolution data should be represented as described in the preceding definition; rather, we intend it as an abstract *model* against which efficient implementations can be tested.

The top level of Figure A (1) contains three versions of a software system, labeled P1.1, P1.2, and P2.2. Step s1 derives a new version P1.2 from P1.1, and step s2 derives another variant P2.2 from P1.1. This is a simple evolutionary hypergraph, which is just an ordinary graph. The lower level in Figure A (1) shows the expansion of the edge s2. The decomposition edges d1.1 and d2.2 show that A1.1 and B1.1 are parts of P1.1, and that A2.2, B1.1, and C2.1 are parts of P2.2. The substep s2.a2 derives A2.2 from A1.1, while the substep s2.c1 derives the new component C2.1 from nothing at all. In a more complete example, C2.1 might be derived from a new requirement.

These two descriptions of the system's history and structure correspond to the hypergraphs labeled H0 and H2 in Figure A (2); the edge from H0 to H2 represents the expansion of the evolution step s2. If we also expand the step s1, corresponding to the edge from H0 to H1, then we must also have a fourth hypergraph H12, containing both edge expansions. The complete evolutionary hypergraph has the diamond shape shown in Figure A (2), in which H0 is the root and H12 the leaf. This model can be improved and extended in many ways, such as by imposing more structure on the set *U* of unique identifiers to define the concepts of version and variant; we have already used such a structure informally in the example.

Our intention has been to abstract away as much detail as possible while still showing the basic concepts. For example, the decomposition of the abstract set A as {s,d} X A′ introduces structure that conveys further information about evolution; by further decomposing A′, even more information can be represented. For example, steps can have attributes and relationships to reflect management decisions, such as deadlines, priorities, and the designers involved.

The evolution control system,[1] based on an earlier version of this model, provides algorithms using the information in the graph to support several different kinds of automation. This support includes first approximations to the decomposition structure of a step derived from the decomposition structure of the current version of the affected components and induced steps implied by dependencies between components. For example, if a requirement is modified then the program components derived from that requirement must also be modified.

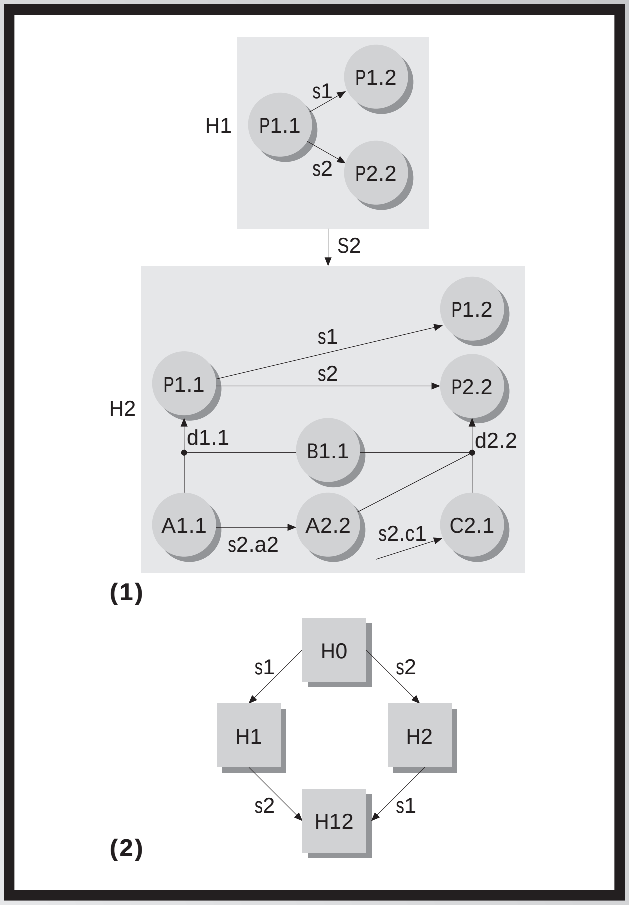


Figure A. *Hierarchical Evolutionary Hypergraph. The topmost part of (1) shows three versions of a software system; the bottom part of (1) shows the expansion of the edge s2. The complete evolutionary hypergraph appears in (2).*

The tool based on such models also has scheduling algorithms that provide estimated completion times for project activities and alerts when project deadlines are affected. The schedule is adjusted as more information becomes available, using the management policies recorded as attributes of steps to automatically assign new tasks to designers as they complete previous tasks. The system also uses the dependency information in project plans to deliver the proper versions of the components needed to carry out each step and to insert the versions of components produced by completed steps in the proper places in the graph. This automates check-in and check-out from the project database.

REFERENCE

1. S. Badr, *A Model and Algorithms for a Software Evolution Control System,* doctoral dissertation, Naval Postgraduate School, Monterey, Calif., 1993.

among versions, variations, and component decompositions. The hypergraph model described in the box on pages 77 and 78 is designed to support such activities.

The US Department of Defense issued MIL-STD-498 in 1994.[10] This standard has evolved into ISO/IEC 12207, which will be adopted as joint standard J-STD-016. These software development standards will have a profound effect on software evolution. They replace several previous standards that mandated the waterfall model, thus creating new opportunities by allowing considerably greater flexibility. But it also requires greater skill levels because it must be tailored to specific projects and organizations. Large and complex software development projects based on specially tailored standards will be difficult to manage without appropriate tool support, and it will be difficult to develop such tools without appropriate formal models. Contrary to typical assumptions in work on software processes, we believe that such models should focus on what the tools do rather than on what the personnel involved do.

SCALABILITY FOR APPLICATIONS

Our analysis of formal methods distinguishes between small-, large-, and huge-grain methods, referring to the size of the atomic parts used, rather than to the size of the system being developed. We reluctantly chose the word *huge* as the next step above *large*, because *large* is already in common use in certain communities; it would have been better if the three steps were instead called fine, medium, and coarse.

Small-grain methods. The classic formal methods fall into the small-grain category. These methods have a mathematical basis at the level of individual statements and small programs, but rapidly hit a complexity barrier when programs get large. In particular, systems for reasoning with pre- and postconditions—such as Hoare axioms, weakest preconditions, predicate transformers, and transformational programming—all have small-size atomic units and fail to scale up because they do not provide structuring or encapsulation. In general, small-grain methods have great difficulty handling changes, and thus fit poorly into the life cycle. Transformational programming is less resistant to change than other small-grain methods, but has the problem that in general there is no bound to the number of transformations that may be needed; this restricts its use to relatively small and well-understood domains.

Large-grain methods. The most important techniques of large-grain programming involve module composition. The CAPS system[11] provides module composition for rapid prototyping, with a dataflow-like semantics that supports hard real-time constraints and with facilities for retrieving reusable software components from a repository. The project is also working on the foundations of software maintenance and developing techniques to support design evolution, requirements tracing, configuration management, and project management. One of these techniques, change merging, has the potential to aid in combining concurrent changes to the same base version of a prototype as well as updating multiple versions of a prototype with a common change.[12]

CAPS consists of an integrated tool set that helps you design, translate, and execute prototypes. These include an evolution control system based on a graph model for evolution, a change merge facility, automatic generators for schedule and control code, and automated retrievers for reusable components.

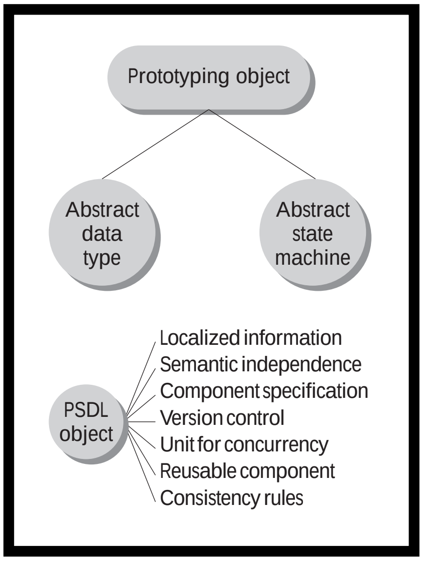


Figure 2. *Class structure and properties of PSDL objects. This prototype system description language provides a simple way of specifying software systems for prototypes and production software.*

The prototype system description language PSDL[11] provides a simple way to abstractly specify software systems for both prototypes and production software. A PSDL program consists of two kinds of objects, corresponding to abstract data types (PSDL types) and abstract state machines (PSDL operators) as shown in Figure 2. Their function is to localize the information for analyzing, executing, and reusing independent objects. They are also the basis for version control and are natural units of work in a distributed implementation.

When an executable Ada module is associated with each atomic PSDL object, CAPS can automatically generate "glue code" that composes these modules into a system having the structure described by the dataflow diagram. This code includes a generated schedule and tests for all the real-time constraints that have been declared; these components can be used to check the design assumptions on which the schedule is based. The system can then be compiled, executed, and tested. Error messages are produced during execution if constraints are violated.

Parameterized programming lets you express designs and system properties in a modular way.

Huge-grain methods. Huge-grain parts are much larger than small-grain statements and large-grain modules. Huge-grain parts may be systems themselves, typically commercial off-the-shelf systems. Developing systems using huge-grain parts is qualitatively different from working with small- and large-grain parts. In particular, correcting some errors in a huge-grain part may be impossible, in which case they must be accepted and worked around. For example, a network protocol such as TCP/IP may have been obtained from an external vendor, so the developers of the larger system do not have access to its source code. If the version being used has a bug, there is no choice but to find a way to avoid that bug. This is often possible because of the multiplicity of features provided in such parts. Specification and requirement methods for huge-grain systems must be robust, effective, easy to learn, and easy to incorporate into the life cycle.

Technology developed for large-grain system development can also be useful for the huge-grain case, since huge-grain parts can often be treated as modules. For example, PSDL's control constraints, such as execution guards and output guards, support adjustments to the behavior of huge-grain parts without access to their source code. The wrapper concept is also relevant. Huge-grain methods are an important area for further research.

Parameterized programming. The object-oriented version of the *parameterized programming*[13] approach is another example of a large-grain method. It uses module expressions, theories, and views to compose systems from subsystems. It distinguishes among sorts for values, classes for objects, and modules for encapsulation. Parameterized programming lets you express designs and high-level system properties in a modular way, and lets you parameterize, compose, and reuse designs, specifications, and code as well.

In this approach, the main programming unit is the *module*, which lets you declare multiple classes together. Module composition features include summing, renaming, enhancing, modifying, parameterizing, instantiating, and importing. The *sum* of modules is a kind of parallel composition that takes account of sharing. *Renaming* lets you assign new names to the sorts, classes, attributes, and methods of modules; *enhancing* lets you add functionality to a module; and *modifying* lets you redefine some of its units.

Parameterized programming was first implemented in the OBJ language, and has also been implemented in the Functional Object-Oriented Programming System (FOOPS) and Eqlog languages. It has a rigorous semantics based on category theory. Much of the advantage of parameterized programming comes from the ability to parameterize modules using theories and views; for example, a higher-order capability can be provided in a purely first-order setting.

Parameterized programming supports *design* in the same framework as specification and coding. Designs are expressed as module expressions and can be executed if specifications that have a suitable form are available. This gives a convenient form of prototyping. Alternatively, prototypes for the modules involved can be composed to give a system prototype by evaluating the module expression for the design. A novel feature of the approach is to distinguish between structuring, genericity, and compositionality in horizontal and vertical modes. *Vertical structure* relates to layers of abstraction, in which lower layers implement or support higher layers. *Horizontal structure* is concerned with module aggregation, enrichment, and specialization. Both kinds of structure can appear in module expressions and both are evaluated when a module expression is evaluated. The approach can also support relatively efficient prototyping through *built-in* modules, which can be composed just like other modules, and which offer a way to combine prototypes with efficient programs in a standard programming language. This is similar to the CAPS approach.

The module and type systems of parameterized programming are considerably more general than those of languages like Ada, Clu, and Modula-3, which provide only limited support for module composition. For example, interfaces in these languages can only express syntactic restrictions on actual arguments, cannot be horizontally structured, and cannot be reused. Lileanna[14] implements many ideas of parameterized programming for the Ada language, including horizontal and vertical composition, following the design of the LIL (library interconnection language) system.[13]

DOMAIN-SPECIFIC FORMAL METHODS

There is much more to formal methods than suggested by the themes dominant in the past, namely synthesis and correctness proofs for algorithms. Although both of these remain interesting topics for theoretical research, their direct impact on the practice of large-scale software development is limited.

Several recent, successful applications of formal methods seem to form a cluster suggesting a new paradigm for applying formal methods. These applications involve a tool having all or most of the following attributes:

- A narrow, well-defined, and well-understood problem domain is addressed, which may have an existing, successful library of program modules.
- There is a coherent user community interested in the problem domain; the users have a good understanding of the domain, good communication among themselves, a standard terminology, and access to financial resources.
- The tool has a graphical user interface that is intuitive to the user community, embodying that community's own language and conventions.
- The tool takes a large-grain approach: rather than synthesizing procedures out of statements, it synthesizes systems out of modules; it may use a library of components and synthesize code for putting them together.
- Inside the tool is a powerful engine that encapsulates formal methods concepts and/or algorithms: it may be a theorem prover or a code generator; users do not have to know how it works, or even that it is there.

We suggest the name *domain-specific formal methods* for this emerging paradigm, in recognition of the role played by the user community and their specific domain. Some systems that fall under this heading include

- Amphion, which combines programs for astronomy calculations,[15]
- CAPS for real-time programming,[8] and
- Panel for multimedia animation.[16]

This paradigm falls into the category of large-grain methods and can potentially be extended to huge-grain problems. The development of domain-specific formal methods should enable our discipline to replace the current practice of inventing new formal models with the more efficient practice of refining and recombining existing application models within supported domains.

This suggests a vision for the future that is less ambitious and more realistic than that of the past. It calls for using formal models and algorithms as a basis for creating computer tools to help solve practical problems that are more limited and well defined than in the past. This vision replaces the unrealistic artificial-intelligence goals of fully automatic software synthesis and verification with the recognition that human understanding and creativity must play an important role and that automated decision support can effectively enhance human capabilities. It also recognizes that requirements changes are a dominant aspect of practical software development that relies on automated tools to make software easier to change.

LIMITS AND PROBLEMS

Despite their many potential benefits, formal methods are not a panacea. We have identified nine specific problems with them[17]:

- Formal notation is alien to most practicing programmers, who have little training or skill in higher mathematics. Also, supporting tools are often insufficiently automated or lack user interfaces suitable for engineers.
- Formal methods papers and training often consider only toy examples taken from existing literature. Although it may be impossible to give a detailed treatment of a realistic example in a research paper or in the classroom, such examples must exist for a method to have credibility. Effective training in formal methods should treat parts of a realistic, nontrivial application.
- Many of the most popular formal methods do not scale up to practical-size problems. The gap between specifications and code is still great. Despite serious and long-term efforts in type theory, weakest preconditions, transformational programming, and so on, coding remains largely manual.
- Some advocates of formal methods dogmatically insist that everything must be proved to the highest possible degree of mathematical rigor. At the least, they argue, it must be machine-checked by a program that allows no errors or gaps, and it should be produced by a machine as well. However, mathematicians rarely achieve or even strive for such rigor; published proofs in mathematics are highly informal and often have small errors. Mathematicians never explicitly mention rules of inference from logic unless they are proving something *about* such rules. The highest levels of formality can be very expensive, and are only warranted for a system's critical aspects.

The highest levels of formality can be very expensive and are only warranted for a system's critical aspects.

- Formal methods tend to be rigid and inflexible. In particular, it is difficult to adapt a formal proof of one statement to prove another, slightly different statement. Since requirements and specifications are constantly changing in the real world, such adaptations are frequently necessary. But classical formal methods have great difficulty in dealing with such changes; their proofs are a discontinuous function of how problems are formulated.

♦ Important aspects of practical software evolution are often ignored. In particular, it is difficult to integrate formal methods into existing software processes. A related difficulty is that when you use multiple methods together you may not be able to integrate their underlying models in a way that supports building a practical software development system to support the methods.

♦ Often vendors do not use the best technology or even understand software development very well; they tend to be interested in profits above all and to have little time for learning either new technologies or their benefits. They often use brute-force methods to speed up projects, forgetting that this can be a shortcut to disaster.

♦ Some formal methods have technical difficulties. A technical deficiency of many small-grain formal methods is that first-order logic is inadequate for expressing the weakest precondition of a loop, as noted in the late 1960s by the logician Erwin Engeler.[18] For example, the weakest precondition for a theorem prover for first-order logic with arithmetic cannot itself be expressed in first-order logic (the postcondition is that the input is a tautology). However, a second-order formulation is adequate, and has been used by us for some years in teaching and research, including the SPEC language used at the Naval Postgraduate School.[9,19]

♦ Finally, certain fundamental limitations are imposed because all formalizations are situated in a certain context. In particular, formalizations are *emergent* in that they are always constructed and interpreted in a context. Formalizations are *contingent* in that their construction and interpretation depend upon details of the context in which this construction or interpretation actually occurs; these details may include interpretations of prior events. Moreover, interpretations are subject to negotiations among interested parties. Formalizations are *open* in that they can always be revised in the light of further analyses. They are also *vague* in that their interpretation is only elaborated to the extent that it is practically useful to do so; the rest is left as tacit knowledge. Further discussion of these points may be found elsewhere,[5] including a general introduction to the social aspects of requirements engineering.

These limits imply that both human effort and context necessarily play a fundamental role whenever formalizations are created, interpreted, or updated. Furthermore, much of the context of that information may be social, such as goals, responsibilities, and needs associated with particular roles in an organization. These considerations are significant for designing tools to support software development. In particular, as an aid to future modifications it is highly desirable to make contextual information available along with specifications and code. The lack of such information is what makes redesign difficult and what motivates current research on reengineering. Clearly, it would be better if such information were systematically recorded in the first place.

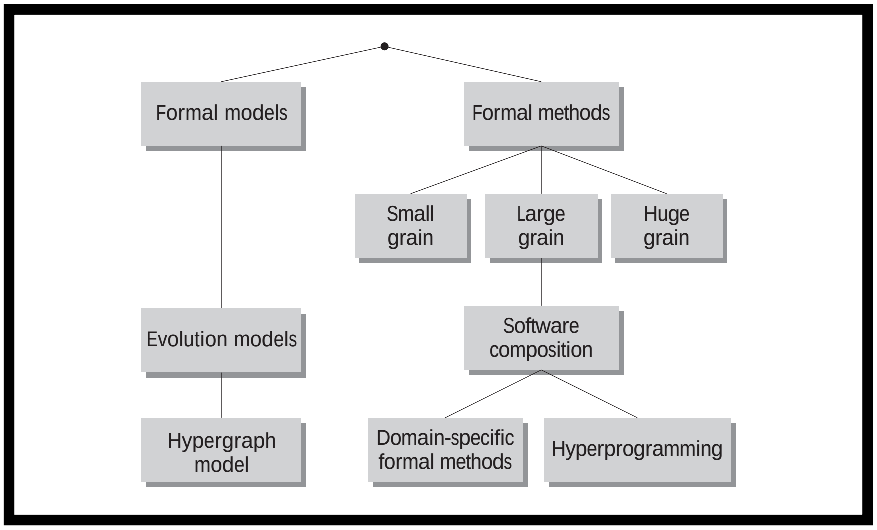


Figure 3. *Classification of some formal methods concepts. Any concept connected by a line to a higher concept is a subclass of that concept.*

LESSONS LEARNED

We have taken a broad view of formalization's role in the software development process and have considered the role of formal methods within that context. In particular, we found that you must understand software evolution to understand the promises and problems of formal methods, that evolution is inevitable and unending in software development, and that much of the pressure for change arises from the social context of system development. Since formal methods tend to be brittle or discontinuous—a small change in

HYPERREQUIREMENTS

The requirements phase of a large-system development project is the most error-prone, and requirements errors are also the most expensive to correct. Therefore improvements here will have the greatest economic leverage. Unfortunately, requirements are one of the least-explored areas of software engineering. There is lively debate about even the definition and scope of the term “requirements.” Ian Sommerville defines requirements capture and analysis as “the process of establishing the services the system should provide and the constraints under which it must operate.”[1] Al Davis suggests that requirements engineering is the analysis, documentation, and ongoing evolution of both user needs and the external behavior of the system to be built.[2] Goguen believes that requirements are properties that a system should have to succeed in its implementation environment.[3] These properties refer to the system’s context of use and thus to social as well as technical factors.

Since many large software projects fail because of social, political, or cultural factors, we must take into account the social context of computer-based systems, in addition to the usual technical factors. Social context is especially important for requirements.

Traceability. We have undertaken several projects on improving the acquisition, traceability, accessibility, modularity, and reusability of the many objects that arise and are manipulated during software development, with a particular focus on the role of requirements.[3] One study administered a detailed two-stage questionnaire to software engineers at a large UK telecommunications firm. The questionnaire revealed that traceability was a major concern that consisted of several different problems that would best be treated in different ways.

Major distinctions appeared between the traceability of prerequirements specification and that of postrequirements specification and between forward and backward traceability. Analysis also showed that access to users was a common difficulty that prevented acquiring necessary information. Further investigation revealed certain policies and traditions that restrict communication within this firm, so that requirements engineers often could not discover what users really needed. One problem was an internal market that restricted communication between “vendors” and “clients” within the firm. Various political considerations also played an important role. Abolition of the internal market for requirements projects, and generally improving the openness of information, could potentially save enormous sums for firms like the one studied.

Tracing dependencies. Another aspect of the traceability problem is the difficulty of maintaining the huge mass of dependencies among the many objects produced by a large software system development effort. Often these objects are not adequately defined; for example, module boundaries may be incorrectly drawn or not even explicitly declared at all and interfaces may be poorly drawn or badly documented. With-out using representations for the objects involved, formal models for the dependencies, and tool support for managing them, it is impossible to know what effect a change will have, and in particular, to know what other objects may have to be changed to maintain consistency.

To meet this challenge, the TOOR system (for Tracing Object-Oriented Requirements) was developed.[4] It is a flexible, user-configurable object-oriented system that supports,

♦ links among objects representing user-definable relationships,

♦ grounding decisions in the prior objects that justify them, and

♦ tracing dependencies.

Particular challenges include formalizing dependencies and developing methods for calculating dependencies and propagating the implications of changes. This approach, called *hyperrequirements*, builds on earlier work on *hyperprogramming*, and is intended to support, by linking related objects, both the social context of requirements decisions and their traceability. Parameterized programming will support reuse, and the generalized notion of relation will support links among design, coding, and maintenance. Other work at Oxford is exploring the use of novel methods from sociology such as ethnomethodology, and the use of situated abstract data types, a new concept that helps bridge the gap between computer technology and its social context.[3]

REFERENCES

1. I. Sommerville, *Software Engineering*, Addison-Wesley, Reading, Mass., 1989.
2. A. Davis, *Software Requirements: Analysis & Specification*, Prentice-Hall, Englewood Cliffs, N.J., 1990.
3. J. Goguen, “Requirements Engineering as the Reconciliation of Social and Technical Issues,” *Requirements Engineering: Social and Technical Issues*, M. Jirotka and J. Goguen, eds., Academic Press, London, 1994, pp. 165–200.
4. F. Pinhero and J. Goguen, “An Object-Oriented Tool for Tracing Requirements,” *IEEE Software*, Mar. 1996, pp. 52-64.

the domain can require a great deal of new work—automation is often vital for their practical application.

The construction, interpretation, and updating of formalism is always situated in a context, and understanding that context can be important in capturing the requirements for large and complex systems. Some fundamental formalization limits arise in this way. Modularity and reuse can help with any approach to improving the quality and reducing the cost of software development. We need traceability to get better control of the software life cycle.

We need traceability to get better control of the software life cycle.

Figure 3 shows the relationships between some of the concepts we've described. A line indicates that the lower concept is a subclass of the higher one.

Building a brighter future. Whatever we learn about software development should be appropriately formalized, implemented, and put into computer science curricula so that future generations can do better than we have. Teaching a formal method while ignoring its use in real projects can have a highly negative impact. For example, students may be taught programming from formal specifications, but not that specifications come from requirements, and that requirements are always changing, often because of social, political, and cultural factors.

As a result, students are not prepared for the rapid change and political problems found in real industrial work. Many students also feel that formal methods turn programming from a creative activity into a boring, formal exercise. We have seen cases in which students have left the discipline because teachers have failed to deal with these problems.

Students need to know how to deal with real programs that have thousands or even millions of lines of code. Most of the examples used in textbooks and the classroom are very small, however, and carefully crafted correctness proofs of simple algorithms give an entirely misleading impression of what real programming is like. Also, most of the techniques taught are small-grain and thus do not scale up to large and complex problems.

Reliable tools based on a formal model can let students do problems that would be impossible by hand. Teachers should also present methods and tools that work on large-grain units—modules—rather than on small-grain units—statements, functions, and procedures—because large-grain methods can scale up, whereas small-grain methods cannot. Suites of sample problems should be developed that systematically show how and when to apply formal methods, and how to combine them with informal approaches. This will require developing appropriate module collections, refining and extending existing formal methods and tools, developing more natural user interfaces, re-thinking process models, revising curricula, retraining teachers, and experimentally validating the resulting methods in practical situations.

If we fail to properly train the next generation of software developers, the problems that we see today will worsen as the size and complexity of systems continue to grow and the dead weight of legacy code continues to mount.

There is no doubt that formal models and methods can be very useful in practical software development. It also seems clear that they are necessary for transforming software engineering into a discipline that is as well understood and well organized as other engineering disciplines, which rely on sound and well-tested mathematical models. The difficulty is that formalization itself plays a more basic role in software engineering than in other engineering disciplines. Because software is still actively expanding into completely new application domains, and because requirements capture is a process of formalization, software development requires the construction of new formal models for each new application, as well as using established formal models.

More emphasis should be placed on *context* in system development and on domain-specific formal methods. However, basic research in computational logic still provides the foundation for many practical applications of formal models and methods, and advances in this area will increase the amount of computer support that can be provided in practice. A short-term view of what technology needs should be avoided, as should overselling formal methods, either as a general field or as an approach to particular applications.

With these caveats, formal methods and formal models should play an increasingly important part in coming to grips with the ongoing crisis engendered by our escalating expectations about the size, complexity, and reliability of software systems. ◆

ACKNOWLEDGMENTS

The research reported in this article has been supported in part by the National Science Foundation under grant number CCR-9058453, the Army Research Office under grant number ARO-145-91, British Telecommunications plc, the European Community under ESPRIT-2 BRA Working Group 6071, IS-CORE (Information Systems COrrectness and REusability), Fujitsu Laboratories Ltd., and a contract under the management of the Information Technology Promotion Agency (IPA), Japan, as part of the Industrial Science and Technology Frontier program "New Models for Software Architectures," sponsored by the New Energy and Industrial Technology Development Organization. We thank Valdis Berzins and David Dampier for their valuable comments on a draft of this article. We also thank Shari Pfleeger for her inspiration and support.

REFERENCES

1. W. Gibbs, "Software's Chronic Crisis," *Scientific American*, Sept. 1994, pp. 86-95.
2. Chaos 97, tech. report, Standish Group Int'l, Dennis, Mass., to appear Jan. 1997 at http://www.standishgroup.com/chaos.html.
3. J. Bowen and M. Hinchley, "Seven More Myths of Formal Methods," *IEEE Software*, July 1995, pp. 34–41.
4. D. Craigen, S. Gerhart, and T. Ralston, "An International Survey of Industrial Applications of Formal Methods," tech. report TR GCR 93/626, US Nat'l Inst. of Standards and Technology, Washington, D.C., 1993.
5. J. Goguen, "Requirements Engineering as the Reconciliation of Social and Technical Issues," *Requirements Engineering: Social and Technical Issues*, M. Jirotka and J. Goguen, eds., Academic Press, London, 1994, pp. 165–200.
6. Luqi, "Software Evolution through Rapid Prototyping," *Computer*, May 1989, pp. 13–25.
7. L.J. Arthur, *Software Evolution: The Software Maintenance Challenge*, Wiley Interscience, New York, 1988.
8. L. Bernstein, "Importance of Software Prototyping," *J. Systems Integration Special Issue: Computer-Aided Prototyping*, Vol. 6, Nos. 1-2, Mar. 1996, pp. 9-14.
9. V. Berzins and Luqi, *Software Engineering with Abstractions*, Addison-Wesley, New York, 1990.
10. Software Development and Documentation, MIL-STD-498, US Dept. of Defense, Washington, D.C., 1994, http://www.itsi.disa.mil/cfs/std498.html.
11. Luqi, V. Berzins, and R. Yeh, "A Prototyping Language for Real-Time Software," *IEEE Trans. Software Eng.*, Vol. 14, No. 10, 1988, pp. 1409–1423.
12. V. Berzins, *Software Merging and Slicing*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1995.
13. J. Goguen, "Principles of Parameterized Programming," *Software Reusability, Volume I: Concepts and Models*, T. Biggerstaff and A. Perlis, eds., Addison-Wesley, New York, 1989, pp. 159–225.
14. W. Tracz, "Parameterized Programming in LILEANNA," *Proc. 2nd Int'l Workshop Software Reuse*, IEEE Computer Soc. Press, Los Alamitos, Calif., Mar., 1993, pp. 66-78.
15. M. Stickel et al., "Deductive Composition of Astronomical Software from Subroutine Libraries," *Conf. Automated Deduction*, Vol. 12, Springer-Verlag, Heidelberg, Germany, 1994.
16. J. Schwartz and W. Snyder, "Design of Languages for Multimedia Presentations," *Proc. 1994 Monterey Workshop: Increasing Practical Impact of Formal Methods for Computer-Aided Software Development*, Naval Postgraduate School, Monterey, Calif., 1994, pp. 46-55.
17. Luqi and J. Goguen, "Some Suggestions for Progress in Software Analysis, Synthesis and Certification," *Proc. 6th Int'l Conf. Software Eng. and Knowledge Eng.*, Knowledge Systems Inst., Skokie, Ill., 1994, pp. 501–507.
18. E. Engeler, "Structure and Meaning of Elementary Programs," *Lecture Notes in Mathematics*, Vol. 188, Springer-Verlag, New York, 1971, pp. 89-101.
19. V. Berzins and Luqi, "An Introduction to the Specification Language Spec," *IEEE Software*, Mar. 1990, pp. 74-84.

Luqi is a professor of computer science at the Naval Postgraduate School, where she leads a team that is producing computer-aided prototyping tools in a distributed high-performance lab. This team also developed the CAPS rapid-prototyping system. Luqi has also worked on software R&D for the Science Academy of China, the Computer Center at the University of Minnesota, International Software Systems, and others.

Luqi received a PhD in computer science from the University of Minnesota. In addition to chairing or serving on the program committees of more than 40 conferences, she is or has been an associate editor for *IEEE Expert*, *IEEE Software*, the *Journal of Systems Integration*, and *Design and Process World*. She is a senior member of the IEEE.

Joseph Goguen is a professor in the Department of Computer Science and Engineering at the University of California at San Diego and director of the Program in Advanced Manufacturing. Previously, he was a professor at Oxford University, a senior staff scientist at SRI International, and a senior member of the Center for the Study of Language and Information at Stanford University.

Goguen received a BS in mathematics from Harvard University and an MS and a PhD in mathematics from the University of California at Berkeley. He is a member of the IEEE.

Address questions about this article to Luqi at NPS, Computer Science, Monterey, CA 93943; luqi@cs.nps.navy.mil; or to Goguen at Dept. of Computer Science and Engineering, University of California at San Diego, 9500 Gillman Drive, La Jolla, CA 92093-0114; goguen@cs.ucsd.edu.