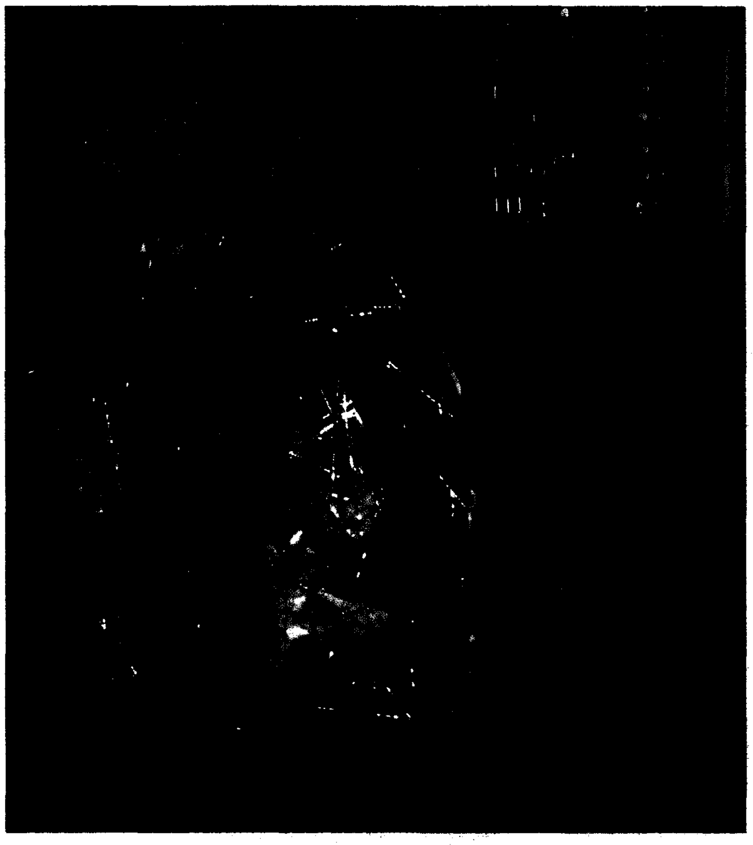
1988

Knowledge-Based Support for Rapid Software Prototyping

Luqi



Knowledge-Based Support for Rapid Software Prototyping

Luqi
The Naval Postgraduate School

Software prototypes are executable initial versions of software systems. Designers use prototypes to clarify requirements by demonstrating selected aspects of proposed system behavior to customers. To be useful in negotiations leading to software development projects, prototypes must be constructed and adapted to rapidly changing requirements. PSDL[1] (our prototyping system description language) and its associated computer-aided prototyping system[2] make this possible on a large scale as well as for embedded systems with real-time constraints. PSDL can express black-box descriptions of systems and decompositions into networks of simpler operators communicating via data streams. Associated prototyping methodology relies on reusable software components, drawn from a software base, to speed up prototype construction.[3] In addition to the software base, the computer-aided prototyping system contains (1) a translator for adapting and interconnecting components, (2) schedulers for meeting real-time constraints, and (3) interfaces for entering design decisions.[2]

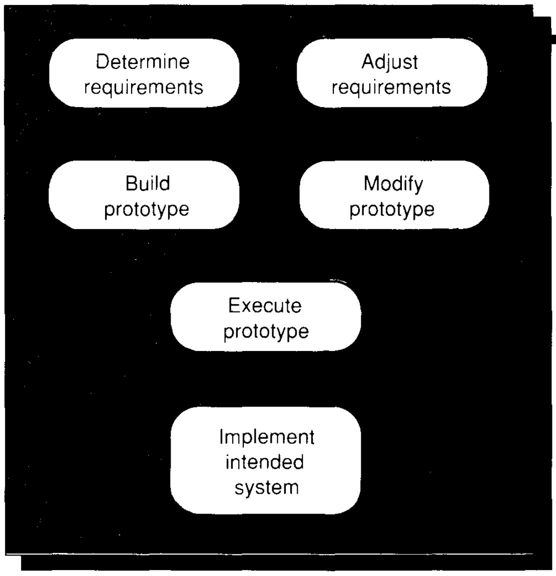


Figure 1. Using the computer-aided prototyping system.

Focusing on the software base, we will discuss the computer-aided prototyping system's knowledge base and show how to apply expert system technology to the software base management subsystem responsible for finding reusable software components with specified properties. We will apply rapid prototyping techniques to enable the detection and correction of requirement errors early in development. Prototyping is most useful in novel or poorly understood problem domains where designers find it impossible to anticipate every customer problem in advance. To enable more effective planning and to ensure that detailed design and implementation efforts will focus on the most important features, the prototyping effort tries to highlight unanticipated aspects of customer problems early on. We seek to automate support for analyzing new problem domains rather than automating the generation of many similar systems in well-understood problem domains. For this reason, we will address a general-purpose computer-aided prototyping system rather than specialized application generators.

Effective application generators require prior analysis of the problem domain to (1) develop a specialized problem description language, and (2) identify algorithms capable of solving all problems that language can express. This process — the easiest for narrowly defined problem domains — requires a substantial tool-building effort for each new application. The investment required to create a special-purpose application generator is justified if we plan to construct many software systems for the same problem domain.[4] The time required to construct application generators precludes their use in rapid prototyping for novel problem domains. The domain analysis required to construct an application generator for a new domain will benefit from prototyping with a computer-aided system.

Because they apply to software development generally, rather than to a fixed application area, computer-aided prototyping systems are more difficult to build than application generators. Consequently, we cannot anticipate application area problems and provide ready-made solutions (as is done in application generators). A completely automated system must solve all problems expressible in its problem statement language. Since automatically generating programs for solving problems in unconstrained domains is beyond the current state of the art, interactive systems offer practical support for prototyping.

We have designed our computer-aided prototyping system to assist human experts with software design. Our system contains knowledge comprising software components and rules describing how these components can be used and combined. Human experts provide guidance for solving problems that cannot be handled by automatically applying the expertise represented in the prototyping system's knowledge base. This structure enables us to build a useful system without requiring prior solutions to all possible problems, and supports the system's evolutionary development by allowing the addition of new components and rules to the knowledge base as we discover gaps in the system's knowledge through practical use.

Our computer-aided prototyping system functions as follows:

Designers obtain requirements from customers as written documents, with extensions and clarifications provided in response to designer questions and prototype demonstrations.

Designers propose a system interface consistent with these requirements, or make adjustments based on customer feedback, and record resulting specifications in PSDL.

Designers submit specifications to the software base management system, which attempts to retrieve or adapt available reusable components to meet those specifications. If this is impossible, then designers must decompose the system into a network of simpler components using PSDL, and repeat the process at more detailed levels until the software base can provide all components.

Figure 1 illustrates this process, in which component specifications have a dual role since they are used both for documenting intended prototype properties and for retrieving reusable components. Because this process is difficult to automate, designers must propose abstractions and carry out top-down design. The software base management system must find relevant components and fill in details. This is analogous to the interaction between mathematicians and an automatic theorem-proving program in which mathematicians propose lemmas as intermediate steps if the automatic procedure cannot solve the main theorem in a reasonable time; the automatic procedure takes care of fine-grained details that are tedious for the mathematician.

Our approach differs from other knowledge-based approaches to program construction by (1) the scale of its knowledge base, and (2) its computer-aided retrieval of reusable components based on specifications. The Programmer's Apprentice project[5] aims at speeding up the programming process via a library of reusable components. Reusable components implemented in the KBEmacs version of the Programmer's Apprentice — components known as "cliches" — represent algorithm fragments rather than complete modules. Using algorithm fragments, KBEmacs focuses on supporting the assembly of a module's implementation rather than assembling systems from complete modules; the scale of examples reported is a few hundred lines of code. We aim to produce software system prototypes larger by several orders of magnitude, and seek to avoid considering the internal structure of reusable components.

Programmers using KBEmacs should be familiar with library cliches (which are referenced by name) and will obtain the best results if they think in terms of these cliches. This is not a very serious problem in the prototypical version of KBEmacs, which contains only a few dozen cliches, but indicates that hundreds or thousands of cliches will probably be needed for serious applications.[5]

Learning and remembering cliches can become burdensome. Our approach uses specification-based retrieval of reusable components to avoid this burden. The software base management subsystem seeks to automate the process of organizing, finding, and combining reusable components in the software base via specifications associated with components. Such capability is useful because it frees designers from having to remember what software components are currently available or from having to browse through components to discover relevant ones. This becomes especially important as available components increase in number (as they will in software bases supporting broad problem domains).

Expert system technology is appropriate for realizing the software base management system because the useful software component set is unbounded in principle and can be enormous in practice. We can view these components as instances of general patterns, with many variations on some of their properties. A practical approach is to store a representative component subset and to provide rules for generating related components (practical since it reduces the effort required to enter components into the system, conserves memory, and imposes structure on the software base).

Retrieving for the software base requires finding or constructing a program that realizes a given specification. Retrieving reusable components from a software base is a difficult search problem in a potentially infinite space. The elements of this space are pairs consisting of a specification and a corresponding executable program in which the elements of each pair are either explicitly stored or effectively constructible from explicitly stored pairs. This suggests using heuristic search methods, which are a common expert system component. Complete algorithmic solutions to the retrieval problem are unlikely because specification equivalence is an insoluble problem if the specification language is strong enough to be expressive. Our system addresses retrieval by relying on a recognizable subset of the equivalence relation on specifications.

Retrieval strategies

The software base management system retrieves reusable software components for meeting a given specification, using less designer time than it would take to code components manually. Since prototyping is most important for large problems, designer throughput provides a more appropriate measure than system delay to realize each component. Perceived time delay — an important factor in the effort required to use a system[6] — is valuable when producing answers with little perceptible delay because it avoids disturbing the designer's thought processes. Since searching a large space takes considerable time, practical designs should incorporate several search strategies ordered by speed. Those that the system can apply with little perceptible delay should be applied while the designer waits. If fast methods fail, retrieval should be spooled (thereby allowing designers to work on different aspects of the problem while the system pursues more time-consuming background search strategies).

The fastest and most superficial search strategy uses exact matches for component specifications. To facilitate retrievals based on exact matching, the system normalizes component specifications, transforming them into standard form.[7] This reduces variations when representing equivalent specifications, making syntactic matching more effective. The system normalizes specifications for retrieval requests and for each component entered into the software base. The system indexes components according to their normalized specifications; consequently, exact match searches can be performed quickly.

Retrieval strategies based on inexact matches and transformations consume more time, and are best applied off line. Checking for inexact matches and applying scribe transformations can both diverge — and both can be controlled by externally imposed time limits adjusted to match available computational resources (we will discuss transformations in detail below). Searching for inexact matches involves checking implications. If QS represents query specifications and CS represents specifications for reusable component C, then C is an inexact match for the query in case CS => QS. Checking implications is not a decidable problem in general, but several partial decision rules exist that will correctly detect successful implications. Partial decision rules include shallow pattern matching based on the identities

$$A \text{ and } B \Rightarrow A$$
$$A \Rightarrow A \text{ or } B,$$

a propositional implication checker (based on truth tables) that ignores atomic predicate semantics and bounded-depth inference procedures sensitive to properties of common data types (for example, integers and sets) and common operations (including equality and orderings). To save time, we should apply inexact match and transformation strategies to relatively small portions of the software base; consequently, we have partitioned the software base according to the values of several categorical properties, which we detail under the "category" slot below. This structure resembles a hash table, in which categorical property values correspond to the hashing function. We identify the partition relevant to a given retrieval request based on property values, and components in the partition are subjected to a "best first" heuristic search that checks for inexact matches and attempts to create matches by applying various transformations. To realize the full potential of software use, such processing is important; exact matches can get you only so far.[8] The software base management system can find more opportunities to reuse software components created by human designers if small modifications for adapting components to a new environment can be made automatically.

An agenda mechanism resembling the one used in the AM system[9] can choose which component to analyze next. AM generates new mathematical concepts, with the most "interesting" concepts appearing first. The interpretation of "interesting" relevant to the software base management system's retrieval mechanism reflects the degree of similarity between component specification and the specification contained in the retrieval request. If a multiple-computer network is available, the system can process several of the most promising components in parallel.

The computer-aided prototyping system's knowledge base contains two kinds of information — descriptions of reusable software components, and rules for combining or adapting reusable components (the next two sections describe this knowledge in more detail).

Declarative knowledge

Primarily, the knowledge base's declarative knowledge contains reusable software components and can be organized naturally into a frame system[10] — each frame corresponding to a software component. Such organization is preferable to monolithic structures (Prolog databases, for example) because the software base can become unwieldy. The frames impose structure on the software base, which the system can use for limiting searches and for organizing secondary storage so that information units likely to be needed together can be retrieved as a single unit.

Frames in a frame system have common slots that have the same interpretation for each frame in the system. Each slot can have several facets. Each facet contains a property related to the slot. The following subsections describe slots important for managing a set of reusable software components.

Specification. The specification slot contains a PSDL specification of the software component. Retrieval is based on the specification rather than on any attempt to analyze implementation code.

Implementation. The implementation slot contains code for software components. Consistent with our principle of specification-based retrieval, we maintain only one implementation for each elementary specification. If a significant difference exists between two modules, module specifications should describe that difference so that retrieval mechanisms can be sensitive to it. If no significant difference exists, keeping both modules wastes space. Performance differences are sometimes important; hence, specifications must reflect these differences. The implementation slot of a nonelementary specification contains a list of other modules satisfying the specification. Nonelementary specifications help the system avoid recalculating common inexact matches, and do not affect the system's computational power in the absence of resource limits.

Category. The category slot contains several properties used for partitioning the software base into disjoint subspaces. By limiting the knowledge base portion that it must search, the system uses such partitioning to improve performance. Categorical properties include the programming language used for implementation, the operating system it runs under, the component type, and maximum execution time. We have found component types of PSDL (functions, state machines, and data types) convenient for describing prototype software systems, since they provide a clear criterion for defining disjoint categories. The system uses maximum execution time to specify modules with real-time constraints, which are restricted to execute in a constant time frame. Therefore, maximum execution time can be represented as a number used to limit retrievals to only those modules capable of meeting a given real-time constraint. To efficiently generate subsets of components meeting a given bound on execution time, each partition's components are threaded together in a list kept sorted with respect to maximum execution time.

PSDL is the language of choice when rapidly prototyping real-time systems.

Alternatives. The alternatives slot contains (1) rules for generating variations on a module, and (2) a list of related modules. The next section describes filter rules that exemplify rules for generating variations. Filter rules contain prescriptions for synthesizing composite modules by combining the current module with other modules that satisfy specified properties. Each related module is associated with a description of conditions under which the module is likely to be useful. For example, a module that finds roots by Newton's method can have a module for finding roots via the bisection method as an alternative, with the conditions that (1) derivatives are not needed, but that (2) end-points of an interval where the function reverses its sign are required as additional inputs to the module. Rules for generating variations are essential when defining the search space, while the list of related modules helps the system become more efficient by supplying heuristic advice regarding where to look next.

Transformations

It has been difficult to reuse software in practice because only rarely are two instances of the "same" software component exactly alike at a superficial level. Instead, many small variations on a theme exist. The number of small variations can be unbounded, and it can be difficult to predict which variation will be needed next (an extreme example is the set of array-sorting routines in standard Pascal). Since Pascal requires that the procedure header specify the type and bounds of the array, a different sorting procedure is needed for each array type and size.

We can provide transformations that adapt or combine components explicitly stored in the software base to accommodate a class of small variations, thereby alleviating this problem. Systems performing such transformations as part of the retrieval process have a much better chance of successfully retrieving a specified component from the software base than do systems that can only return components explicitly stored in the software base. This capability also reduces the need for designers to manually adapt reusable components after retrieval — especially important in rapid prototyping, where designer time is at a premium. The constructed implementation's efficiency is less important in prototyping, where the primary purpose is to discover system behavior acceptable to customers. Once component specifications have stabilized, identifying performance bottlenecks and reimplementing critical modules more efficiently can be justified.

A well-known approach to the small-variations problem defines parameterized generic components, thereby providing a single representation for a whole class of related components. Both the specification and implementation of generic components have one or more formal parameters that the matching process can bind to adapt components to specific applications. The retrieval mechanism must include a transformation that creates the corresponding generic-code-template instantiation. This transformation can involve some computation to expand substitutions in line for programming languages (including Pascal) that do not explicitly support generic units. Such transformations may be needed even for languages supporting generic code units (Ada, for instance) because the software base can provide a more powerful or flexible parameterization mechanism than that provided by the programming language.

Another kind of transformation involves small local rearrangements to the reusable component interface. These transformations include permuting input and output parameters, ignoring extra output parameters, and filling in values for extra input parameters. Such transformations embody general-purpose strategies encoded in knowledge base rules. The transformation for filling in extra input parameters resembles the process of determining an actual parameter value for a generic module via the matching process. For example, a query requesting a component to calculate length (**v**) (where **v** is a three-dimensional vector) can be satisfied by an operation magnitude (**v**,3) where magnitude (**v**,*n*) calculates the length of an *n*-dimensional vector.

Other transformations yield building blocks for a composite component. An important expert system function for retrieving reusable components involves a limited amount of bottom-up design, necessary if we are to free designers from having to remember all reusable components in the software base. If we limit this knowledge to the expert system, the expert system must steer decomposition in directions that match available components. One way to achieve this uses composition rules (heuristics indicating plausible ways to extend a module). Some composition rules are applicable generally, while others are associated with specific components or component classes in the software base. This kind of information fits in a module's alternatives slot.

Guard rules and filter rules form two important composition rule categories. Guard rules describe decompositions induced by case analysis, and produce conditional statements by matching reusable component specifications against part of a retrieval specification. If the retrieval specification is S and the reusable component specification has the form $G \Rightarrow S$, the component can be augmented with a PSDL control constraint of the form

triggered if G

provided that the guard G can be expressed in a form executable in PSDL. We generate guard rules by matching parts of the retrieval specification against a component specification. If the match succeeds, the guard rule provides a partial implementation that can be completed by providing other conditional implementations applicable if G is false.

Figure 2 illustrates a typical decomposition produced by the guard rule. PSDL control constraints below the dataflow diagram give conditions under which each operator is invoked. The "?" represents an unknown operator that must be found or constructed to complete the implementation. This operator must meet the specification

not $G \Rightarrow S$.

Guard rules are important because they enable retrieval of partial operations that satisfy query specification under a restricted set of conditions, thus providing some bottom-up guidance to designers about decomposing problems in a way that matches available components. By reporting conditions under which a known method will satisfy specification, such rules also allow useful responses for specifications not satisfiable in all cases. This helps expose and correct conceptual oversights on the part of designers who may be contemplating the most common case and fail to notice that some circumstances exist under which normally desired results cannot be achieved. Guard rules that result in partial implementations covering normal cases can help focus designer attention on abnormal cases, possibly leading to the identification of missing exception conditions or error messages in prototype design. For example, searching for a component that calculates a set's maximum element may result in a guarded operator that fulfills the specification in cases where the set is not empty. Such a response focuses designer attention on whether the set can be empty and, if it can, on identifying an appropriate response for the component being prototyped. This represents cases where bottom-up guidance from the knowledge base can identify specification faults and suggest changes to prototype design.

Filter rules provide another way to suggest decompositions based on bottom-up criteria. Filter rules factor specifications, enabling them to be met by two-component dataflow decompositions in which (1) the first component is a filter operation available in the software base, and (2) the filter rule derives the second component's specification from the query specification. For example, the filter rule associated with a sorting operator matches query specifications of the form

$$(x \text{ IN } s \Leftrightarrow P(x)) \ \& \ \text{sorted}(s).$$

Figure 3 illustrates decomposition produced by this filter rule. The sorting-operator filter rule specifies that operator "?" must produce output sequence "s" such that

$$x \text{ IN } s \Leftrightarrow P(x).$$

As in the decomposition suggested by the guard rule, "?" represents an unknown operator that must be found or constructed. Filter rules, sensitive to filter operation semantics, must be specified along with the filter operation when it's added to the software base.

The need for computer-aided rapid prototyping of software systems, particularly systems with real-time constraints, motivated our work on methods for retrieving reusable software components. While reusable components can effectively reduce designer effort, we must relieve designers of the need to remember the contents of large component libraries. We have outlined an expert system structure for retrieving reusable software components from a knowledge base that uses component specifications. Rule-based retrieval, combined with a limited ability to adapt and combine available components, is essential for making extensive software reuse practical in prototyping because standard software components appear with widely ranging variations in many practical applications. The likelihood of being able to reuse components increases if the system can provide many variations on each component that was manually constructed and entered into the software base. Rules for composing stored components are important for reducing the designer's knowledge load. We can use such rules to provide a limited amount of bottom-up design based on available components of which designers may not be aware.

We have described our approach to the automated retrieval of reusable software components in terms of the prototyping language PSDL because it supports a computer-aided prototyping system and contains several features that make our approach easier to realize — explicit black-box specifications for each component, for instance, and facilities for describing dataflow decompositions subject to nonprocedural control constraints. Our approach can be applied to other languages (including Prolog) by establishing suitable conventions and usage patterns. An essential part of our approach is the storing of a specification for each reusable software component, which can be done in the context of many different languages. However, since it contains facilities for expressing timing constraints, and since the associated execution support system contains facilities for realizing such constraints, PSDL is the language of choice for the rapid prototyping of real-time systems. Prototyping is especially important for this class of systems because their requirements are difficult to discover and understand.

Developing a complete computer-aided prototyping system is a long-range research project. We have designed and partially implemented the first version of such a system. The computer-aided prototyping system's subprogram for retrieving reusable components (still in the design stage) will require substantial effort to implement and evaluate because we must assemble an effective set of reusable software components to refine the proposed system's rules and test its effectiveness. Important areas for future research include finding more effective transformation sets and more efficient matching algorithms. We must also assemble an effective set of reusable software components for the software base. Existing components must be generalized, specified, and certified (or thoroughly tested, at least) before they can become useful members of the software base. To build up a comprehensive and effective software base, an extended incremental process is needed; experience with using initial versions of the software base will indicate the most important directions for growth.

Acknowledgments

This work was supported in part by the National Science Foundation under Grant No. CCR-8710737. We thank Henry Ayling for his careful editing and Armard Prieditis, Jack Mostow, Yuh-Jeng Lee, Robert Herndon, and Valdis Berzins for their comments and suggestions on earlier drafts of this article.

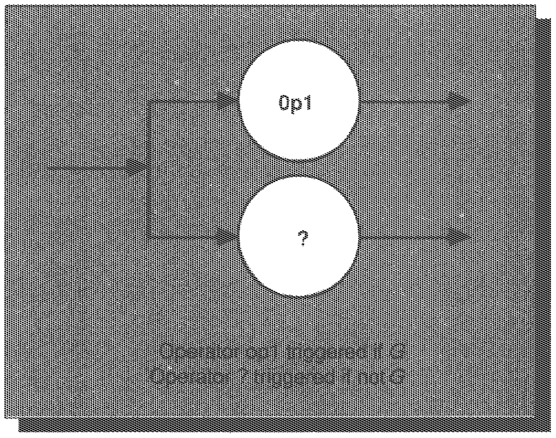


Figure 2. A guard rule decomposition.

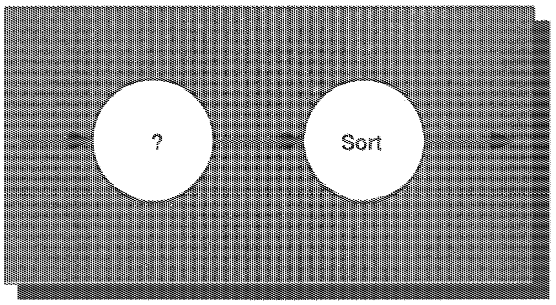


Figure 3. A filter rule decomposition.

References

1. Luqi, V. Berzins, and R. Yeh, "A Prototyping Language for Real-Time Software," *IEEE Trans. Software Engineering*, Oct. 1988.

2. Luqi and M. Ketabchi, "A Computer-Aided Prototyping System," *IEEE Software*, Mar. 1988, pp. 66-72.

3. Luqi and V. Berzins, "Rapidly Prototyping Real-Time Systems," *IEEE Software*, Sept. 1988, pp. 25-36.

4. P. Freeman, "A Conceptual Analysis of the Draco Approach to Constructing Software Systems," *IEEE Trans. Software Engineering*, July 1987, pp. 830-844.

Continued on page 18

5. R. Waters, "The Programmer's Apprentice: A Session with KBEmacs," *IEEE Trans. Software Engineering*, Nov. 1985, pp. 1296-1320.

6. S. Card, T. Moran, and A. Newell, *The Psychology of Human-Computer Interaction*, Lawrence Earlbaum Associates, Hillsdale, N.J., 1983.

7. Luqi, "Normalized Specifications for Identifying Reusable Software," *Proc. ACM-IEEE Fall Joint Computer Conference*, IEEE Computer Society, 10662 Los Vaqueros Circle, Los Alamitos, Calif., Oct. 1987.

8. J. Mostow and M. Barley, "Automated Reuse of Design Plans," *Proc. Int'l Conf. Engineering Design*, American Society of Mechanical Engineers, Boston, Mass., Aug. 1987, pp. 632-647.

9. R. Davis and D. Lenat, *Knowledge-Based Systems in Artificial Intelligence*, McGraw-Hill, New York, N.Y., 1982.

10. M. Minsky, "A Framework for Representing Knowledge," in *Readings in Knowledge Representation*, R. Brachman and H. Levesque, eds., Morgan Kaufmann, Los Altos, Calif., 1985, pp. 245-262.

Luqi (pronounced lew-chee) has been an assistant professor of computer science at the Naval Postgraduate School since 1986. Her research interests include software development tools, rapid prototyping, real-time language, and design methodology. She received her BS in computational mathematics from Jilin University (PRC) and her MS and PhD in computer science from the University of Minnesota, where she taught and performed software research and development. Before joining the Naval Postgraduate School, she worked for International Software Systems, Inc., and did research for the Science Academy of China in Beijing.

Luqi can be reached at the Computer Science Dept., Naval Postgraduate School, Monterey, CA 93943-5100.