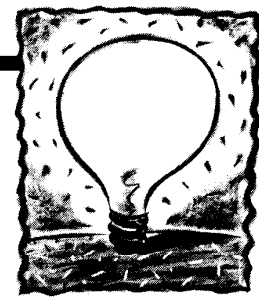

Intelligent Assistance for Software Development and Maintenance

Gail E. Kaiser, *Columbia University*
Peter H. Feiler, *Software Engineering Institute*
Steven S. Popovich, *Columbia University*

Using relatively simple technology, Marvel understands the user's actions and their consequences. In many cases it will do tasks automatically, lightening the workload.

In a 1973 article, Terry Winograd wrote of his dream of an intelligent assistant for programmers.[1] The fundamental requirement for an intelligent assistant, he wrote, 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 that would provide early error checking, answer questions about the program and the interactions among program parts, handle trivial programming problems, and automate simple debugging tasks.

We have developed an environment that handles the first two duties, early error checking and answering questions about programs. Our environment has a certain understanding of the systems being developed and how to use tools to produce software. It aids individual programmers and helps coordinate programmer teams.

Our assistant's knowledge is described in a model and achieves intelligence by interpreting the model. We have not yet applied the model of this environment to other project aspects, such as project management, which are handled by some integrated project support environments.

Our model draws from research into software engineering and artificial intelligence. From software-engineering research, we gained experience in building and using particular tools and environments in specific development processes. From artificial-intelligence research, we discovered suitable structures to represent knowledge about software entities and the role of tools in the development process.

The result is Professor Marvel — Marvel for short — an environment that supports two aspects of an intelligent assistant: It provides *insight* into the system and it actively participates in development through *opportunistic processing*. Like its fictional counterpart, the Kansas magician who turned out to be the wizard in *The*

0740-7459/88/0500/0040/$01.00 ©1988 IEEE

Wizard of Oz, Marvel can produce impressive results with relatively simple technology.

Marvel's roots and the prototype implementation is described in the box on p. 43. A more elaborate implementation that will extend Marvel's concepts to nonprogramming activities is under way.

Key components

To fulfill Winograd's fundamental requirement, an intelligent assistant must understands what it does. However, there is a spectrum of intelligent systems. Most software tools are moronic assistants that know what to do but do not understand the purpose of the objects they manipulate or how their tasks fit into the development process. In other words, they know the *how* but do not understand the *why*.

A development environment cannot understand why it performs an activity unless it knows

- the properties of the objects it manipulates,
- the system's tools and activities, and the objects they manipulate,
- the preconditions under which a tool or activity can be activated, and
- the results or postconditions of each activity (the state of development after an activity terminates).

Object base. Marvel has two key components. The first is a database that stores data represented as objects, as in object-oriented languages. This object base maintains all the entities that are part of the evolving system, all the information about the history and status of the project, and all the tools used in development and maintenance.

The object base defines the object classes and the relationships among objects (such as one object is a component of another and when applied to another object will produce a third). The object base is active: Accessing objects may trigger action.

Process model. The second key component is a model of the development process that imposes a structure on programming activities. The model is an extensible collection of rules that specify the conditions that must exist for particular tools to be applied to particular objects. Some rules are relevant only when a user invokes a tool, others apply when the environment initiates tool processing, and still others apply equally to both cases.

Interpretation through forward and backward chaining lets the environment perform activities automatically when it knows the results of these activities will soon be required by the user.

Most software tools are moronic assistants that know the how but do not understand the why.

Rather than add intelligence to individual tools, the model encapsulates all the intelligence in the environment, so it is not necessary to modify the tools. The box on p. 47 illustrates the potential for intelligent assistance by describing how an object base and a development model enhance two well-known programming tools.

Insight

Marvel has insight, which means it is aware of the user's activities and can anticipate the consequences of these activities based on an understanding of the development process and the produced software.

Insight lets individual programmers become informed more quickly about the structure and relationships in the software product, 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 helps coordinate 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.

The two key elements that support insight are a rich, structured information repository and a set of mechanisms that make appropriate information available at appropriate times. The information repository is the object base. The access mechanisms fall into two categories, those that support direct access or browsing, and those that support retrieval.

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

However, unlike most object-oriented languages, Marvel's object base is persistent: It retains its state across invocations of the environment. This lets Marvel provide a file-less environment. Marvel exposes its users only to the logical entities comprising the target system, not to the physical storage organization of directories and files. Other knowledge-based environments offer similar capabilities in their database support.[2]

Each class defines certain properties of an object and inherits other properties from its superclass or superclasses. Some properties, called attributes, define the contents and status of objects. Other properties, called methods, define the development activities applicable to the objects of a class. Attributes may be simple values (integers and strings) or they may represent relationships with other objects.

Simple attribute values include object names, object status (such as if it has been analyzed for static semantic errors), and string entities (such as pieces of source text or binary object code). Attributes that represent relationships include the logical, syntactic structure (for example, a module is composed of procedures, types, and variables), semantic dependencies (such as intended use — indicated by the import clauses of modules — or actual use as demonstrated by the invocation of a procedure). Relationships are bidirectional by default, which permits more flexible querying. 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 object base, and inferred or derived by Marvel where possible. Users are spared the tedium of entering redundant information.

Information access. Information in the object base is accessed for two reasons: (1) viewing and querying and (2) modification. Both users and tools may access information.

Users generally modify the structural hierarchy, the names of objects, and source-text attributes through a view of the object base. A view is the subset of information in the object base that is currently relevant. Other attributes (analysis status or use relationships) are maintained by tools to reflect the current state of the target system. Users can also browse and query this auxiliary information.

Browsing. Browsing takes place according to views. The default view is the logical structure (the library-module-component hierarchy) of the target system. For example, the user sees program libraries containing modules, which in turn contain other modules or indivisible components (procedures, types, variables, and so on).

The user navigates through this structural hierarchy just as he navigates through directory structures in file systems. However, limited bandwidth prohibits exposing the user to the complete structure at once (unless we use very small fonts!), which is generally all right in any case because of information overload.

Views can be displayed and browsed many ways. In Marvel, objects and their parts have selectable textual representations. By selecting such an entity, the user specifies the current focus and by doing so determines processing and command selection. Hence, Marvel has an object-oriented interface.

Marvel tries to balance the amount of information presented to the user. One view displays a single level of the structural hierarchy. If the user selects an object to edit, it 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 another window.

Another view shows multiple levels of the hierarchy at once. This lets Marvel respond to user requests for more context information, reducing the need for repeated user queries or browsing operations. For example, a view of a module's content contains the names of the component objects and their type (whether they are procedures or documents). Similarly, Marvel provides visual feedback of values for certain essential attributes (if a module contains an error, for example), thus eliminating additional queries while still avoiding information overload.

Marvel also lets the user navigate by following cross-references, such as opening the specification of a module referenced in the import list of another module. Such cross-link browsing capabilities make it easier for the user to get an impression of the context of a piece of software.

In summary, the browsing capability lets the user manually navigate through the object base, changing the focus. This lets Marvel track user actions, anticipate consequences, and help the user cope with the consequences. 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.

Queries. A general answering capability supports searches of the object base according to conditions phrased in a stylized command language: "Retrieve all software objects with proper name *x*," for example, or "Retrieve all modules that contain errors."

The search space can be constrained several ways. One way is through particular search conditions, such as by object type or attribute value. Another way is to limit the the search to a particular substructure, such as searching a procedure in a particular library. Marvel also prunes the search space by using dependency information, such as import and actual procedure use.

Queries may be explicit or implicit. Explicit queries are initiated by the user. Marvel has predefined, short forms of common queries, such as:

- What components use a particular function?
- Are certain components not used at all? (Useful during maintenance and cleanup.)
- Which components (or modules) have errors?
- Which components have a particular error? and
- Is anybody else intending to or modifying a particular component (or module)?

Such queries let the user get an impression of the structure and connectivity of the software to be modified or maintained.

Implicit queries are initiated by Marvel for several reasons. It does so when it encounters an exceptional condition and needs essential information to repair the problem. For example, if the user wants to edit procedure *p*, but procedure *p* is not in the module currently in focus, Marvel queries the object base for a procedure named *p*. If the query returns a unique element, Marvel can change the focus; if there are many procedures named *p*, Marvel asks the user to choose one.

A second reason for Marvel to generate implicit queries is to present a query result

Marvel: Past, present, and future

Marvel's concepts are based on our experience with another environment that provided assistance to users. We extracted the properties that made that environment an active assistant into Marvel's model.

The concepts of this model have been validated through a first prototype implementation, based on the earlier environment, that supports the rules and strategies. This prototype has been followed by an implementation with full object base support and dynamic extension of the object base structure and the set of rules and strategies.

Marvel's ancestry. In the late 1970s and early 1980s, we and other members of the Gandalf project developed a multiuser, software-engineering environment called Smile.[1] Smile, which supports programming in C and runs on Unix, has been used on the Gandalf[2] and Gnome[3] projects at Carnegie Mellon University and by the Inscape project[4] at AT&T Bell Laboratories, and has been distributed to at least 40 sites.

Smile passes the crucial test of supporting its own maintenance. It has supported the simultaneous activities of seven to 10 programmers. The largest system developed and maintained in Smile has about 61,000 lines of source code.

Smile is a relatively intelligent assistance. It supports insight and opportunistic processing. It provides a file-less environment to its users, answers queries, coordinates the activities of multiple programmers, and automatically invokes tools. It hides the particulars of the Unix file system and utilities and presents its own model of the programming world. Smile's object base 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 hard-coded into the environment.

Marvel's proof of concept. We chose first to validate Marvel's concept of rules and strategies. We started with Smile for the prototype implementation. This lets us concentrate on the implementation of the rule-processing facility with minimal extensions to Smile's simple object base, yet still gave us an operational environment prototype. It also let us compare the prototype with the original Smile system, which has been in use for several years.

This implementation of Marvel replaced Smile's hard-coded knowledge about the software-development process with rules. Rules and strategies are written using a text editor, and the text file is parsed by a rule compiler.

The rule compiler translates rule preconditions and postconditions 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 link these predicates and relations to each relevant rule.

A rule set and strategy can be loaded at start-up and additional strategies can be loaded later, but there is no checking among simultaneously used strategies. Individual rules can be separately turned on and off.

Forward chaining, backward chaining, and the abillity 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. To support the rule interpretation, we added some attributes and relations to Smile's hard-coded object base.

The performance resulting from this simple-minded approach is unacceptable for large numbers of rules and large object bases, but was satisfactory for processing the rules describing Smile's behavior. 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 object base 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 hard-coded knowledge. The working prototype provided us with feedback for improvement in a number of areas. These were taken into account in a second implementation of Marvel.

Looking into the future. After the concept prototype of Marvel was completed at SEI, an implementation of Marvel that is independent of the Smile implementations was begun.

One version of this implementation is operational. It includes enhancement of the rule interpreter and an extensible object base. In this implementation, the rule interpreter supports consistency checking and merging of strategies as they are loaded dynamically, as well as dynamic unloading of strategies. The new object base supports object-class hierarchies and dynamic extensibility of structures stored in the object base. We have published details of this object-base implementation.[5]

Our work is progressing in several areas. We are adding multiple-user support to the new object-base implementation. We are investigating concurrency and recovery support through long transactions. To support Smile's capability of background processing, we are considering extending the rule interpreter to allow concurrent rule firing.

References

1. G.E. Kaiser and P.H. Feiler, "Intelligent Assistance without Artificial Intelligence," *Proc. Compcon*, CS Press, Los Alamitos, Calif., 1987, pp. 236-241.
2. A. Nico Habermann, D. Notkin, "Gandalf: Software-Development Environment," *IEEE Trans. Software Eng.*, May 1985.
3. D.B. Garlan and P.L. Miller, "Gnome: An Introductory Programming Environment Based on a Family of Structure Editors," *SIGPlan Notices*, May 1984, pp. 65-72.
4. D.E. Perry, "Software Interconnection Models," *Proc. Int'l Conf. Software Eng.*, CS Press, Los Alamitos, Calif., 1987, pp. 61-69.
5. G.E. Kaiser et al., "Database Support for Knowledge-Based Engineering Environments," *IEEE Expert*, Summer 1988.

to the user automatically. For example, say 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. Marvel can use the same information to check if the affected components are accessible (have been reserved by the user) for modification. The result of this query can again be presented to the user, or Marvel can attempt to reserve and/or add new editing tasks to the user's agenda.

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.

Opportunistic processing

Marvel performs opportunistic processing, which means it undertakes simple development activities so programmers need not be bothered with them. In our model only menial activities are automated, such as determining when the

source code has changed, invoking the compiler, and recording errors found during compilation.

Marvel performs an activity when the opportunity arises, between the time a user's action causes additional processing and the time the user requests the results of the action. This form of assistance differs from intelligent assistants such as the Programmer's Apprentice (also known as KBEmacs[3]), which focuses on automatic program construction.

In addition to objects, the object base maintains the process model that helps Marvel decide when to apply tools on the user's behalf. The process model is an extensible collection of rules consisting of a precondition, an activity, and many postconditions.

Marvel carries out its actions by interpreting the rules in different ways. Forward chaining lets Marvel invoke tools as soon as their preconditions are satisfied; backward chaining lets it find the tools whose postconditions satisfy the preconditions of other tools that have been activated.

The extent of this automation is controlled through strategies. Each strategy specifies a certain degree of assistance that is appropriate for a type of user or law of programming activity. For example, Marvel automatically performs different functions for an long-term user than it would for a novice. Similarly, Marvel may report on the use of undefined variables less frequently when new code is written than during test and debugging.

It is important to realize two facts about the use of rules in Marvel. First, Marvel consists of a generic kernel. An instance of Marvel is created by supplying a description of the object base structure and the process model to the kernel. Second, only systems managers need to write object base descriptions, rules, and strategies. Users select from strategies defined for them to choose a desired behavior of Marvel. They can extend the set of strategies if desired.

Rules. Marvel rules are based on condition/action pairs. When the condition is true or satisfied, the action is applied to working memory (in this case, the object base). However, these so-called production rules are inadequate because they do not separate the invocation of a tool from the results produced by the tool, which we must do to integrate existing tools without modification. Therefore, we divide a rule into three parts: a precondition, an activity, and a postcondition.

Figure 1 shows a compile rule that illustrates the properties of these three parts.

Preconditions. A precondition is a Boolean expression that must be true before an activity can be performed. The operands of a precondition are objects and their attributes.

In Figure 1, notcompiled(module) is a precondition for the compile-module activity. Assuming that static semantic analysis and code generation are separate activities, the precondition also requires all semantic analysis to have completed successfully. This takes the form of "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* did not find any errors.

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

in(module, component c)
  {edit component c}
notanalyzed(component) and
  notcompiled(module);
```

Figure 1. Compile rule and edit rule.

Activities. The activity part of a rule represents an integral development task, such as compile module and edit procedure. Activities are medium-grained: Low-level editing commands applied during the course of an edit-procedure activity are not considered activities. Nor are high-level commands, such as "fix bug," because they involve many tasks and perhaps many users.

In the object base, each activity is associated with a tool that carries it out. Each tool has an attribute that determines if it can be invoked by the environment without human intervention. For example, the compile-module activity is associated with the compiler, which can be invoked automatically; the edit-procedure activity is associated with an editor, which requires human interaction.

Postconditions. A postcondition is an assertion that becomes true when an activity is completed. A postcondition can consist of several alternative assertions. Each alternative reflects a different result of the activity. For example, the compile rule in Figure 1 shows compiled(module) and errors(module) as the two possible assertions, capturing the fact that compilation may succeed or fail. The postcondition alternatives are mutually exclusive — only one gets asserted, based on the result of the activity. Both preconditions and postconditions are written as well-formed formulas in first-order, predicate calculus.

Our rules are similar syntactically to Hoare's assertions,[4] where a programming language construct is associated with its preconditions and postconditions. If the preconditions are true before the language construct is executed, the postconditions will be true afterward. However, the semantics of Marvel's postconditions differ from Hoare's in that the purpose of the postcondition is not verification, but to update the object base.

Controlled automation. Forward and backward chaining contribute to oppor-

```
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 2. Change rules and reserve rule.

tunistic processing by letting Marvel use rules to determine what needs to be done and what can be done automatically.

Forward chaining. If the preconditions of an activity are satisfied and the activity is one that it can perform, Marvel does so without human intervention. This behavior is similar to language-oriented editors, which automatically perform actions like type checking and code generation when a user makes a subtree replacement in a program's abstract syntax tree.

Marvel would interpret the rule in Figure 1 to mean that the assistant may compile all modules *M* if all the components of *M* have been analyzed successfully and *M* has not yet been compiled. If a module was previously unsuccessful at compiling, the postcondition errors(module) will be true. The compile-module activity will not be reported unnecessarily while errors(module) is true, because the precondition notcompiled(module) cannot be satisfied. If the user edits a component to fix the error, the edit activity will cause notcompiled(module) to be true again, and compilation can be attempted.

Forward chaining means Marvel can perform this second attempt at compilation when that precondition is satisfied. It does not have 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 new postconditions are generated, using otherwise idle computing resources.

Backward chaining. If a user invokes an activity whose preconditions are not satisfied (execute program, for example), Marvel looks for activities it can perform to generate postconditions that would satisfy the preconditions. It uses backward chaining to do so; this is similar to Make.

When a user requests regeneration of an executable system after changes have been made to its source code, Marvel uses dependency information it maintains in the object base to determine which modules 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 find and repair bugs, for example. In general, Marvel will not automatically perform activities that invoke tools requiring human intervention.

Consider the case of a large programming team where multiple users are not permitted to change the same module at the same time. This might be handled with a rule like that in Figure 2, which requires each user to reserve a module before changing it. The preconditions for the reserve-module activity are (1) the module has not been reserved (not reserved(module)) and (2) the module has been saved by the version-control tool (saved(module)).

The second rule in Figure 2 states that the change-component activity cannot be done unless the module that contains the component is reserved. The change-component activity lets the user modify the specification of a component, as opposed to edit component, which lets the user modify the component's body only.

The third rule in Figure 2 states that not only should the containing module be reserved, but the user must reserve any other modules whose components use the component that will be changed (*c* and *k* are two objects of the same type). Backward chaining lets Marvel automatically reserve any modules whose components may be modified to remain consistent with the changed component. It also prevents the user from modifying the specification of a component when other modules cannot be reserved (according to the first rule), which means that someone else is currently working on them. Thus, the user does not start a job he may not be able to finish.

Hints and strategies. When Marvel performs opportunistic processing, it must choose the degree of automation wisely. In other words, it must adapt to the user's current goals. To do this, Marvel selects appropriate points on the spectrum between the earliest and latest time an activity can be performed automatically and disables automatic processing when it gets in the user's way. We have provided Marvel with hints and strategies to help it make these decisions.

A hint is a rule with no postconditions. The preconditions of a hint are used to help Marvel decide when to apply a tool whose preconditions are satisfied.

For example, it makes sense that Marvel should delay recompiling a module automatically even when preconditions are satisfied if a user with modification rights is browsing the module. The rationale is that the user may decide to edit some components, and the generation of code will have been wasted. This is captured in a hint shown in Figure 3, giving this precondition for the compile-module activity. When Marvel follows a strategy that includes this hint, compilation is delayed until the user changes his focus to another module.

Of course, the user must be allowed to invoke the compiler without changing focus to another module. That is why this precondition is stated as a hint, not 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; ignored during backward chaining.

A strategy is a collection of hints and rules that 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

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

Figure 3. Compile hint.

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, the user selects appropriate strategies by telling the environment something about his intentions: for example, that he is a manager versus a programmer, developing a new software system versus maintaining an old software system, or making major changes versus making a minor revision. A strategy whose rules and hints result in automatic type checking immediately after each component is edited would be appropriate for a minor revision, but not for a major change involving many interrelated components.

Handling side effects

Using a tool often causes side effects. For example, the analysis tool invoked for the analyze-component activity may change the values of several component attributes. 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 tool attributes. In such cases, the secondary activities are often described by their own rules, and these must be considered for further processing.

Figure 4 shows some rules related to a component's uses attribute. The uses attribute lists the other components the component depends on. The first rule gives the obvious preconditions and postconditions for the analyze-component activity. The second rule states that 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 states 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.

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 if it is imported. If q is not already imported, Marvel notes that imports(module, component) is a postcondition of the import-component activity (the third rule) and further realizes it can perform the import-component activity.

So it considers the preconditions of the import-component activity. Marvel queries its object base to find the module that does contain q. If q is already exported from this module, Marvel imports it. If not, backward chaining lets Marvel follow the preconditions of this activity given in the fourth rule, add q to the exports of its module, import q into the original module, and finally allow the analysis tool to set the uses attribute of p.

This is only one possible strategy. It ignores the possibility that distinct procedures named q might be found in more than one module. Sometimes language-specific typing information can narrow the possibilities, but Marvel usually must interrupt the user to explain its dilemma and ask which q is intended.

Another possibility is that there is no component named q in the object base. If so, 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. Or Marvel could ask the user to create the procedure (or its stub) before continuing the analysis, but this might be intrusive.

The preferred solution 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 the above discussion, import component and export component do not require human interaction, so Marvel can carry out the repairs. 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 compo-

```
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. Analyze rule, uses rule and import/export rules.

nent without the appropriate With clause should be detected as an error. A second alternative would require Marvel to ask the programmer if q is a typographical error before carrying out all the previously described actions.

Over time the modular structure of systems degenerates. For systems written in languages with explicit export/import declarations, such as Ada, the number of these declarations tends to increase, even though some imported components are no longer used.

Marvel can maintain such old code by providing both rigid and flexible strategies in the same environment. Flexible strategies let it reflect the actual usage of components automatically in the export/import lists, removing unnecessary exports/imports and adjusting exports/imports as the code is being reorganized. Rigid strategies provide stability during development phases such as testing and integration by taking the export/import declarations as givens to be checked against.

In Figure 4, Marvel implicitly queried its object base to locate procedure q. Implicit queries are necessary to determine if preconditions are satisfied and to find the next rules to be applied in forward and backward chaining. Implicit queries are also used to anticipate the postconditions of activities. This lets Marvel notify the user as soon as a user action is likely to lead to adverse results.

Consider the two rules in Figure 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 with the change-component command, he indicates to Marvel his intention to modify the component specification. Marvel notices that forward chaining after the completion of the editing activity would propagate to other components based on the used-by attribute, whose reprocessing might result in error.

Instead of letting the user edit the component specification blindly, Marvel can query the object base and inform the user of the potentially affected sites. This lets the user abort his change-component command if he was not aware of the potential damage caused by the intended change.

Adding knowledge to tools

Make[1] has a simplistic world model consisting of files and command lines. A Make file 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 time stamp of an input file is later than the time stamp 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 object base consists of files that have a single attribute, their time stamp. 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 versus object files, of modules versus systems, of programmers or of programming.

How can we add this knowledge 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 defines the attributes, or properties, of its objects. For example, a module-object-code object might have a history attribute that describes how it was generated and a derivation-of attribute that points to the object representing the corresponding source code.

Rules would then be added to model the part of the 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, and task management, assuming all these tools were similarly augmented with knowledge of software objects and with understanding of their roles in the development process.

The Cornell Program 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 somewhat improved 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 reevaluate 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 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 runtime stack.

Adding this kind of knowledge to the Synthesizer Generator would make its editors relatively intelligent. For example, they could then simulate attribute reevaluation 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 and Experience*, April 1979, pp. 255-265.
2. T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Comm. ACM*, Sept. 1981; reprinted in *Interactive Programming Environments*, D.R. Barstow, H.E. Shrobe, and E. Sandewall, eds., McGraw-Hill, New York, 1984.
3. T. Reps and T. Teitelbaum, "The Synthesizer Generator," *SIGPlan Notices*, May 1984, pp. 41-48.

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

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

Figure 5. Change and analyze rules.

A sample session

Figure 6 shows a snapshot of Marvel in the middle of a procedure edit. The screen has two windows: In the large window is a transcript of a session in which the user is interacting with the Marvel command interpreter. The window is scrollable, so the complete transcript is accessible. In the small window Marvel presents an item in the object base for the user to edit using his favorite editor; in this case, Emacs. The bottom of the screen shows icons that are part of the X Windows system.

The transcript in the large window shows interactions between the user and Marvel that demonstrate some of Marvel's behavior. At the beginning of the session, the user enters an existing workspace to modify a system, in this case an interactive program for fractional arithmetic. This workspace is a Marvel database that is private to the user. It is connected to a public database, where the baseline version of the software resides.

One module has previously been reserved from the public database and made available for modification in the private workspace. All other parts of the system that physically reside in the public database are accessible transparently for reading.

The user's attention is focused on the object that represents the whole program, which is indicated by the prompt showing the system name — Fractions. First, the

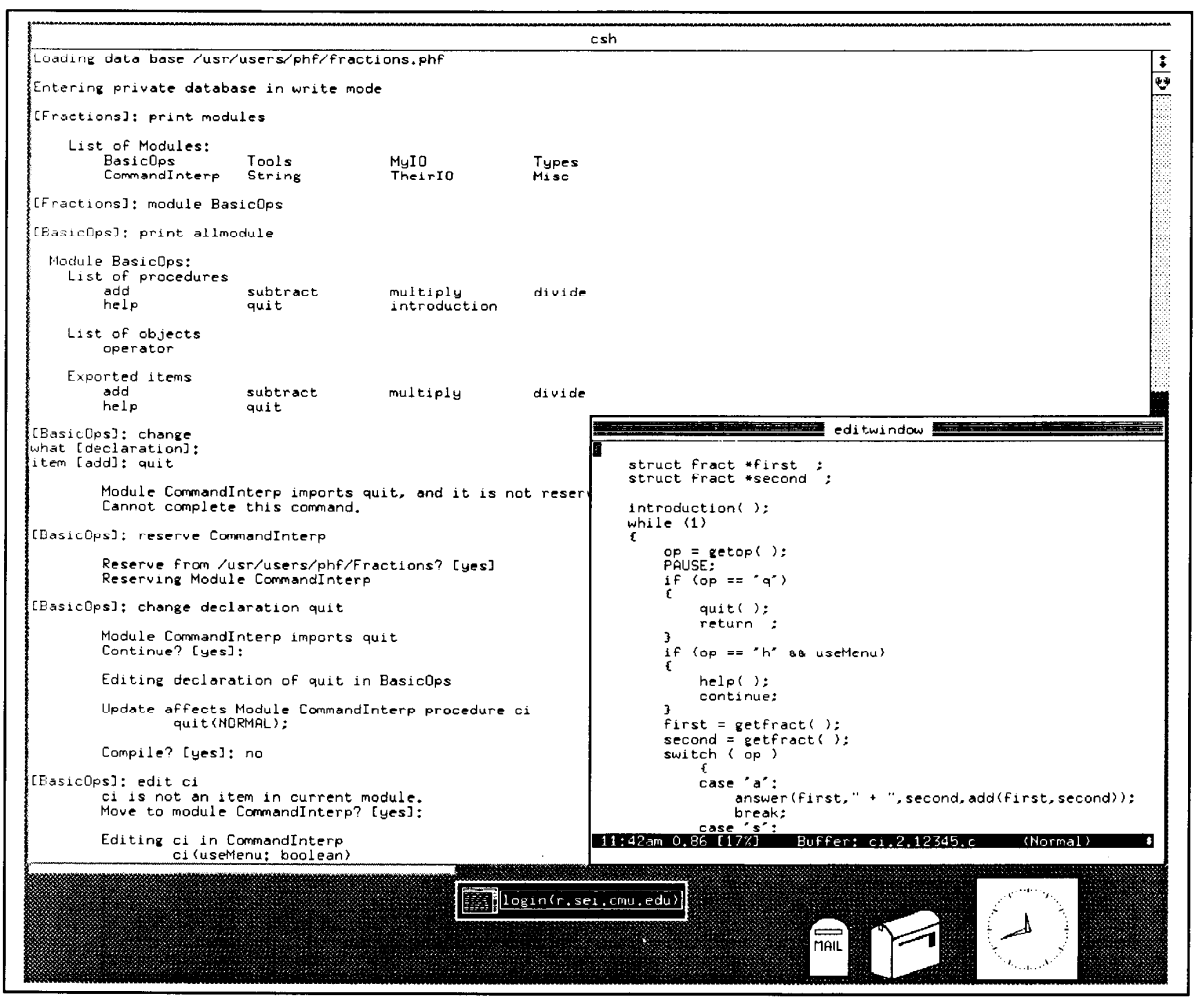

Figure 6. A Marvel screen.

user requests a view of the system, namely, its list of modules. The user then focuses on the BasicOps module, and the prompt changes.

Now the user requests another view, in this case a more detailed view of a particular module. Because the user did not specify a module name, the system chose the module in the current focus. The result is a view showing all the components of the module (several procedures and an object; the module does not contain data-type definitions), and a list of components that are available externally as part of the module specification (exported items).

With the Change command, the user attempts to modify the Quit procedure's specification. The system prompts for missing command parameters, providing defaults. Marvel first performs an implicit query to determine the consequences of the planned change. The user is informed that the Quit procedure is used by another module for which the user does not have modification rights. Under the default strategy, chosen by the user, Marvel does not reserve the module, but aborts the command.

The user then explicitly reserves the module. Marvel confirms that the module is to be reserved from the public database, and a second modification attempt succeeds. The user is informed which components are potentially affected before the actual editing, and is asked after the modification if the affected components should be analyzed and compiled as well. Because 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 the Ci procedure. Marvel changes the focus to the appropriate module, displays the procedure specification, and presents the user with the procedure body in the editor window.

The model embodied in the Marvel environment formalizes the concepts of insight and opportunistic processing by

- maintaining all knowledge about both the specific development effort and the general development process in the object base,
- making multiple views of the object base available both to users and tools,
- modeling the development process as rules that define the preconditions and postconditions of development activities, and
- gathering collections of rules into strategies.

This lets Marvel provide software-engineering environments that intelligently assist development and maintenance efforts by individuals and teams of users through controlled automation, using available development tools. ❖

Acknowledgments

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 drafts of this article and made many useful criticisms and suggestions. Purvis Jackson assisted us with technical editing.

This work was started while Kaiser was a visiting computer scientist at the SEI. The first prototype implementation was done at the SEI. Research on Marvel continues at Columbia University, supported in part by Kaiser's Digital Equipment Corp. faculty award, in part by a grant from Siemens Research and Technology Laboratories, and in part by the Defense Dept.

References

1. T. Winograd, "Breaking the Complexity Barrier (Again)," *Proc. ACM SIGPlan-SIGIR Interface Meeting on Programming Languages — Information Retrieval*, ACM, New York, 1973, pp. 13-30; reprinted in *Interactive Programming Environments*, D.R. Barstow, H.E. Shrobe, and E. Sandewall, eds., McGraw-Hill, New York, 1984.
2. D.S. Wile and D.G. Allard, "Worlds: An Organizing Structure for Object-Bases," *SIGPlan Notices*, Jan. 1987, pp. 16-26.
3. R.C. Waters, "KBEmacs: Where's the AI?" *AI Magazine*, Spring 1986, pp. 47-56.
4. C.A.R. Hoare, "An Axiomatic Approach to Computer Programming," *Comm. ACM*, Oct. 1969, pp. 576-580, 583.

Gail E. Kaiser is an assistant professor of computer science at Columbia University, where she received a Digital Equipment Corp. faculty award. Her research interests include programming environments, evolution of large software systems, application of artificial-intelligence technology to development and maintenance, reusability, object-oriented languages and databases, and distributed systems.

Kaiser received an MS and PhD in computer science from Carnegie Mellon University, where she was a Hertz fellow, and a BS from the Massachusetts Institute of Technology.

Peter H. Feiler is a senior computer scientist at the Software Engineering Institute, where he is a member of the team that is evaluating Ada environments. His research interests include development support environments, interactive development tools, application of artificial intelligence to software engineering, and support for concurrent applications.

Before joining SEI, he was a research scientist and group leader at Siemens Corporate Research and Technology Laboratories. He received a Vordiplom (BS) in mathematics and computer science from the Technical University in Munich and a PhD in computer science from Carnegie Mellon University. He is a member of the ACM, AAAI, and the Computer Society.

Steven S. Popovich is a graduate student at Columbia University. His research interests include programming environments, distributed systems, and artificial intelligence.

Before beginning his graduate studies, he worked for the Software Engineering Institute, the Carnegie Group, MindBank, Siemens Research and Technology Laboratories, and Carnegie Mellon University. He received a BS in computer science from Carnegie Mellon University.

Questions about this article can be addressed to Kaiser at Computer Science Dept., 450 Computer Science Bldg., Columbia University, New York, NY 10027.