Box A: Artificial Intelligence and Software Engineering Approaches to Intelligent Assistance

Artificial intelligence researchers have moved us closer to Winograd's dream by developing a *knowledge-based* approach to programming. The knowledge includes relationships among program units, both in the abstract and with respect to a particular target system. The Masterscope package of Interlisp[1], the Formalized System Development testbed (FSD[2]) and its offspring CommonLisp Framework (CLF[3]) maintain cross-referencing information in order to answer queries about interactions among program units. FSD's knowledge base also understands the abstract relationships among program units. For example, FSD 'knows' that a system consists of modules and individual software objects (functions, variables, *etc.*) and that each of these classes of objects has particular properties, such as a maintainer and whether or not it has been compiled.

Some knowledge-based programming environments also include some of the rules governing the software development process. For example, wide-spectrum languages such as V[4] and Gist[2] have been augmented with rules that aid the programmer in interactively translating from higher to lower level specifications and from specifications to executable code. CMS[5] provides a formal representation of the software project model, time (for scheduling), and software development activities. For example, if a 'capability' is a desired feature of the target system, then it must be realized by a 'component' of the system; a 'task' must be defined to specify who is in charge of the component and when it is due. Genesis' Activity Manager[6] provides similar facilities.

While this work in AI was progressing, researchers with backgrounds in software engineering were also trying to achieve their own version of Winograd's dream. These researchers developed *tools* that automated certain aspects of the programming process. For example, Make[7] automatically rederives an executable system when part of the source code changes. SCCS[8] requires programmers to reserve their modules for change and thus ensures that software evolves in an orderly manner. RCS[9] supports multiple versions of software objects, manages parallel lines of development, and aids programmers in merging versions.

Collections of tools have been integrated into interactive programming environments that support a particular programming language. The Synthesizer[10] combines language-oriented editing that prevents syntactic errors with immediate feedback about static semantic errors; it

also permits programmers to interleave execution and debugging with editing. The Gandalf Prototype[11] added a module interconnection language with incremental, intermodule consistency checking to a C programming environment[12] similar to the Synthesizer. Such environments can also be generated from formal specifications; for example, the Synthesizer Generator[13] takes as input an attribute grammar for the desired programming language and produces a language-based environment with incremental type checking and code generation.

Unfortunately, the knowledge-based approach and the tools approach have progressed more-or-less independently. The individual tools incorporate a small bit of knowledge about a particular programming problem, but this knowledge cannot be augmented. The programming environments are hardcoded (either by handcoding or generation) with a particular view of the software development process that defines the interaction between the programmers and the target system, but this knowledge is not available to the users of the environment. The knowledge-based environments are much more general, but currently represent only a tiny fraction of human expertise about software development and maintenance.

References:

1. Warren Teitelman and Larry Masinter, ‘‘The Interlisp Programming Environment’’, *IEEE Computer*,Vol. 14No. 4April 1981, pp. 25-34, Reprinted in David R. Barstow, Howard E. Shrobe, and Erik Sandewall (Eds.), *Interactive Programming Environments*, McGraw-Hill Book Co., New York, 1984.

2. Robert Balzer, ‘‘A 15 Year Perspective on Automatic Programming’’, *IEEE Transactions on Software Engineering*,Vol. SE-11No. 11November 1985, pp. 1257-1268.

3. David S. Wile and Dennis G. Allard, ‘‘Worlds: an Organizing Structure for Object-Bases’’, *2nd SIGSOFT/SIGPLAN Symposium on Practical Development Environments*, December 1986, pp. 16-26, Proceedings published as *SIGPLAN Notices*, vol. 22, no. 1, January 1987.

4. Douglas R. Smith, Gordon B. Kotik and Stephen J. Westfold, ‘‘Research on Knowledge-Based Software Environments at Kestrel Institute’’, *IEEE Transactions on Software Engineering*,Vol. SE-11No. 11November 1985, pp. 1278-1295.

5. Beverly L. Kedzierski, ‘‘Knowledge-Based Project Management and Communication Support in a System Development Environment’’, *4th Jerusalem Conference on Information Technology*, Jerusalem, Israel, May 1984.

6. C.V. Ramamoorthy, Vijay Garg and Rajeev Aggarwal, ‘‘Environment Modelling and Activity Management in Genesis’’, *SoftFairII: 2nd Conference on Software Development Tools, Techniques, and Alternatives*, December 1985, pp. 2-10.

7. S. I. Feldman, ‘‘Make — A Program for Maintaining Computer Programs’’, *Software — Practice & Experience*,Vol. 9No. 4April 1979, pp. 255-265.

8. M. J. Rochkind, ‘‘The Source Code Control System’’, *IEEE Transactions on Software Engineering*,Vol. SE-1 1975, pp. 364-370.

9. Walter F. Tichy, ‘‘RCS — A System for Version Control’’, *Software — Practice and Experience*,Vol. 15No. 7July 1985, pp. 637-654.

10. Tim Teitelbaum and Thomas Reps, ‘‘The Cornell Program Synthesizer: A Syntax-Directed Programming Environment’’, *Communications of the ACM*,Vol. 24No. 9September 1981, Reprinted in David R. Barstow, Howard E. Shrobe, and Erik Sandewall (Eds.), *Interactive Programming Environments*, McGraw-Hill Book Co., New York, 1984.

11. Nico A. Habermann, David Notkin, ‘‘Gandalf: Software Development Environments’’, *IEEE Transactions on Software Engineering*,Vol. SE-12No. 12December 1986, pp. 1117-1127.

12. Peter H. Feiler and Raul Medina-Mora, ‘‘An Incremental Programming Environment’’, *IEEE Transactions on Software Engineering*,Vol. SE-7No. 5September 1981, pp. 472-482.

13. Thomas Reps and Tim Teitelbaum, ‘‘The Synthesizer Generator’’, *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, April 1984, pp. 41-48, Proceedings published as *SIGPLAN Notices*, vol. 19, no. 5, May 1984.

@boo B: Smile

Working within the tools approach (see @boo A) during the late 1970's and early 1980's, the authors and other members of the Gandalf project developed a multiple-user software engineering environment called SMILE.[1, 2] This environment supports programming in C and runs on Unix*. SMILE has been relied on by the Gandalf[3] and Gnome[4] projects at CMU and by the Inscape project[5] at AT&T Bell Labs, and has been distributed to at least forty sites. SMILE passes the crucial test of supporting its own maintenance. It has supported the simultaneous activities of at least seven programmers, and the largest software system developed and maintained in SMILE has approximately 61,000 lines of source code.

SMILE is relatively close to achieving intelligent assistance. It provides a 'fileless environment' to its users, answers queries, coordinates the activities of multiple programmers, and automatically invokes various tools; it hides the particularities of the Unix file system and utilities and presents its own model of the programming world. SMILE's objectbase is implemented through a combination of file system and in-core object structure that is kept persistent in a file. SMILE's knowledge of software objects and the programming process is hardcoded into the environment. From experience with SMILE and other environments, the authors gained insights into the development of practical environments and became convinced that a generalization of SMILE's internal architecture would ease development of intelligent assistants for software development and maintenance. The resulting architecture is the topic of this article.

References:

1. Nico A. Habermann, David Notkin, ''Gandalf: Software Development Environments'', *IEEE Transactions on Software Engineering*,Vol. SE-12No. 12December 1986, pp. 1117-1127.

2. Gail E. Kaiser and Peter H. Feiler, ''Intelligent Assistance without Artificial Intelligence'', *Thirty-Second IEEE Computer Society International Conference*, San Francisco, CA, February 1987, pp. 236-241.

3. David Notkin, ''The GANDALF Project'', *The Journal of Systems and Software*,Vol. 5No. 2May 1985, pp. 91-105.

4. David B. Garlan and Philip L. Miller, ''GNOME: An Introductory Programming Environment Based on a Family of Structure Editors'', *SIGSOFT/SIGPLAN Software*

---

*Unix is a trademark of AT&T

Engineering Symposium on Practical Software Development Environments, Pittsburgh, PA, April 1984, pp. 65-72, Proceedings published as *SIGPLAN Notices*, vol. 19, no. 5, May 1984.

5. Dewayne E. Perry, ‘‘Version Control in the Inscape Environment’’, *9th International Conference on Software Engineering*, Monterey, CA, March 1987, pp. 142-149.

@boo C: Validation of Marvel's Architecture

The purpose of the prototype implementation of MARVEL is to validate the practicality of the MARVEL architecture. The implementation of MARVEL's rules and strategies must be able to capture at least the capabilities of an existing environment that supports automation. We chose SMILE both as the environment whose functionality was to be modelled, and as the environment to use as the basis for the prototype implementation. Starting with SMILE for the prototype implementation permitted us to concentrate on the implementation of an objectbase mechanism and the rule processing facility, but still have an operational environment prototype. It permitted us to compare the prototype with the original SMILE system, which has been in practical use for several years, and to exercise the prototype on real-world software that has been developed under SMILE.

This implementation replaced SMILE's hardcoded objectbase structure with one that is more appropriate to the MARVEL architecture. Single inheritance is supported, with a root OBJECT class. Inheritance is static, however, meaning later changes to superclasses do not affect the subclasses and, in fact, later changes to classes do not affect previously created instances of the classes. Instances of objects are explicitly created, updated and deleted, so garbage collection is not required. Views are implemented through different report generators, and the user interface is text based.

This implementation of MARVEL also replaced SMILE's hardcoded knowledge about the software development process with rules. Rules and strategies are written using a text editor, and the text file is parsed by the rule compiler. The rule compiler translates the preconditions and postconditions of rules into (1) a 'fast load' syntax tree and (2) symbol table structures that link each occurrence of a predicate or a relation in a precondition with a potentially satisfying postcondition and *vice versa* and also links these predicates and relations to each relevant rule. A specific rule set and strategy can be loaded at startup and additional strategies can be loaded later, but there is no consistency checking among simultaneously employed strategies. Currently it is not possible to unload complete strategies, but individual rules can be separately turned off and on.

Forward chaining, backward chaining, and the ability to turn strategies on and off are implemented through an interpreter that works directly with the structures produced by the rule compiler. This rule interpreter takes a simple approach for processing rules rather than

employing a match network mechanism; the entire condition of every applicable rule is rechecked whenever a relevant predicate or relation is asserted or negated. The performance resulting from this simple-minded approach is unacceptable for large numbers of rules and/or large objectbases. Forward chaining proceeds breadth-first using a queue of rules whose preconditions are satisfied. Backward chaining is depth-first, attempting to derive the desired postconditions of one candidate rule before trying an alternative rule.

Once the objectbase and the rule compiler and interpreter were in place we were able to capture SMILE's knowledge about programming activities and their automation in rules and strategies and replace the hardcoded knowledge. In the process we were able to rectify a couple of shortcomings in the automation capabilities of the original SMILE system. Since SMILE permits tools to execute concurrently with the user in the background, special attention needed to be given in the implementation of the rule interpreter to support limited concurrent flow of action for rule processing and updating of the objectbase.

@boo D: Adding Knowledge to Software Development Tools

Make[1] has a simplistic world model consisting of files and command lines. A 'makefile' defines dependencies among files and gives the command lines for restoring consistency among dependent files. Make's notion of consistency is based entirely on files and time: If the timestamp of an input file is later than the timestamp of an output file, then the indicated command line is passed to the Unix shell. Make is widely used for generating a new executable version of a system after one or more source files have been modified.

However, Make's knowledge is primitive. Its objectbase consists of files that have a single attribute, their timestamp. Make does not know anything about applying tools to files; it just handles command lines as indivisible strings. Make does not have any understanding of source files *vs.* object files, of modules *vs.* systems, of programmers or of programming.

Consider how this knowledge might be added to Make. First a notion of an *object* is defined, where each object belongs to a *class*. One class might be 'system', while another might be 'module'. Each class would define *attributes*, or properties, of its objects. For example, a 'module object code' object might have a 'history' attribute, which describes how it was generated, and a 'derivation of' attribute, which points to the object representing the corresponding source code. Then *rules* would be added to model the part of the software development process relevant to Make. One rule might be that a 'programmer' object can modify a 'module' object; another might state that after such a modification, the 'module' object is no longer consistent with its 'derivation' attribute and there is an obligation to restore this consistency. A third rule might state that a precondition for a 'programmer' to test a 'system' is that all 'module object code' objects that are components of the corresponding 'executable system' must be consistent with their 'module'.

If Make were armed with this knowledge, then it would be more intelligent than it is now. It would then be easier to integrate Make with other tools that support configuration management, version control, task management, *etc.*, assuming all these tools were similarly augmented with knowledge of software objects and with understanding of their roles in the software development process.

The Synthesizer[2] also has a simplistic world model, consisting of nodes in a parse tree. The nodes have types, such as 'program' and 'identifier'. When an 'identifier' node is inserted as a

child of an ‘expression’ in the parse tree, the Synthesizer compares the identifier’s name with the names defined in the symbol table. If not found, the part of the display corresponding to the new node is highlighted; the highlighting is removed when a matching ‘identifier’ node is inserted as a child of a ‘declaration’. The immediate feedback provided by the Synthesizer makes it easy to correct static semantic errors while the programmer is still in the context of editing a program.

The primitive knowledge of the Synthesizer has been improved somewhat in the Synthesizer Generator[3]. The Synthesizer Generator uses a knowledge base that defines classes of nodes such as ‘expression’, attributes of nodes such as ‘type’, and equations that specify dependencies among attributes. The language-based editors produced by the Synthesizer Generator automatically re-evaluate the attribute equations whose input attributes have changed in value. However, these editors do not know that the purpose of updating attributes to provide immediate feedback to programmers about static semantic errors and to incrementally generate the object code needed to test the program. With this understanding, the editors could, for example, separate error detection from error reporting according to whether the programmer is making many changes or only one; in the first case, the programmer is unlikely to want to hear about errors after every keystroke.

This knowledge could be added to the Synthesizer Generator and the language-based editors it produces *via* rules that model the part of the software development process relevant to program editing. One rule might be that a ‘programmer’ object can modify the parse tree represented by a ‘program’ object. A second rule might state that the editor has an obligation to notify the ‘programmer’ of any errors in the program; another might say that a precondition for a ‘programmer’ to resume execution of ‘program’ is that no substantive changes have been made to any ‘procedure’ already on the run-time stack. Adding this kind of knowledge to the Synthesizer Generator would make its editors (relatively) more intelligent. For example, they could then simulate attribute re-evaluation at appropriate points to obtain insight into the consequences of the programmer’s actions and warn the programmer about changes that invalidate the internal execution state of the debugger.

References:

1. S. I. Feldman, ‘‘Make — A Program for Maintaining Computer Programs’’, *Software — Practice & Experience*,Vol. 9No. 4April 1979, pp. 255-265.

2. Tim Teitelbaum and Thomas Reps, ‘‘The Cornell Program Synthesizer: A Syntax-Directed Programming Environment’’, *Communications of the ACM*,Vol. 24No.

9September 1981, Reprinted in David R. Barstow, Howard E. Shrobe, and Erik Sandewall (Eds.), *Interactive Programming Environments*, McGraw-Hill Book Co., New York, 1984.

3. Thomas Reps and Tim Teitelbaum, ‘‘The Synthesizer Generator’’, *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, April 1984, pp. 41-48, Proceedings published as *SIGPLAN Notices*, vol. 19, no. 5, May 1984.

Intelligent Assistance for Software Development and Maintenance

Gail E. Kaiser
Columbia University
Department of Computer Science
New York, NY 10027

Peter H. Feiler
Steven S. Popovich
Carnegie Mellon University
Software Engineering Institute
Pittsburgh, PA 15213

Final version to appear in **IEEE Software**, May 1988.
5 June 1987

Overview

This article presents an architecture for *controlled automation* in software development environments. Controlled automation enables environments to behave as intelligent assistants by answering questions about the software project and automatically invoking tools to further the users' goal of producing a working software system. The discussion of the architecture focuses primarily on the programming stages of development and maintenance. An environment assists programmers by understanding the technical aspects of the evolving software system and by actively participating in the programming process. The architecture supports these capabilities by providing two kinds of knowledge representation: (1) the knowledge specific to a particular software project is represented as entities in a database and (2) the knowledge that models programming activities in general is represented as rules amenable to forward and backward chaining. These rules enable an environment to automatically carry out each activity sometime between when its conditions are satisfied and its results are required. The rules are grouped into collections called *strategies*. One or more specific strategies are employed according to each user's current context and goals, and determine when forward or backward chaining should be applied and which rules are considered during chaining. This architecture has been validated through a prototype implementation that models the capabilities of an existing environment that supports automation.

1. Introduction

In 1973, Winograd[1] discussed his dream of an *intelligent assistant* for programmers. He proposed that the most fundamental requirement for an intelligent assistant is that it understand what it does, that is, it should be based on an explicit model of the programming world. Winograd described an imaginary programming environment, **A**, that would assist programmers by providing early error checking, answering questions about the program and the interactions among program parts, handling trivial programming problems, and automating simple debugging tasks.

This article presents an architecture for intelligent assistance that combines results of software engineering and artificial intelligence research, briefly surveyed in Box A. Software engineering provides experience building and using particular software development tools and environments in the context of specific software development processes. Artificial intelligence provides suitable structures for representing knowledge about software entities and the role of tools in the

software development process. The architecture supports environments that incorporate certain understanding of the developed software systems and of the usage of tools for producing the software. In particular, they handle the first two duties described by Winograd: early error checking and answering queries about programs. The environments aid not only individual programmers, but assist in the coordination of multiple programmer teams. The architecture has not yet been applied to other aspects of software projects, such as project management or life cycle support, which are handled by some project support environments.[2]

The architecture supports two basic aspects of an intelligent assistant: to provide *insight* into the software system and to actively participate through *opportunistic processing*. Insight is the capability of an environment to be aware of users' activities and to anticipate the consequences of these activities based on an understanding of the development process and the target software system. Insight assists individual programmers by permitting them to become informed more quickly about the structure and relationships in the target system, to be aware of the consequences and side effects of their tasks, and to be guided in the job of making even major changes to a system and getting it back into a consistent state. Insight also assists in coordinating the activities of multiple programmers so they can accomplish their tasks without interfering with each other, knowing that the results of simultaneous work will be combined in a controlled way.

Opportunistic processing is the capability of an environment to undertake simple software development activities, so that programmers need not be bothered with them. This processing must be carried out without interfering with the users. The architecture automates only menial activities, such as determining when the source code has changed, invoking the compiler, and recording errors found during compilation. It performs an activity when the opportunity arises, *i.e.*, at an appropriate time between the time a user activity causes additional processing and the time the user requests the results of an activity; thus the term 'opportunistic' processing. This is in contrast to some intelligent assistants such as the Programmer's Apprentice (also known as KBEmacs[3]) and CHI[4], which focus on automatic programming.

The architecture has been validated through a prototype implementation that is capable of capturing the intelligent assistance embedded in an existing software development environment, called SMILE. We call the environment that embodies our architecture PROFESSOR MARVEL, or MARVEL for short, because Professor Marvel was the (Kansas) name of 'the man behind the

curtain' in the movie *The Wizard of Oz*. The analogy is from the ability of MARVEL to produce impressive results using relatively simple technology. SMILE is described in Box B, while the prototype implementation of MARVEL is discussed in Box C. A more elaborate implementation, which will permit the extension of the MARVEL concepts to non-programming activities, is underway.

The rest of this article explains how the architecture combines the software engineering and artificial intelligence approaches to support intelligent assistance for software development and maintenance. Section 2 gives an overview of the architecture. Section 3 describes the assistance provided by the insight mechanisms while section 4 demonstrates how the opportunistic processing mechanisms support active participation. A short sample session script illustrates the use of the MARVEL. The conclusions summarize the contributions of this research.

2. An Architecture for Intelligent Assistance

According to Winograd, the distinguishing feature of an intelligent assistant is that it understands what it does. There is a spectrum, however, to artificially intelligent systems. Most software tools can be thought of as moronic assistants that know only what to do and do not understand the purpose of the objects they manipulate or how their tasks fit into the software development process. In other words, they know the 'how', but do not understand the 'why'.

A software development environment cannot understand why it performs particular activities unless it knows

- the properties of the objects it manipulates,
- tools and activities, and the objects they manipulate,
- the preconditions under which a tool or activity can be activated,
- the results or postconditions of each activity, *i.e.*, the state of development after termination of an activity.

Therefore, the architecture consists of two key components. The first is a database, where each datum is represented as an object in the sense of object-oriented languages. This *objectbase* maintains all entities that are part of the evolving software system, all information about the history and current status of the project, and all tools used during software development and maintenance. The objectbase defines the various classes of objects and the relationships among objects, such as one object is a component of another and a particular object may be applied to another object to produce a third object. The objectbase is active in the sense that accessing objects may trigger certain actions.

The second key aspect is a *model* of the software development process that imposes a structure on programming activities. The model consists of an extensible collection of rules that specify the particular conditions that must exist for particular tools to be applied to particular software objects. Some rules are relevant only when a tool is invoked by a user, others apply when tool processing is initiated by the environment, while some rules apply equally to both cases. Forward and backward chaining permit the environment to perform activities automatically when it knows the results of these activities will soon be required by its users. The article describes mechanisms that enable each user to select the automation of desired activities and control undesired processing.

Box D illustrates the potential for intelligence assistance by describing a thought experiment where an objectbase and a model enhance the capabilities of two well-known programming tools. Rather than add intelligence to specific tools individually, the goal of the architecture is to encapsulate all the intelligence in the environment, so that it is not necessary to modify the tools. The following two sections explain the mechanisms that permit the MARVEL environment to exhibit insight and opportunistic processing, respectively, and thus intelligently assist its users.

3. Providing Insight

The two key elements to supporting insight are a rich, structured information repository and mechanisms to make available appropriate information at appropriate times. The information repository is represented as an objectbase. The access mechanisms fall into two categories, those that support direct access, or browsing, and those that support retrieval, *i.e.*, queries.

3.1. An Objectbase

MARVEL's objectbase is conceptually related to object-oriented programming languages,[5] in that each object is an instance of a *class*, which defines the type of the object. The objectbase contains a set of software objects, which represent both the system and its history of development. Object types include module, procedure, type, design description, user manual, development step, *etc.* Typing permits MARVEL to provide an object-oriented user interface, meaning the environment makes available only those commands that are relevant to the object under consideration within the context of the user's recent activities.

Unlike most object-oriented languages, however, the objectbase is 'persistent', meaning it retains its state across invocations of the environment. This enables MARVEL to provide a

‘fileless environment’; that is, it exposes its users only to the logical entities comprising the target software system, not to the physical storage of the software in terms of directories and files. Similar capabilities are found in the database support for other knowledge-based environments.[6]

Each class defines certain properties of an object and *inherits* other properties from its superclass(es). Some properties, called *attributes*, define the contents and status of objects. The software development activities applicable to the instances of a class are defined by a second category of properties, called *methods*. The values of attributes may be simple values (integers, strings, *etc.*) or they may represent relationships with other software objects. Simple attribute values include names of objects, status information about objects such as whether a software object has been analyzed for static semantic errors and the analysis was successful, or string entities such as pieces of source text or binary object code. Some of the relations represent the logical (syntactic) structure; for example, a module is composed of procedures, types, and variables. Other relations express various types of semantic dependencies, such as intended use — indicated by the import clauses of modules — or actual use, as demonstrated by the invocation of a procedure. Relations are *bidirectional* by default, permitting the information to be queried more flexibly, *e.g.*, a user can ask for all uses of procedure *p* as well as all uses of other procedures by procedure *p*. All information about objects is maintained in the objectbase, and inferred or derived by MARVEL where possible: users are spared the tedium of entering redundant information.

3.2. Accessing Information

Information in the objectbase is accessed for two purposes: for viewing and querying, and for modification. Both users and tools may access information. Users generally modify the structural hierarchy, the names of objects, and source text attributes; they do this through particular *views* of the objectbase. A view is defined as the subset of the information in the objectbase relevant to the current goals. Other attributes (*e.g.*, analysis status or use relationships) are maintained by tools to reflect the current state of the target software system. Users can also benefit from browsing and querying this auxiliary information.

3.3. Browsing

Browsing takes place according to views. The logical structure of the target system provides a natural default view. For example, the user sees program libraries containing modules, which in turn contain other modules or indivisible software components, *i.e.*, procedures, types, variables, *etc.* The user can navigate through this structural hierarchy the same way users navigate through directory structures in file systems. However, limited bandwidth prohibits exposing users to the complete structure at one time (unless very small fonts are used!), which may be undesirable in any case because of the problem of information overload.

Views can be displayed and browsed in a variety of ways. MARVEL procides a style of browsing where objects and subparts of objects can have selectable textual representations. Selection of such an entity specifies the current *focus of attention*. Processing and command selection are sensitive to the current focus, presenting the user with an object-oriented interface.

MARVEL tries to balance the amount of information presented to the user. One view displays only a single level of the structural hierarchy. A selected object can be edited, *e.g.*, the source code attribute of a procedure object or the proper name of an object, and can be opened for viewing if the component represents a reference to another object. The newly opened object can be viewed in the current window, or in a separate window. Another view shows multiple levels of the hierarchy at once. This permits MARVEL to respond to user requests for more context information, reducing the need for repeated user queries or browsing operations. For example, when viewing the content of a module, the view contains the names of the component objects as well as their type, *i.e.*, whether they are procedures or documents. Similarly, provision of visual feedback of values for certain essential attributes, *e.g.*, whether a module contains errors, eliminates additional queries by the user while still avoiding information overload.

MARVEL permits the user to gain insights not only by navigating through the primary logical structure, *e.g.*, the library-module-component hierarchy, but also by following other cross-reference links such as opening to the specification of a module referenced in the import list of another module. Such cross-link browsing capabilities permit the user to get an impression of the context of a piece of software with reduced effort.

In summary, the browsing capability allows the user to manually navigate through the objectbase, changing the focus of attention. This permits MARVEL to track user actions, anticipate the consequences, and assist him to cope with them. However, manual navigation is inadequate

for general search tasks. For example, if the user maintains a system with 150 modules, trying to find the three modules with outstanding errors can be a tedious task if done by browsing. A general querying capability combined with a browsing capability solves this problem.

3.4. Queries

A general question answering capability supports searches of the objectbase for software objects based on search conditions. Examples include 'retrieve all software objects with proper name x' or 'retrieve all modules that contain errors' (phrased in a stylized command language). The search space can be constrained in several ways. One way is through particular search conditions, such as objects of a particular type, or attributes having particular values. Other constraints are expressed in terms of the primary logical structure by indicating the search to be limited to a particular substructure, *e.g.*, search of a procedure in a particular library. MARVEL also prunes the search space by utilizing dependency information such as import and actual uses of procedures.

Queries may be either explicit or implicit. *Explicit queries* correspond to queries posed by a user. MARVEL predefines short forms for a set of common queries. Examples of such queries are: 'what components use a particular function', 'are certain components not used at all' (useful during maintenance or cleanup), 'which components/modules have errors', 'which components have a particular error', and 'is anybody else intending to modify or actually modifying a particular component/module'. Such queries allow the user to get an impression of the structure and connectivity of the software to be modified or maintained.

Implicit queries are initiated by MARVEL. There are several reasons for it to do so. The environment may have encountered an exceptional condition and generated a query to find essential information in an attempt to repair the problem. For example, if the user requests to edit procedure *p*, but procedure *p* is not within the current focus (*i.e.*, the current module), MARVEL queries the objectbase for a procedure named *p*. If the query returns a unique element MARVEL can change the focus; if there are choices, it asks the user to make the selection.

A second reason for implicit queries is that the result of the query may be automatically presented to the user. An example is the provision of change simulation. When a user gives the command to edit the specification of a module component that is being exported, MARVEL informs the user of the expected extent of the consequences and requests confirmation to go ahead

with the editing. The same information can in turn be used to check whether the affected components are accessible for modification (*i.e.*, have been reserved by the particular user). The result of this query can again be presented to the user, or MARVEL can go ahead and attempt a reservation and/or add new editing tasks to the user's agenda (see section 4). Implicit queries are made when the result of the query provides insight into expected activities, making the user aware of the potential consequences of his actions during development and maintenance.

3.5. Summary

- The objectbase maintains both the structural relationships among objects and additional relationships defined by the tools provided by the environment.
- Multiple views of the objectbase support browsing and queries according to appropriate abstractions of the information in the objectbase.
- Multiple views are also available to tools, and support implicit queries by the environment to anticipate the consequences of user actions.

4. Supporting Opportunistic Processing

In addition to software objects, the objectbase also maintains the meta-knowledge MARVEL uses to know when to apply sequences of tools on behalf of the users. This meta-knowledge is defined as an extensible collection of rules consisting of a precondition, an activity, and multiple postconditions. Forward chaining allows the assistant to invoke tools as soon as their preconditions are satisfied, *i.e.*, the earliest point of activation, and backward chaining makes it possible to find the tools whose postconditions satisfy the preconditions of other tools whose activation is requested, *i.e.*, the latest point of activation. Strategies tailor this controlled automation to the needs of each particular user; for example, MARVEL may perform different functions for an long-term user than for a novice. It is important to realize that the rules and strategies are written by an *environment maintainer*; individual users see the effects of such descriptions in the choice of available strategies and in the resulting behavior of the environment in providing active assistance.

4.1. Rules

MARVEL rules are based on production rules, *i.e.*, condition/action pairs. When the condition is true, or satisfied, then the action is applied to working memory (in this case, the objectbase). However, production rules are inadequate, because it is necessary to separate the invocation of a tool from the results produced by the tool in order to integrate existing tools without modifica-

tion. Therefore, the action is divided into two parts, an activity and a postcondition. Because rules have postconditions, the original conditions are called ‘preconditions’.

The *activity* part of a rule represents an integral software development task. For example, "compile module" is one activity and "edit procedure" is another. Low-level editing commands (delete character, move cursor, and so on) applied during the course of the "edit procedure" activity are not themselves considered activities. High-level goals such as ‘fix bug’ are not activities, since they involve many tasks and perhaps several users. Thus an activity represents a middle-ground in granularity.

Each activity is associated in the objectbase with a tool that actually carries out the activity. One attribute of each tool is whether or not it can be invoked implicitly by the environment without human intervention. For example, the "compile module" activity is associated with the compiler, which can be applied automatically; the "edit procedure" activity is associated with a text editor (or a syntax-directed editor), and an attribute of the editor indicates that editing requires human interaction.

The *precondition* part of a rule is a boolean expression that must be true before an activity can be performed. The operands of a precondition include software objects and their attributes. For example, "notcompiled(module)" might be an appropriate precondition for the "compile module" activity. If static semantic analysis is separated from code generation, another predicate for the same activity would be "for all components c such that in(module, component c): analyzed(component c)", where "analyzed(c)" is true only if the analysis of component *c* (a procedure, a type, a variable declaration, *etc.*) did not find any errors.

A *postcondition* is an assertion that becomes true when an activity is completed. Both preconditions and postconditions are written as well-formed formulas (wffs) in the first order predicate calculus. Thus rules are syntactically similar to Hoare’s assertions,[7] where a programming language construct is associated with its preconditions and postconditions; if the preconditions are true before the language construct is executed, then the postconditions will be true afterwards. The semantics of MARVEL’s postconditions differ from Hoare’s in that the purpose of the postcondition is to update the objectbase rather than to be verified. Further, a programming activity may result in any one of multiple postconditions. The choice is determined by the result of the tool invocation, and the chosen postcondition is asserted. For example, two mutually exclusive postconditions for the "compile module" activity might be "compiled(module)" and "errors(module)". The "compile" rule describing the above properties is given in figure 4.1.

```
notcompiled(module) and
   for all components c such that in(module, component c):
       analyzed(component c)
    { compile module }
compiled(module) |
errors(module);
```

Figure 4-1: Compile Rule and Edit Rule

4.2. Forward and Backward Chaining

Forward and backward chaining contribute to opportunistic processing by enabling MARVEL to use the rules to determine what needs to be done and what can be done automatically. Forward chaining implies that if the preconditions of an activity are satisfied, and the activity is one that the assistant is capable of performing, then MARVEL may perform the activity without human intervention. Forward chaining supports behavior familiar from language-oriented editors.[8] When the user makes a subtree replacement in the abstract syntax tree representing the program, the editor automatically performs various actions, such as type checking or code generation for the modified part of the program.

Consider again the rule in figure 4.1. MARVEL uses forward chaining to interpret this rule to mean that the assistant may compile any and all modules *M* such that all the components of *M* have been successfully analyzed but *M* has not yet been compiled. One of the preconditions is "notcompiled(module)" of the "compile module" activity and "errors(module)" is included as one of the possible postconditions. If a module has been previously compiled unsuccessfully, "errors(module)" will be true. The "compile module" activity cannot be performed when "errors(module)" is true, because its preconditions cannot be satisfied. If a user then edits a component, perhaps to fix the error, a rule representing the edit activity will set "notcompiled(module)" to be true, permitting compilation to be tried again.

Forward chaining implies that MARVEL <u>may</u> perform an activity when its preconditions are satisfied. The assistant is not required to perform the activity as soon as the preconditions are true, or at any particular time thereafter. However, it may go ahead and apply the tool, and use forward chaining to determine additional activities whose preconditions are now satisfied as a result of generating the postconditions of the first activity. The following section explains how MARVEL decides when to apply forward chaining.

Backward chaining implies that if a user invokes a tool whose preconditions are not satisfied, MARVEL should find other activities it can perform whose postconditions might satisfy the preconditions of the activity requested by the user. Backward chaining requires that the assistant exhibit this behavior. The behavior supported by backward chaining is behavior familiar from Make[9]. When a user requests regeneration of an executable system after changes have been made to its source code, the tool uses dependency information previously supplied by the software development team to determine which source files must be recompiled. Of course, it may not be possible to satisfy all the preconditions, and in this case the user is informed of the problem; MARVEL is not expected to, for example, find and repair bugs. In general, MARVEL will not automatically perform activities that invoke tools requiring human intervention.

Consider the case where MARVEL supports a large programming team where multiple users are not permitted to change the same module at the same time. This might be handled by requiring each user to reserve a module before changing it. The preconditions and postconditions for the "reserve module" activity are stated in the first rule shown in figure 4-2 ("saved(module)" is true when the module has been saved by the version control tool). The second rule states that the "change component" activity cannot be carried out unless the module that contains the component is reserved.

```
not reserved(module) and saved(module)
    { reserve module }
reserved(module, userid);

reserved(module, userid)
    { change component }
notanalyzed(component) and notcompiled(module);

for all components k such that in(module, component k)
    and uses(component k, component c):
        reserved(module, userid)
    { change component c }
```

Figure 4-2: Change Rules and Reserve Rule

The "change component" activity permits the user to modify the specification of a component ("edit component" permits the user to modify only the body). The third rule of figure 4-2 states that not only should the containing module be reserved, but it is necessary to reserve any other modules whose components use the component that will be changed (the names *c* and *k* are used to distinguish multiple objects of the same type). Backward chaining makes it possible for

MARVEL to automatically reserve any modules whose components may have to be modified in order to restore consistency with the changed component. It also prevents the user from modifying the specification of a component when these other modules cannot be reserved (according to the first rule).

4.3. Employing Strategies

When MARVEL attempts to assist the user by performing opportunistic processing it must wisely choose the degree of automation. It does so by selecting appropriate points on the spectrum between the earliest and latest time an activity can be performed automatically, and by disabling certain automatic processing when it gets in a user's way; in other words, MARVEL must dynamically adapt to the current user's goals. This problem is remedied by including hints and strategies in the architecture to aid the assistant in making these decisions.

A *hint* is similar to a rule, except there are no postconditions. The preconditions of a hint are used to guide the assistant in choosing when to apply a tool whose other preconditions are satisfied. Consider again the rule from figure 4.1. Suppose MARVEL should not automatically recompile a module, even though the preconditions are satisfied, as long as a user who has modification rights is browsing through the module. The rationale is that the user may soon decide to edit some components of the module, and then the generation of code will have been wasted. This is captured in a hint, shown in figure 4-3, giving this precondition for the "compile module" activity (angle brackets are used for parentheses in both rules and hints). When the assistant follows a strategy including this hint, compilation is delayed until the user changes his focus to another module.

```
not reserved(module) or
    < reserved(module, userid) and
      not equals(module, focus(userid)) >
    { compile module }
```

Figure 4-3: Compile Hint

Of course, the user must be able to invoke the compiler without changing focus to another module. This is why this precondition to "compile module" is stated as a hint, rather than as part of a rule. Hints apply only to the opportunistic processing of the environment, not to user-initiated activities. In other words, hints are considered during forward chaining and ignored during backward chaining.

A *strategy* consists of a collection of hints and rules. These hints and rules apply only when the strategy is in force. MARVEL employs strategies by combining their rules and hints. One or more strategies may be employed at the same time. When this results in more than one rule for the same activity, all their preconditions must be satisfied, but only one member of the set of postconditions may be asserted.

MARVEL cannot choose its own strategies. Instead, each user selects the appropriate strategies by informing the environment that he is, for example, a manager *vs.* a programmer, developing a new software system *vs.* maintaining an old software system, or making major change *vs.* making a minor revision. For example, a strategy whose rules and hints result in automatically performing type checking immediately after each component is edited would be appropriate for a minor revision, but not for a large scale change involving many interrelated components. MARVEL has predefined strategies for these cases; these strategies can be modified or replaced and new strategies can be added by an environment maintainer tailoring the environment. In general, individual users would not modify the rules or strategies.

4.4. Activities as Side-Effects

Often a tool has the effect of performing additional activities as side-effects. For example, the analysis tool invoked for the "analyze component" activity may change the values of several attributes of software components. Setting the value of an attribute is considered an activity, resulting in a situation where one action of MARVEL is embedded inside another rather than being a consequence of forward or backward chaining. This case demonstrates a limitation of MARVEL's rules: secondary actions whose arguments are not simple derivatives of the arguments of the preconditions or the activity cannot easily be expressed as postconditions. Instead, potential side-effects are indicated by attributes of the tool. In such cases, the secondary activities are often described by their own rules, and these must be considered for further processing. For example, some rules related to the "uses" attribute of a component are given in figure 4-4. The "uses" attribute lists the other components that the component depends on.

The first rule gives the obvious preconditions and postconditions for the "analyze component" activity. The second rule in figure 4-4 states a component *c* cannot use another component *k* unless component *k* is in the same module or is imported into the module. The third rule means that a component cannot be imported by a module *M* unless it is exported by another module *N*. The fourth rule states that a component cannot be exported by a module unless it is in that module.

```
notanalyzed(component)
    { analyze component }
analyzed(component) |
errors(component);

in(module, component c) and
    < in(module, component k) or imports(module, component k) >
    { component c uses component k }
uses(component c, component k);

exports(module N, component) and
    not equal(module M, module N)
    { import component }
imports(module M, component);

in(module, component)
    { export component }
exports(module, component);
```

Figure 4-4: Analyze Rule, Uses Rule and Import/Export Rules

Consider what happens when the analysis tool finds that procedure *p* (a component) calls procedure *q* (another component) and tries to set the "uses" attribute of procedure *p* to include procedure *q*. If *q* is in the same module as *p*, there is no problem; the attribute is set and the analysis continues. If *q* is not in the same module, MARVEL checks whether it is imported. In the case where *q* is not already imported, the assistant notes that "imports(module, component)" is a postcondition of the "import component" activity (third rule) and further realizes it can perform the "import component" activity. So it considers the preconditions of this activity. MARVEL queries its objectbase to find the module that does contain *q*. If *q* is already exported from this module, the assistant performs the "import component" activity. If not, backward chaining permits the assistant to follow the preconditions of this activity given in the fourth rule of figure 4-4. The assistant can add *q* to the exports of its module, then actually import *q* into the original module, and then finally permit the analysis tool to set the "uses" attribute of *p*.

This is only one possible strategy, which ignores the possibility that distinct procedures named *q* might be found in more than one module. Sometimes language-specific typing information can be used to narrow down the possibilities, but in general MARVEL must interrupt the user to explain its dilemma and to ask which *q* is intended. The assistant can then proceed as described in the previous paragraph.

There is one more possibility: there is no component named *q* in the objectbase. Therefore,

MARVEL considers the "add component q" activity, whose postcondition is of course the existence of *q*. If permitted by the current strategy, MARVEL could carry out this activity on its own, by creating a stub for the procedure within the module where the use occurs. Alternatively, MARVEL could ask the user to create the procedure (or its stub) before continuing the analysis, but this might be intrusive; a preferred alternative is to inform the analysis tool of the problem and prevent it from performing the "procedure p uses procedure q" activity. This causes the analysis tool to terminate unsuccessfully, generating the "errors(p)" predicate among its postconditions.

In this discussion, "import component" and "export component" are among the activities that do not require human interaction, permitting MARVEL to carry out the repairs illustrated by the example. An alternative strategy requires the assistant to take the imports and exports as given. This might be appropriate for languages, such as Ada*, that include their own module constructs, where reference to an external component without the appropriate 'with' clause should be detected as an error. A second alternative would require MARVEL to ask the programmer whether or not *q* is a typographical error before carrying out all the previously described actions.

Over time the modular structure of software systems degenerates. For systems written in languages such as Ada with explicit export/import declarations, the number of these declarations tends to be monotonically increasing, even though some imported components are no longer used. The assistant can support the maintenance of such 'old code' by providing both rigid and flexible strategies in the same environment. Flexible strategies permit the assistant to reflect the actual usage of components automatically in the export/import lists, *e.g.*, remove unnecessary exports/imports or adjust exports/imports as the code is being reorganized by the user. Rigid strategies provide stability during development phases such as testing and integration by taking the export/import declarations as givens to be checked against.

4.5. Implicit Queries

In the example of figure 4-4, MARVEL automatically queried its objectbase to locate procedure *q*. When the environment performs a query on its own, rather than in response to a user command, we call this an *implicit query*. Implicit queries are necessary to determine whether the preconditions of rules and hints are satisfied and to find the next rules to be applied in forward and backward chaining.

---

*Ada is a trademark of the U.S. Department of Defense Ada Joint Projects Office.

Another application for implicit queries is to anticipate the postconditions of activities. This enables MARVEL to notify the user in a timely manner when a user action is likely to lead to adverse results. Consider the two rules shown in figure 4-5. Through forward chaining, changing a component will lead to semantic analysis, which may result in errors. When a user invokes the editor on a particular component, MARVEL anticipates this forward chaining and notes the possible "errors(component)" postcondition. This causes it to implicitly query its objectbase to determine likely potential error sites.

```
reserved(module, userid)
    { change component }
notanalyzed(component) and notcompiled(module);

notanalyzed(component)
    { analyze component }
analyzed(component) |
errors(component);
```

Figure 4-5: Change and Analyze Rules

Of course, MARVEL cannot guess what modifications the user will make and how these will affect other components. However, it can take advantage of the "used-by" attribute to determine those components most likely to be affected. Both the "used-by" attribute and its inverse ("uses") are listed in the objectbase among the potential side-effects of the editor tool. (Note that the granularity of dependencies can be more refined if editing is done with a syntax-directed editor rather than a text editor.) The environment informs the user of the potential sources of semantic inconsistencies by presenting the list of components given by "used-by" attribute of the component argument to the editor. The user can take this information into account and choose whether or not to abort his "change component" command.

Implicit queries were also evident in the example of figure 4-2. A user gave the "change component" command. Backward chaining led the assistant to query the objectbase to determine whether all the modules affected by the proposed change were reserved by this user. If not, MARVEL would attempt to automatically reserve all the necessary modules. If some of these modules are reserved by other users, the assistant would present the results of its implicit queries to the user to explain why the requested activity is not permitted.

4.6. Summary

- Rules define the preconditions that must be satisfied before a tool can be applied and the alternative postconditions of each tool.
- Hints define the preconditions that must be satisfied before a tool can be automatically applied by the environment; unlike rules, hints do not affect the activities of users.
- The architecture supports forward chaining from the postconditions of completed activities to the preconditions of other tools and backward chaining from the preconditions of desired activities to the postconditions of other tools, respectively.
- Tools may have side-effects that cannot be directly expressed as postconditions, but these are nevertheless considered with respect to forward and backward chaining.
- The environment performs implicit queries to determine the attributes of software objects and the potential side-effects of tools.
- Strategies group together rules and hints that are appropriate for particular users and for particular phases of software development and maintenance. The architecture enables the assistant to employ particular strategies during forward and backward chaining.

5. A Sample Session

Figure 5-1 shows a sample session in MARVEL. It represents a snapshot of the workstation screen after the user has been interacting with MARVEL and is in the middle of editing a particular procedure. The screen shows two windows. In the large window the user interacts with the MARVEL command interpreter. It shows a transcript of the user's session sofar. The window is scrollable, *i.e.*, the complete transcript is accessible to the user. The smaller window in the right lower half is an edit window, which is brought up by MARVEL for the user to edit a particular item in the objectbase using his favorate editor, in this particular case Emacs. The bottom of the screen shows several icons that are part of the general X window view of the underlying Unix system.

The transcript in the large windows shows a series of interactions between the user and MARVEL. These interactions demonstrate some of the behavior of MARVEL. The beginning of the session the user enters an existing workspace to modify a software system, in this case an interactive program for fractional arithmetic. This workspace, a MARVEL database, is private to the user, and is connected to a public database, in which the baseline version of the software resides. At the time the user enters the workspace, one module has been reserved from the public database and made accessible for modification in the workspace. All other parts of the software system, which physically reside in the public database are transparently accessible for reading.

The user's focus of attention is placed on the system object representing the whole program, which is indicated by the prompt showing the system name. First, the user requests a particular view of the system, namely the list of modules composing the system. The user then changes the focus of attention to the module "BasicOps". Notice the change in the prompt. Next, another view is requested, in this case a more detailed view of a particular module. Since the user did not specify a module name, the system chose the module in the current focus of attention. This view shows all components of the module (several procedures and an object; the module does not contain data type definitions) as well as the list of components that are made available externally as part of the module specification in form of an export list. With the "change" command, the user next attempts to modify the specification of the procedure "quit". The system prompts for missing command parameters, providing defaults. MARVEL first performs an implicit query to determine the consequences of the planned change to a specification. The user is informed that the procedure "quit" is used by another module, for which the user currently does not have modification rights. Under the chosen strategy MARVEL does not automatically reserve the module, but aborts the command. The user explicitly reserves the module. MARVEL confirms that the module is to be reserved from the public database it is aware of. After the reservation the second modification attempt succeeds. The user is informed of potentially affected components before the actual editing, and asked upon completion of the modification whether the affected components should be analyzed and compiled as well. Since the user expects to correct the affected procedure he declines the offer. The modified component is analyzed and compiled in the background, while the user issues the "edit" command to make a local modification to procedure "ci". MARVEL changes the focus of attention to the appropriate module, displays the procedure specification, and presents the user with the procedure body in the editor window. This is the time when the screen snapshot is taken. The dialog shown in the session transcript can be adapted by the user through choice of different strategies to influence verboseness and automation by MARVEL.

6. Conclusions

This article presents an architecture for intelligent assistance consisting of an objectbase and a model of the software development process. The advantage of an objectbase is it permits the assistant to present a 'fileless environment', so its users are concerned only with the logical entities associated with software project and not with the details of the underlying file system and

Figure 5-1: Sample MARVEL Session

operating system. The advantages of an explicit model of software development are that it can constrain the relationships among the software objects to maintain consistency and can automate bookkeeping chores and other menial development activities.

Similar notions have previously been promoted by other researchers as the fundamental basis for a programming environment that understands what it is doing. The specific contribution of this research is the formalization of insight and opportunistic processing. Insight and opportunistic processing are made possible by

- maintaining all knowledge about the particular software development effort and the general software development process in the objectbase,
- multiple views of the objectbase, available both to users and to tools,

- modeling the software development process as rules that define the preconditions and alternative postconditions of software development activities,
- representing alternative collections of rules as strategies.

MARVEL's architecture combines knowledge with already available software development tools to produce software engineering environments that intelligently assist software development and maintenance efforts by individuals as well as teams of users.

Acknowledgements

Dave Ackley, Naser Barghouti, Susan Dart, Mark Dowson, Bob Ellison, David Garlan, Dan Miller, John Nestor, Gavin Oddy, Cecile Paris, Colin Tully, Nelson Weiderman, Ursula Wolz and the anonymous referees reviewed previous drafts of this article and made many useful criticisms and suggestions. Purvis Jackson assisted us with technical editing. The work reported in this article was started while Dr. Kaiser was a Visiting Computer Scientist at the Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA. The first prototype implementation was done at the Software Engineering Institute. Further research on MARVEL is currently continuing at Columbia University. Research on MARVEL is supported in part by Dr. Kaiser's DEC Faculty Award, in part by a grant from Siemens Research and Technology Laboratories, and in part by the Department of Defense.

References

1. Terry Winograd, ''Breaking the Complexity Barrier (Again)'', *SIGPLAN-SIGIR Interface Meeting on Programming Languages — Information Retrieval*, Gaithersburg, MD, November 1973, pp. 13-30, Reprinted in David R. Barstow, Howard E. Shrobe, and Erik Sandewall (Eds.), *Interactive Programming Environments*, McGraw-Hill Book Co., New York, 1984.

2. Mark Dowson, ''ISTAR — An Integrated Project Support Environment'', *2nd SIGSOFT/SIGPLAN Symposium on Practical Development Environments*, December 1986, pp. 27-33, Proceedings published as *SIGPLAN Notices*, vol. 22, no. 1, January 1987.

3. Richard C. Waters, ''KBEmacs: Where's the AI?'', *The AI Magazine*,Vol. VIINo. 1Spring 1986, pp. 47-56.

4. Douglas R. Smith, Gordon B. Kotik and Stephen J. Westfold, ''Research on Knowledge-Based Software Environments at Kestrel Institute'', *IEEE Transactions on Software Engineering*,Vol. SE-11No. 11November 1985, pp. 1278-1295.

5. *Object-Oriented Programming Systems, Languages and Applications*, Portland, OR, September 1986, Proceedings published as *SIGPLAN Notices*, vol. 21, no. 11, November 1986.

6. David S. Wile and Dennis G. Allard, ‘‘Worlds: an Organizing Structure for Object-Bases’’, *2nd SIGSOFT/SIGPLAN Symposium on Practical Development Environments*, December 1986, pp. 16-26, Proceedings published as *SIGPLAN Notices*, vol. 22, no. 1, January 1987.

7. C. A. R. Hoare, ‘‘An Axiomatic Approach to Computer Programming’’, *Communications of the ACM*,Vol. 12No. 10October 1969, pp. 576-580, 583.

8. Tim Teitelbaum and Thomas Reps, ‘‘The Cornell Program Synthesizer: A Syntax-Directed Programming Environment’’, *Communications of the ACM*,Vol. 24No. 9September 1981, Reprinted in David R. Barstow, Howard E. Shrobe, and Erik Sandewall (Eds.), *Interactive Programming Environments*, McGraw-Hill Book Co., New York, 1984.

9. S. I. Feldman, ‘‘Make — A Program for Maintaining Computer Programs’’, *Software — Practice & Experience*,Vol. 9No. 4April 1979, pp. 255-265.