AI Techniques in Software Engineering

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

CUCS-515-89

Stoll, H.W. (1986). Design for Manufacturing—An Overview. *Appl. Mech. Rev.* 39(9):1356–1364.
Stotts, P.D., and Ning Cai, Z. (1988). Modelling Temporal Behavior of Robot Lattices with Binary Timed Petri Nets. University of Maryland, College Park, Md.
Stotts, P.D., Jr., and Pratt, T.W. (1985). Hierarchical Modeling of Software Systems with Timed Petri Nets. Proc. Int. Workshop on Timed Petri Nets, pp. 32–39.
Szenes, K. (1982). An Application of a Parallel Systems Planning Language in Decision Support-Production Scheduling. In: *Advances in Production Management: Production Management Systems in the Eighties*. Proc. IFIP WG 5.7 Working Conf., Bordeaux, France. pp. 241–249.
Tulkoff, J. (1981). Lockheed's GENPLAN. Proc. 18th Numerical Control Society Annual Meeting and Tech. Conf., Dallas, Texas.
Tzafestas, S.G. (1986). Knowledge Engineering Approach to System Modelling, Diagnosis, Supervision and Control. *Preprints*, IFAC Int. Symp. on Simulation of Control Systems, Vienna, pp. 17–31.
Tzafestas, S. (1988a). AI Techniques in Control: An Overview. In: *AI, Expert Systems and Languages in Modelling and Simulation* (IMACS Proc. 1987) (C. Kulikowski and G. Ferrate, Eds.). North-Holland, Amsterdam.
Tzafestas, S.G. (1988b). Expert systems in CIM Operations: Key to Productivity and Quality. Proc. 3rd Int. Symp. in Systems Analysis and Simulation, Berlin, GDR.
Tzafestas, S.G. (1988c). System Fault Diagnosis Using the Knowledge-Based Methodology. In: *Fault Diagnosis in Dynamic Systems: Theory and Applications* (R. Patton, P. Frank, and R. Clark, Eds.). Prentice-Hall Intl. (UK) Ltd., London, Chapter 15.
Tzafestas, S. (1988d). *Knowledge-Based System Diagnosis, Supervision and Control*. Plenum, New York.
Tzafestas, S.G., and Tsihrintzis, G. (1988). ROBBAS: An Expert System for Choice of Robots. In: *Managerial Decision Support Systems and Knowledge-Based Systems* (M. Singh and D. Salassa, Eds.). Elsevier/North-Holland, Amsterdam.
Tzafestas, S.G., Singh, M., and Schmidt, G. (1987). *System Fault Diagnostics, Reliability and Related Knowledge-Based Approaches*, Vol. 2, D. Reidel, Dordrecht and Boston.
Valette, R., Cardoso, J., Aatabakhche, H., Courvoisier, M., and Lemaire, T. (1984). Petri Nets and Production Rules for Decision Levels in FMS Control. Proc. 12th IMACS World Congress, Paris, Vol. 3, pp. 522–524.
Vancza, J. (1988). Organizing Classificatory Knowledge by Induction: A Case Study in Manufacturing Process Planning. Proc. 12th IMACS World Congress, Vol. 4, pp. 258–260.
Viswanadham, N., and Narahari, Y. (1987). Colored Petri Net Models for Automated Manufacturing Systems. Proc. IEEE Int. Conf. Robotics and Automation, Raleigh, N.C., pp. 1935–1990.
Walker, A. (1985). Syllog: An Approach to Prolog for Nonprogrammers. In: *Logic Programming and Its Applications* (M. van Caneghem and D.H.D. Warren, Eds.). Ablex, Norwood, N.J.
Warman, E. (1985). AI in Manufacturing: An Organic Approach to Manufacturing Cells. *Data Proc.* 27(4):31–34.
Warren, D.H.D. (1974). WARPLAN: A System for Generating Plans. DCL Memo 76, Dept. of AI, Edinburgh Univ.
Weill, E., Spur, G., and Eversheim, W. (1982). Survey of Computer-Aided Process Planning Systems. *Ann. CIRP* 31(2).
Wilkins, D.E. (1985). Recovering from Execution Error in SIPE. *Comput. Intell. J.* 1.
Williams, T.J. (1983). Developments in Hierarchical Computer Control Systems. Proc. CAPE'83, Amsterdam.
Woo, T.C. (1982). Feature Extraction by Volume Decomposition. Proc. Conf. CAD/CAM Technology in Mechanical Engineering, Cambridge, Mass., pp. 76–94.
Zenie, A. (1985). Colored Stochastic Petri Nets. Proc. Int. Workshop on Timed Petri Nets, pp. 262–271.
Zenie, A. (1987). Qualitative and Quantitative Validation of a DDMS Model Using a CSPN. In: *Applied Modelling and Simulation of Technological Systems* (P. Borne and S. Tzafestas, Eds.). Elsevier/North-Holland, New York, pp. 537–545.

CHAPTER 7

AI TECHNIQUES IN SOFTWARE ENGINEERING

GAIL E. KAISER

1 INTRODUCTION

The idea of using artificial intelligence techniques to support programming has been around for a long time. The earliest notion was to avoid programming entirely. The human user would just tell the computer what to do, without saying how to do it, and the computer would do the right thing. Even if this were feasible, however, it would be much too tedious, since each time the user would have to repeat the details of what he wanted done. So the goal of programming was to explain things to the computer only once, and then later on be able to tell the computer to do the same thing again in some short form, such as the name of the "program." Thus the idea evolved that a user would somehow tell the computer what program was desired, and the computer would write down the program in some internal form so that it could be remembered and repeated later. The assumption was that the resulting program would be correct, complete, efficient, easy to use, and so forth. It would also be exactly what the human user wanted.

Several problems would have to be solved to achieve this goal. The first is determining exactly what the human really wants. This is a notorious problem in software engineering; the customer states extensive requirements, the company develops software that seems to them to meet all requirements to the fullest extent, but then the software is next to useless for the customer because what he said he wanted was not really what he needed. It may have been a computerized version of manual procedures that were themselves idiosyncratic, or there may be a better way to do things once computers are introduced, or the customer's employees just might not be ready to accept comput-

erization. Today there is often no explicit customer, but a perceived marketplace for which software is developed. Understanding and predicting the marketplace is more of a black art than a science.

When there is a specific customer, or an internal marketing group simulating the potential customers, some of these problems can be solved by *rapid prototyping*. Rapid prototyping generally involves a "quick and dirty" version of the program, often in an interpretive language such as Lisp or Smalltalk. This version usually does only a tiny fraction of the things needed and most likely does them extremely slowly. The point is that it is relatively fast and inexpensive to build a prototype, and then the customer can get the feel of the eventual program by playing with this prototype. In theory, any problems with the requirements are discovered during this phase, when only a small amount of time and energy have been expended. Some programs go through multiple prototypes before the customer finally agrees that this was what he was looking for. Then development of production quality software begins.

The second problem with our scenario of the user telling the computer what is desired and the computer then writing the program is, How does the user tell the computer what he wants? Once upon a time, it was assumed that natural language would do the trick. But research on natural language understanding has not advanced to the point where this scenario is feasible and possibly never will. One problem is that natural language is inherently imprecise, and it may be impossible for the human to express *exactly* what he wants in natural language. Another problem is that conversation among humans presumes a large amount of shared context, based on similar education, social and corporate culture, working together toward a common goal or on a shared task, and generally being in the same place at the same time. Popular psychology dwells on how people do not always understand each other, even when they speak the same language and have very similar backgrounds, except for the very simplest of communication scenarios. The knowledge acquisition task is formidable for humans, perhaps impossible for computers (at least until someone invents the "mind reading" module).

Once natural language was abandoned (or perhaps just put on the shelf for a while), researchers turned to more precise notations for describing what is desired. Formal and informal specification languages have been developed for expressing the requirements for computer programs. Formal specifications are typically based on mathematics and logic, and the concerns include making sure a particular specification is complete, consistent, and correct. An often neglected issue is whether the specification defines what the program should not do as well as what it should do. An informal specification, by definition, cannot be entirely complete, consistent, and correct. Some means must be provided for user feedback, for example, "No, that's not quite right—I really meant such-and-such." The obvious preference is for an interactive, incremental style of debugging the specification, where the user can quickly try out small changes to each part and immediately determine at least the gist of the resulting change in the final program.

Given that the computer understands what the human wants, the third problem is producing the software. It must run on the available hardware and devices, most likely using the available operating system and utilities rather than the bare machine. Resource usage must be reasonable—there are many slow, wasteful ways to do things, but these are not usually acceptable. However, some resource-related desires may not be feasible to achieve in a given operating environment, while others involve trade-offs—for example, the usual time–space trade-off. Some requirements are just plain impossible with current technology, or with any technology; for instance, there are physical limits to how fast a single-processor computer can execute instructions.

There have been a number of research efforts toward this goal of *automatic programming* based on both formal and informal notations. One caveat is in order: Program generation is not the same thing as automatic programming. As with automatic programming, the user provides a formal specification as input and the system generates the desired program. However, automatic programming systems are general-purpose, while program generation systems support an extremely limited but very well understood domain (for example, window systems, parsers, database reports, syntax-directed editors). In the following section, we briefly sketch two of the best-known and longest-term automatic programming efforts, and provide references to the literature for these and a few other projects.

A perhaps more realistic approach to applying artificial intelligence to software development is to automatically perform certain menial tasks rather than attempt to take over the creative activities such as programming. The term "menial" is not intended to be pejorative. Menial tasks have to be done but do not involve the same levels of analysis and synthesis as does programming. Most bookkeeping tasks are relatively menial, for example, keeping track of the status of bug reports, major and minor releases, documentation updates, test data used for regression test suites, and so on. Invocation of many tools, particularly batch (noninteractive) tools, is also menial; it is necessary for someone (or something) to know what processing tools to invoke in what order with which switches and arguments and where to store away their results.

Here again we run into the problem of how to describe the menial tasks that need to be done and, more significantly, exactly under what circumstances they should be done. In most cases, it is not appropriate for the human user to remember that the task has to be done, recognize that now is the best time to do it, and then tell the computer "do such and such task right now," because this is likely to take more effort on the part of the human than just doing the task herself. Therefore, the system must continuously monitor the user's activities in order to keep track of what is going on and what is the right thing to do next.

The bulk of this chapter is concerned with *intelligent assistance* as a practical alternative to automatic programming. Intelligent assistance is loosely defined as any knowledge-based technology that assists human users

in carrying out their activities in a manner that does not require any creativity on the part of the assistant. Intelligent assistance is feasible as an extension of existing software development environments and tools and needs relatively little additional effort to become commercially viable.

2 AUTOMATIC PROGRAMMING

Refine is a commercial product marketed by Reasoning Systems, Inc. It is based on many years of research on automatic programming and intelligent assistance at Kestrel Institute (Goldberg, 1986; Kedzierski, 1984; Smith et al., 1985). Refine is essentially a programming environment for a wide-spectrum programming language called the Refine language. A wide-spectrum language is one that includes a range of facilities from very high-level constructs that are not directly executable (except perhaps by *extremely* slow interpretation) to relatively low-level (and efficient) constructs in a conventional programming language. Refine in particular integrates programming language constructs from set theory, logic, conventional procedural programming, and transformation rules. The rules are used to semiautomatically transform a Refine program—basically a formal specification of the desired software system—into a conventional Lisp program. The user interacts with the transformation process when Refine gets stuck or has multiple choices. Refine may be considered an automatic programming system, since it generates a software system from its specification, but it cannot operate without a lot of help from a human user. Very large programs have been successfully developed using Refine, including the Refine system itself. However, since Refine generates Lisp, which is itself a prototyping language, even the final output of the Refine system is really only a prototype.

A number of other research projects have investigated program transformation systems, with two basic orientations. The first, which appears for example in CIP (Broy and Pepper, 1981), is based on a small set of mathematical formalisms. The system automatically replaces components of the specification by applying correctness-preserving transformations. In contrast, Draco (Freeman, 1987), FSD (Balzer, 1985), PDS (Cheatham et al., 1979), and similar systems allow transformations to be selected by the programmer to reflect design decisions. This kind of system can deal with incomplete specifications and a growing catalog of transformations, as the programmer works together with the system to produce a complete program.

The Programmer's Apprentice (Rich and Waters, 1988a; Waters, 1986) is a long-term research project at the MIT AI Laboratory. Several automatic programming systems have been developed as part of this project. The best known is KBEmacs, for Knowledge-Based Emacs, which extends the well-known Emacs word processing system (Stallman, 1981) with facilities for understanding a very restricted natural language description of the desired program and interactively producing a program in any one of several languages (e.g., Lisp, Ada). The description is in terms of a library of programming

clichés, which consists of abstract program fragments ranging from very simple abstract data types such as lists to abstract notions such as synchronization and complex subsystems such as peripheral device drivers. The apprentice interacts with the human user, who is thus able to make corrections and rearrange parts of the program. At any time the human can completely take over the programming task, interact directly with Emacs and other programming tools, and then return control to the apprentice. Only very small programs have been produced to far, in large part due to underestimating the inherent difficulties of this approach (Rich and Waters, 1988b). Another aspect of the Programmer's Apprentice project has been to "reverse engineer" existing programs by recognizing the clichés. This technology is still in its infancy but has great potential for solving at least part of the "corporate memory" (or lack thereof) part of the maintenance problem, since it may become possible to extract (and presumably go on to change) previous design decisions.

3 INTELLIGENT ASSISTANCE

Genie and Marvel are two representative examples of intelligent assistants. Both take over some of the programmer's more menial burdens. Genie is passive, essentially an intelligent help system component of a programming environment, and responds only to questions from the programmer. Genie figures out and tells the programmer what sequence of commands (i.e., bookkeeping operations and tool invocations) should be used to accomplish some goal selected by the programmer. Marvel is an active knowledge-based programming environment and continuously monitors the programmer's activities. Marvel participates when appropriate by automatically carrying out sequences of commands (again, bookkeeping operations and tool invocations) according to goals set in advance by the programmer or project management. For simplicity in the rest of this chapter, we will refer to commands, functions, operations, tools, and so on, as "commands."

3.1 Genie

Programming environments provide resources and facilities intended to support and assist programmers. A conflict arises between creating an environment simple enough for a new user of the environment yet sophisticated enough to accommodate an expert. Note that a new user may be an expert programmer, while an expert user of the environment need not be a programmer at all, as most large software development teams involve some nonprogrammers; to reduce confusion, we will refer to "users" rather than "programmers." A common solution to this conflict is to expose beginners to a set of starter commands but also provide more comprehensive features they can learn later. However, many beginners get trapped in the starter set, since they are not encouraged to progress to more powerful commands. One solution to this problem would be an *automated consultant* that answers a user's

questions about the environment in a manner designed to provide this encouragement, enhancing rather than detracting from the user's productivity.

All programming environments can be characterized as consisting of a set of commands with which a user can accomplish tasks or goals specific to software development in that environment. The means of access to the environment might include command languages, menus with keystroke or pointing devices, or even more sophisticated interfaces such as speech. At the core, however, a set of commands must be executed as a *plan* (i.e., sequence of steps) to carry out some task for the user.

An intelligent assistant that can behave as an automated consultant, giving appropriate help for the task at hand, can increase user productivity and the quality of the software product. The problem is then how to provide the appropriate information that neither swamps the novice with too much complex information nor insults the expert by providing an overly pedantic tutorial. The problem lends itself well to a solution using expert system techniques, namely, how to choose and articulate appropriate information from a vast and complex knowledge base.

Genie (Wolz, 1988; Wolz and Kaiser, 1988) is an intelligent assistant that behaves as an automated consultant. It generates text based both on what the user is trying to do and what the user already knows how to do. Genie takes a task-centered approach in which the help given is a direct function of a user's needs within the current context. In particular, the focus is on the content of the answer provided to a user, in order to be immediately useful without wasting time or attention span.

The first component of the approach is a small rule base that defines the assistant's behavior and a large hierarchical knowledge representation that provides the assistant's domain knowledge. The domain knowledge includes explicit information about the relationships between the tasks that can be performed within the environment, the plans used in accomplishing them, and the commands that make up the plans. The rule base allows Genie to reason about the actions associated with commands but also allows it to analyze whether plans can satisfy goals and which of many equally good plans is most appropriate in a given state of software development. In a modification request (MR) environment, for example, a task might be to read a set of MRs submitted by customers and forward the subset concerned with a particular subsystem to the manager responsible for that subsystem. The plan for executing that task will depend upon the particular commands available within the MR environment.

The second component follows from the beliefs that classifying commands, plans, and goals according to the level of expertise is inappropriate and that global categorization of users as "novice," "intermediate," or "expert" is inadequate. Instead, information on an individual's exposure to goals, plans, and commands influences what specific information the intelligent assistant presents following a user's query. Expectations about what the user knows and should be told is based on the tasks that the user has completed in the past

rather than on broad ad hoc classifications of commands and plans as "easy" or "hard." A task-centered representation is used as a user model in order to exploit the structure of the knowledge base. Decisions about how to answer a user's questions are based on an analysis of the match between the knowledge base and the user model. Taking another example from an MR environment, a user may have extensive experience with forwarding MRs to other users and almost none with annotating completed MRs—to refer to the corresponding source code changes—and installing them in the permanent database. Such a user will not fall nicely into a categorization of expertise. A question relating to sending simple messages will require introducing very little new information into the discussion, while a question about modifying and installing MRs may require an extensive introduction to database facilities.

3.1.1 CONSULTING IN PROGRAMMING ENVIRONMENTS. In order to use an environment effectively, a user must know its capabilities and how to make best use of them. This requires access to information that describes the specific features of the programming environment, that is, the commands available. It also requires access to methods or plans for best accomplishing goals.

There is a large middle ground between a novice who knows only the rudiments of an environment and an expert who has gained complete mastery over it. The continuum in between is one in which user expertise may not be optimal for a given task. When the user must take time to find the appropriate command or develop an efficient technique, productivity decreases. Furthermore, in some environments in which the tasks are perceived as primarily bureaucratic (e.g., writing documentation), users may rely on inefficient methods that are well known rather than taking time to develop more sophisticated expertise. A primary reason for the inefficiency of learning new skills within a programming environment is that users bear the burden of locating the appropriate information. This is typically done by searching through manuals, asking help of others, or simply experimenting with the environment. Expert system techniques should be able to provide mechanisms that can relieve some of this burden. The objective of the Genie research effort is to address these issues and offer a theory of how to build an intelligent assistant that aids users in extending their expertise with the programming environment.

There is rarely a direct correspondence between a precise statement of a user's task and a plan to satisfy it. It is more often the case that the user's goal is poorly defined. Furthermore, a goal may be satisfied by more than one plan. The problem presents itself as requiring a mapping of many user queries to many possible answers. In order to constrain the potential mappings, user queries can be categorized at least partially as relating goals to plans as summarized in Fig. 7-1.

There is a distinction between information that is definitional and information that is instructional. Figure 7-2 further refines this distinction. *Definitional information* is more appropriate for reminding someone about some-

1. Command specification: What does command C do?
2. Goal satisfaction:
   a. What do I do to accomplish goal G?
   b. Plan P accomplishes goal G, but is there a better way?
3. Analyze or debug a plan:
   a. What does plan P do?
   b. Why doesn't plan P accomplish goal G?

FIGURE 7-1
Typical questions users ask.

thing they have previously used, while *instructional information* is more appropriate for introducing new commands. These types differ not only in their format and level of detail, but also in their emphasis and the degree to which related information is included. Clarifying and elucidating require a careful mixture of reminding and introducing. Genie addresses only the first four types of answers. Marvel, described in the next section, automatically generates and executes plans for the user.

Although the categorization in Fig. 7-1 constrains the question, while the taxonomy in Fig. 7-2 constrains the answer provided, the requisite knowledge and the processes needed to search that knowledge are still complex. The processes include the abilities to estimate the user's goal, to understand the user's plan, to evaluate the current situation in order to formulate an answer that does not digress from the current task, to analyze the user's plan in terms of the estimate of the goal and within the current situation, and to choose an appropriate answer and explanation depending on the user's current knowledge of the environment. This requires knowledge of the commands provided, the possible tasks that can be accomplished, the plans that may accomplish those goals, the things that typically go wrong (bugs), and what the user currently does and does not know about the commands, goals, plans, and bugs.

Introduce:	Present commands and plans that the user has not encountered before.
Remind:	Briefly describe commands and plans that the user has been exposed to but may have forgotten.
Clarify:	Explain details and options about commands and plans to which the user has been exposed.
Elucidate:	Clear up misunderstandings that have developed about commands and plans to which the user has been exposed.
Execute:	Perform commands and plans directly for the user.

FIGURE 7-2
Types of responses a consultant might provide.

Much of this cannot be completely known. For example, it seems unlikely that all possible goals achievable within a given programming environment will be known before the environment is used extensively. It also does not seem possible to predict with certainty what the user's task is and what the user knows. Thus not only must the processes described above operate with incomplete information, but also they ought to be able to do so effectively. Innovative techniques or novel applications ought to be easily and reliably incorporated into the knowledge base.

From an AI perspective, these issues can be encapsulated in two fundamental problems:

1. How can the search through a vast and complex knowledge base be restricted in order to glean the appropriate information for the immediate needs of the user?
2. What decisions must be made in order to choose the appropriate form in which to present that information?

Genie solves both of these problems.

3.1.2 AN INTELLIGENT ASSISTANT FOR CONSULTING. Consulting can be characterized as a three-stage process of question understanding, problem analysis, and answer generation. Genie's understanding component is currently a simple menu-based front end, sidestepping the natural language understanding problem, since it concentrates on the latter two stages: analysis, through a rule base called the Plan Analyst, and generation, through a rule base called the Explainer. The organization of Genie is depicted in Fig. 7-3. Genie attempts to answer a question by doing a two-phase search of the knowledge bases. In the first, the Plan Analyst constructs a relationship between the user's question, his user model, and the capabilities of the environment in an attempt to find the most appropriate information. Based on the Plan Analyst's output, the Explainer constructs a coherent textual explanation that takes into account what the user already knows. Both rule bases will be discussed extensively in the examples later in this section. The structure of the knowledge representation and details of the understanding and generation components that are not obvious from the examples follows below.

Genie's "understanding" component is a simple menu-based interface. Figure 7-4 shows the top-level menu, which is a reformulation of the questions of Fig. 7-1. The user can select a task or a command by typing the proper word or phrase at a command prompt or by browsing a menu of goals or commands. The menus can be arranged alphabetically, or the order of presentation can be based on links between related goals in the expert knowledge base. Plans that can be identified by name from the knowledge base can be entered from the command prompt. Otherwise, the user must construct a plan by selecting an ordered list of commands and goals.

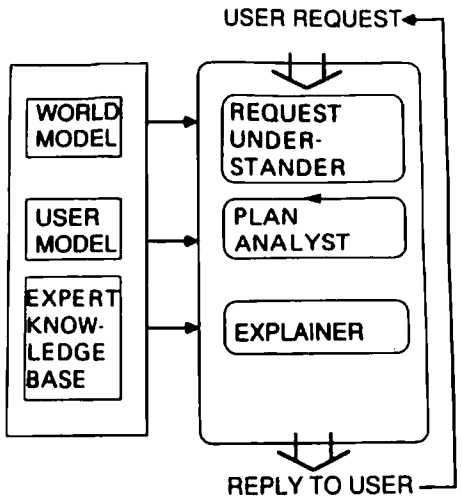

FIGURE 7-3
Genie organization. Please select a question:

When Genie is invoked within a programming environment, both the expert knowledge base and user model are loaded. The world model is constructed on the basis of the user's current status within the software development project. Depending on the question type selected, the user is prompted to provide a command C or a goal G, or to construct a plan P.

The expert knowledge base is a hierarchy of the goals that can be satisfied in the target environment. Goals contain links to alternative plans for satisfying the goal. A plan can be linked to a subgoal or an ordered sequence of subgoals that describe how it can be executed, or to a command that executes it directly. Encoded within a goal are links that describe the relationship between plans.

Commands describe the operators, functions, tools, and so on, of the environment. Their representation includes information about the correct syntax of the command, its precondition and postcondition, and the actions associated with switches and parameters. The precondition defines a state that must

1. What does *(select command)* do?
2. What do I do to accomplish *(select goal)*?
3. I use *(construct plan)* to *(select goal)*, but is there a better way?
4. What does *(construct plan)* do?
5. Why doesn't *(construct plan)* accomplish *(select goal)*?

FIGURE 7-4.
Top Level Menu for Question Selection

be true before a command can be correctly executed. It may also contain a link to a goal that could satisfy it. The postcondition encodes the actions of commands when they are applied to the world model. Currently the world model is represented as a simple add/delete list that describes possible states in the environment. Therefore postconditions are encoded as directives to add or delete a state from the world model. (It would probably be better to maintain the world model as an objectbase, as is done for Marvel.)

The user model has exactly the same representation as the expert knowledge base. It contains a history of what the user has done in the past in terms of what tasks have been completed and what plans and commands were used. It is currently coded and updated by hand, but a monitoring system like Marvel could update the user model automatically.

Most of Genie's responses are stereotypical. At the same time, the content of a response must be customized to the user's needs and expertise. Therefore, a rule-based system that ultimately leads to "canned" text is inappropriate, since the text is fixed in advance. Similarly, since Genie's range of discourse is limited, a completely open-ended natural language generation facility seems equally inappropriate. Template filling is a technique that allows both customization and stereotyped responses. To generate an answer, the Explainer selects an appropriate set of *response agenda* based on the output of the Plan Analyst. The response agenda comprises directives for filling textual templates. Representative templates are presented in Fig. 7-5.

3.1.3 MAIL SYSTEM EXAMPLE. The feasibility of this approach is explored through a relatively simple example environment. In particular, Genie has been applied to the real-world problem of the Berkeley Unix mail system, notorious for the great power it provides experts and the great confusion it creates for novices and even long-term nonexpert users. While electronic mail systems are not programming environments, they are mandatory components or adjuncts of programming environments and provide a smaller-scale laboratory for experimenting with intelligent assistance. It is important to keep in mind that Genie is not intended to replace this mail system but to augment it with intelligent behavior that makes its capabilities accessible to casual users.

We now consider two example queries based on the question types in Fig. 7-1 to demonstrate Genie' capabilities. The examples describe the rules used by the Plan Analyst to select the appropriate information. They also show typical scenarios of how the content of the user model and the user's question affect the output of both the Plan Analyst and the Explainer.

The first question is: What does `type` do? This is an instantiation of the "What does C do?" category of Fig. 7-1. In order to ask this question, the user selects question 1 in the menu of Fig. 7-4. A second menu allows the user to enter a command name, or to search commands alphabetically or by traversing

```
COMMAND_INTRODUCE(c)
        {c->name} is used to {c->satisfies->description}. It has the form {c->form},
        where FOR_EACH (x,f->parameters, "{x} refers to {px->description}").
        {c->name} requires that EXPAND_PRECOND(c->precond). It causes
        EXPAND_POSTCOND(c->postcond). For example,
        EXAMPLE(c->form,WM).

COMMAND_REMIND(c)
        {c->name}: {c->form}. It is used to {c->satisfies->description}. For
        example, EXAMPLE {c->form,WM}.

GOAL_REMIND_SIMPLE(g)
        You can {g->description} by using the command {g->command}. For
        example, EXAMPLE(c->form,WM) would
        EXPAND_POSTCOND(c->postcond).

GOAL_INTRODUCE_SIMPLE(g)
        GOAL_REMIND_SIMPLE(g). You must make sure
        EXPAND_PRECOND(g->command->preconds).

GOAL_ INTRODUCE_COMPLEX(g,fault)
        In order to {g->description}, you must
        FOR_EACH(gx,g->subgoals,"GOAL_INTRODUCE_COMPLEX(gx)"). IF
        fault DESCRIBE_FAULT(fault->plan). The commands to {g->description}
        are
        FORMAT_PLAN_INSTANTIATION(gx,g->subgoals,gx->command,WM).
        SHOW_MAPPING(gx,g->subgoals,gx->description,gx->command).

GOAL_REMIND_COMPLEX(g)
        In order to {g->description}, use
        FORMAT_PLAN_INSTANTIATION(gx,g->subgoals,gx->command,WM).
        SHOW_MAPPING(gx,g->subgoals,gx->description,gx->command).

/* Operations appear in capital letters. Variables are surrounded by braces. WM = World Model. Simple
        goals are satisfied directly by commands. Complex goals are satisfied by a plan that
        maps to subgoals. */
```

FIGURE 7-5
Representative response agenda.

links between goals. Using one of these methods the user indicates that the desired command is `type`.

Figure 7-6 shows the portion of the expert knowledge base required to answer this question. The Plan Analyst uses the following rules to determine what information is relevant to the Explainer:

1. If the user model contains command C, then report knowledge of C, else report no knowledge of C.
2. If there exists a command that is directly satisfied by some goal H, which has the least complex link to the goal G that satisfies command C, then D = that command.
3. If D exists in the expert knowledge base and the user model contains C, then report knowledge of D.

In the example, the Plan Analyst would determine whether the user already knows about `type`, and in this case, since there is a link to `print`, whether the user knows about `print`. The outcome of this analysis is passed to the Explainer.

Four analyses are possible based on the existence of C and D in the user model. These are illustrated in Fig. 7-7 along with the corresponding Explainer output. If the user knows nothing about either `type` or `print`, Genie generates the standard introductory template for `type` and does not overwhelm the user with the fact that `print` is a synonym. Figure 7-8 shows how the response agenda for `COMMAND_INTRODUCE(type)` is filled from the expert knowledge base. If the user knows about `print`, Genie states the fact that `type` is a synonym, reminds the user about `print`, and then introduces `type`. If the user knows about `type` but not `print`, Genie reminds the user about `type` and makes an aside that there is a synonym for `type` called `print`. Finally, in the last case, if the user knows about both, Genie just reminds him about `type`.

The second question is: How can I reply to a message? This is an instantiation of the "How can I satisfy G?" category of Fig. 7-1. To ask this question, the user selects question 2 in the menu of Fig. 7-4. In a second menu, the user selects the desired goal. Let us assume the user chose "`reply.to.message`." In this case it might be easier to locate the goal by

```
Goal type.goal,
        G_type:  Direct /* Satisfied directly by command */
        Satisfied by:  Command type;
        Related goals:  RL1
Goal print.goal,
        G_type:  Direct
        Satisfied by:  Command print;
        Related goals:  RL1
Goal display.list.of.messages,
        Description:  display each message in the sequence specified
        G_type:  Subgoals /* Satisfied through subgoals */
        Satisfied by:  Goal type.goal; Goal print.goal;

Command print,  Form: print (message_list)
        Precondition:  PR1 and PR2 and PR3,
        Postcondition: PO1,
        Satisfies:  print.goal
        Parameters: message-list
Command type,   Form: type (message_list)
        Precondition:  PR1 and PR2 and PR3,
        Postcondition: PO1,
        Satisfies:  type.goal
        Parameters: message-list

Plan P1,  state: (exists contents_of (message_list))
        use: list.message
Plan P2,  state: (at read-level)
        use: get.to.read.level
Plan P3,  state: (size (message-list) > screen-size)
        use: set.window.scroll

Rule RL1, type.goal, print.goal
        Relation: synonyms
```

FIGURE 7-6
Expert knowledge for question 1.

```
/* user model does not contain either type or print */
Plan Analyst output:  command: type     no_knowledge
Explainer output:     COMMAND_INTRODUCE(type)

/* user model contains print, but not type */
Plan Analyst output:  command: type no_knowledge
                      command: print knowledge
Explainer output:     DESCRIBE_LINK(type,print)
                      COMMAND_REMIND(print)
                      COMMAND_INTRODUCE(type)

/* user model contains type,  but not print */
Plan Analyst output:  command: type knowledge
                      command: print no_knowledge
Explainer output:     COMMAND_REMIND(type)
                      MAKE_SIDE_COMMENT(DESCRIBE_LINK(type,print))

/* user model contains both type and print */
Plan Analyst output:  command: type knowledge
Explainer output:     COMMAND_REMIND(type)
```

FIGURE 7-7
Responses to question 1.

searching a goal-based menu rather than an alphabetized one. Let us further assume that the world model contains a message that was sent only to that user and not to other group members. The Plan Analyst constructs a trace through the goal hierarchy and passes it to the Explainer. The Plan Analyst uses the following rules:

1. If the user model contains a plan P for G, then report `user_plan = P` and user's knowledge of relevant commands.
2. If the expert knowledge base contains a most efficient plan Q for G, then report `best_plan = Q` and user's knowledge of any relevant commands.
3. If the expert knowledge base does not contain P (the user's plan), then report `plan_not_known = P`.

```
type is used to type a sequence of messages on the terminal.  It has
the form:

        type (message_list)

where (message_list) refers to a sequence of messages. type requires
that the contents of the message_list exist, that the user is at read
level and that the messages fit on the screen.  It causes the text of
each message in the message list to be displayed on the screen.  For
example:

        type 1:3

        displays messages 1 through 3.
```

FIGURE 7-8
Text generated to introduce the command type.

4. If `P = Q`, then report `best_plan = user_plan`.
5. If `plan_not_known` is a valid plan, report `plan_not_known`, else report `fault = plan_not_known`.

Three possible responses are illustrated in Fig. 7-9. If the user does not know anything about how to reply to a message, Genie selects a "best" plan based on the context and metaknowledge of links. In this case, the context indicates that the response should be to reply only to the sender, and the metaknowledge indicates that a task should be done now rather than later. Since the user knows about "`compose.message`," the only relevant command is `Reply`.

Figure 7-10 shows how the response agenda for this case is expanded to produce text. If the user has replied to messages in the past and does it efficiently, then Genie simply reminds the user about the command `Reply`. However, if the user seems to know how to reply to messages but does it awkwardly, then Genie introduces a better way. Genie explains why it is better by providing the links between goals of the user's plan and the better plan. Genie considers a plan to be awkward when the user's plan does not match Genie's plan or when the user's plan is not even in the expert knowledge base. The latter case is the last case shown in Fig. 7-9. Here the plan works and is classified as not known, rather than as faulty. Ideally plans that happen to work but are not already in the expert knowledge base would be added automatically.

```
/* user model contains send.mail compose.message */
Plan Analyst output:  user_plan: nil
                      best_plan :reply.to.message -> reply.now ->
                        reply.only.to.sender
                      command: Reply no_knowledge
Explainer output:     SUMMARIZE.PLAN(best_plan)
                      GOAL_INTRODUCE_SIMPLE(reply.to.message)

/* user model contains reply.to.message
   -> reply.now -> reply.only.to.sender */
Plan Analyst output:  best_plan = user_plan
                      best_plan: reply.only.to.sender
                      command: Reply knowledge
Explainer output:     GOAL_REMIND_SIMPLE(reply.only.to.sender)

/* user model contains reply.to.message -> save.message
   -> leave.read.level -> send.message */
Plan Analyst output:  best_plan: reply.now -> reply.only.to.sender
                      plan_not_known: plan -> reply.now
                              ->save.message ->
                              leave.read.level -> send.message
                      command: Reply no_knowledge

Explainer output:
    GOAL_INTRODUCE_COMPLEX(reply.only.to.sender,fault->plan)
```

FIGURE 7-9
Genie's responses to question 2a.

```
In order to reply to a message it is assumed you want to reply right
away and reply only to the sender.  To do this, you must indicate you
wish to reply and compose a message.  You can indicate you wish to
reply by using the command 'Reply'.  For example,

                Reply

would put you in write mode, the receiver of your message would be
identical to the writer of the message you just received.
```

FIGURE 7-10
Text generated to introduce the goal "reply to a message."

A refinement of the second question is: To reply to a group of users, I reply to each individually—is there a better way? This is an instance of the "Given P, is there a better plan for G?" category of Fig. 7-1. In this case the user must identify the question type and select a goal and plan. Let us assume the user selected the goal "`reply.to.all`" and the plan

```
FOR_EACH (x in group)
send.mail.to.individual
```

In the first case in Fig. 7-9, the world model contains a message that was sent to the user and others. In the second case, it contains a message that was sent only to the user. In the third case, the world model does not contain any message.

This question is analyzed using rules 2–5 of the last example. Rule 1 is unnecessary since plan P chosen by the user should be in the user model.

Three possible responses are illustrated in Fig. 7-11. In the first, the message to which the user wishes to reply was addressed to a group of users. Genie chooses to tell the user about the `reply` command since a group exists in the world model. In the second case, the message was addressed only to the user. Genie chooses a plan that requires the user to identify a group of users. In both cases, since the user knows how to send mail, Genie simply reminds the user about how to send mail and describes the links between the suggested plan and the user's. In the third case, the context does not allow a choice between these plans. Genie presents both options. Both plans are preferred to the user's plan because they require less work on the user's part. In the event that the user's plan is equivalent to the suggested solution, Genie would inform the user of this and follow links to justify why the user's plan is best.

In summary, Genie is an intelligent assistant for automated consulting within programming environments. The research focuses on answer generation. Genie's knowledge is separated into two components, a rule base that captures knowledge of how to consult, and a frame-based hierarchical knowledge representation that encodes knowledge of the domain about which to consult—the programming environment (the Berkeley Unix mail system in the example). Users are not categorized along a spectrum of expertise, nor commands along a spectrum of level of difficulty. Instead Genie reflects a task-centered approach where an answer to a question about the environment is based on knowledge of what the user has done in the past and is trying to accomplish now. There are several other research efforts similar to Genie, where the intelligent assistant is essentially passive and answers user questions. Rather than preencode expected plans, however, Grapple (Huff and Lesser, 1988) and Agora (Bisiani et al., 1988) perform planning as needed with respect to the commands provided by the programming environment.

3.2 Marvel

Software systems are getting larger and more complex all the time. Typically, many programmers work together on developing and maintaining a system composed of numerous parts. Each part often has several variants, for instance because of revisions to repair errors or to run on different kinds of computers, which are combined into configurations that select the appropriate

```
/* the world model contains message that was sent to user and others */
Plan Analyst output:
user_plan: reply.to.each.in.group
best_plan: reply.to.all -> reply.group.known
command: reply no_knowledge

Explainer output:
GOAL_REMIND_SIMPLE(reply.to.each.in.group)
GOAL_INTRODUCE_COMPLEX(reply.group.known)
DESCRIBE_LINK(reply.to.each.in.group, reply.group.known)

/* the world model contains message that was just sent to user */
Plan Analyst output:
user_plan: reply.to.each.in.group
best_plan: reply.to.all -> reply.group.create.alias
command: alias no_knowledge

Explainer output:
GOAL_REMIND_SIMPLE(reply.to.each.in.group)
GOAL_INTRODUCE_COMPLEX(reply.group.create.alias)
DESCRIBE_LINK(reply.to.each.in.group, reply.group.create.alias)

/* the world model does not contain explicit reference to a message */
Plan Analyst output:
user_plan: reply.to.each.in.group
best_plan:reply.to.all -> reply.group.create.alias
command: reply no_knowledge
command: alias no_knowledge

Explainer output:
GOAL_REMIND_SIMPLE(reply.to.each.in.group)
GOAL_INTRODUCE_COMPLEX(reply.group.create.alias)
DESCRIBE_LINK(reply.to.each.in.group, reply.group.create.alias)
GOAL_INTRODUCE_COMPLEX(reply.group.known)
DESCRIBE_LINK(reply.to.each.in.group, reply.group.known)
```

FIGURE 7-11
Response to question 2b.

variants of each part of the system. Programmers working on a large-scale system spend a considerable portion of their time coordinating their activities, locating the right system components, and building tools to help them in their efforts. These facts point to the importance of programming environments that manage and automate the "menial" jobs programmers would otherwise do manually.

It is not sufficient to create a single, very powerful programming environment to take care of all these menial tasks. Each software project has its own characteristics: its own organization, its own development method, and its own relations among its components. It is thus necessary for the programming environment to understand these characteristics and behave differently for different characteristics. In other words, a knowledge-based programming environment can provide better assistance than a "moronic" programming environment that treats every development project in the same way.

Many programming environments are tailored to a chosen programming language, support some particular development methodology, and incorporate a selected set of tools. A knowledge-based programming environment, on the other hand, is fairly easy to retarget to another language, another methodology, or new tools by modifying the knowledge base. Marvel (Kaiser et al., 1988a; 1988b) is a knowledge-based programming environment in the sense that its behavior is dictated by the policies set by the manager of each distinct project. Marvel actively participates in the development of a system by automating many of the tasks peculiar to that project as well as those common to a wide range of projects. In order to do this, Marvel needs to know the following:

- The organization of the project including the *classes* or types of software objects, such as source code in the programming language, binary machine code, and text documentation, and the valid operations on these objects, such as compilers, loaders, and word processors.
- The relations among the various objects input and produced by the project—for example, one object is a variant of another and uses a third.
- The rules specific to the chosen development process, as determined partially by management and partially by the requirements of the software development tools available. Marvel rules are based on the rules of production systems.

All three factors may change over the project's lifetime, so Marvel must be able to adjust dynamically.

Marvel uses this knowledge to tailor itself to the specific needs of each individual project. For example, the team of programmers working on project A might need a set of commands C(A) while another team working on project B might need another set of commands C(B). If Marvel knows how the commands in each set interact among themselves and how they manipulate the different components of systems A and B, respectively, it can participate in the development of A and B in different ways. Marvel would automatically invoke

commands from the set C(A) only with project A, and only in accordance with the rules that define A's development process. It can do the same for B.

3.2.1 ORGANIZATION OF PROGRAMMING ENVIRONMENTS. A Marvel programming environment is created by tailoring a standard kernel to the policies desired for the system being developed or maintained. This is done in two phases, as illustrated in Fig. 7-12. First, a skilled user called the *superuser* writes a description of the project using a special notation called the Marvel Strategy Language (MSL). This description specifies the organization of the software project (e.g., a program is made up of modules and procedures) and models the process of development of that particular project (e.g., a procedure can be printed only after it has been formatted). Any user can then load this description into Marvel and start using this instantiated Marvel to work on the project. Marvel actively participates in the development of the project by using the description it was fed, organizing the components of the project in the way requested, and automatically invoking the appropriate commands at the right times.

For example, a superuser might write a description for an arbitrary C system, stating that any such project would include a number of module (source file) objects, whose attributes (subparts) include macros, types, variables, and procedures. The description might also state how to invoke commands on the corresponding objects to do C-specific type checking, compilation, and so on. It could further state that this project requires programmers to sign off all program modifications with their respective managers, and the environment should automate this communication task by sending electronic mail after a programmer completes each modification task.

The organization of a project is the way that the various software artifacts are set up in relation to each other. Marvel uses an *objectbase* to main-

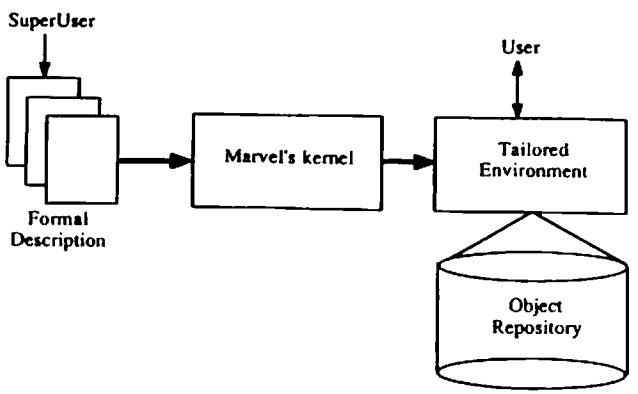

FIGURE 7-12
Marvel organization.

tain all the software artifacts that are components of the software system itself or are used in its development, including mechanisms for invoking external tools such as the editors, compilers, and mail systems represented by commands. Each object in the objectbase is an instance of a class (that is, the class defines the type of the object). The organization of the project includes all the classes and the *syntactic relations* among the various classes. Figure 7-13 shows three classes of objects, PROJECT, MODULE, and PROCEDURE. From this declaration, Marvel deduces that instances of the MODULE class are enclosed in PROJECT objects.

```
objectbase:

    PROJECT:: = superclasses: ENTITY:
        printname : string ;
        status : (Linked, NotLinked) = ''NotLinked'';
        timestamp : real = ''0.0''
        mods : set_of MODULE ;
        executable : binary ;
    END

    MODULE:: = superclasses: ENTITY:
        status : (ModIsComp,ModInComp,ModNoComp) =
               ''ModNoComp'';
        Procs : set_of PROCEDURE;
    END

    PROCEDURE:: = superclasses: ENTITY:
        analyzed : (Analyzed,InAnalysis,NotAnalyzed) =
                 ''NotAnalyzed'';
        edited : (Edited,NotEdited) = ''NotEdited'';
        state : (Changed,NotChanged) = ''NotChanged'';
    END
Relations:

    inproject: PROJECT MODULE
    exp: MODULE PROCEDURE
    imp: MODULE PROCEDURE
    contains: MODULE PROCEDURE

/* Each class consists of its name, a list of zero or more
superclasses (ENTITY is the built-in root class), and a list of zero
or more attributes. Class names are in UPPERCASE. Attributes are
typed, and may be initialized. The type is either a class name, a
built-in type given in italics, a constructor (none shown), or an
enumerated set of values. Relations consist of their names and the
names of two classes, since only binary relations are supported
currently. MSL keywords are only underlined.*/
```

FIGURE 7-13
Classes and relations.

Besides syntactic relations among classes of objects, components of a software project can be related by *external* relations. Unlike syntactic relations, these do not imply any particular organization on the participating components. The relations are simply stored in the objectbase to maintain information that cannot be deduced from the hierarchical structure. Using the example in Fig. 7-13, a relation named `contains` maps the MODULE class to the PROCEDURE class. An instance of this relation, `contains(m,p)`, would be stored in the objectbase.

The syntactic and external relations define how the project is set up. However, Marvel must also understand the model of development and/or maintenance desired for the system. Marvel needs to know the capabilities of each object, that is, the operations that can be performed on the object, including the activities that affect more than one object.

3.2.2 OPPORTUNISTIC PROCESSING IN PROGRAMMING ENVIRONMENTS. One aspect of intelligent behavior by a programming environment is understanding the process of software development and maintenance. This requires understanding the activities that can be performed to transform the system from one stage in its life cycle to another. For example, Marvel needs to know how to add a new object to the project. If the team is developing a C program, Marvel needs to know how to add a new procedure definition in terms of how the procedure relates to all the existing objects, including how to place it in the source file representing its module, what other modules (files) to recompile, and so on. The *rules* part of strategies instantiates Marvel with the model of the development process. Rules formalize the application of one object (a command) to another (a software artifact) and define Marvel's automatic behavior.

Marvel rules are based on those of expert systems but differ from most others by having three parts instead of the conventional condition–action pair. Marvel rules have preconditions, an activity, and a set of postconditions, where preconditions are more or less the same as the traditional condition and the activity together with the postconditions roughly correspond to the action. As in expert systems, preconditions and postconditions are written in the first-order predicate calculus.

One big difference is that expert system rules have a single action, deterministically selected by the condition. Marvel rules, on the other hand, must have multiple postconditions, since it is impossible for the condition to uniquely determine the result. A programming activity might have a set of postconditions, exactly one of which is true after the activity terminates. The processing performed by the command (tool) invoked in the activity determines which of the postconditions is true. For example, a compiler might produce either error messages or object code, but which one cannot be determined except by running the compiler. This notion of multiple postconditions distinguishes Marvel's rule base from most other rule-based systems. Through chaining the preconditions and postconditions of several rules, Marvel performs what is called *opportunistic processing*, because Marvel carries out

chores as the opportunity arises. Marvel uses both *forward chaining* and *backward chaining* to invoke commands automatically, switching from one to the other as explained below.

The typical operation of opportunistic processing is as follows. The user requests some command. Marvel checks whether the precondition of the command is already satisfied (i.e., it evaluates to "true"). If so, Marvel invokes the command. If not, Marvel attempts to do whatever is necessary to satisfy the precondition. This involves finding one or more other rules whose postconditions may change the state of the project in such a way as to make the precondition true. So Marvel tries to invoke the command in the activity part of these rules. But before it can do that, the preconditions of these rules must be true. So Marvel applies another round of backward chaining. Eventually, Marvel will either satisfy the original precondition and invoke the command originally requested by the user, or it will explain to the user why it is impossible to do so given the rules and what the user has to do to fix things.

After Marvel has invoked a command, the correct one of its several postconditions is asserted, changing the state of the project—usually in some small way, just new values for a few attributes. But the postcondition may cause the preconditions of one or more other rules to now be satisfied, so Marvel can go ahead and automatically carry out the corresponding activity (that is, invoke its command), under the assumption that the user will soon want its results. These commands have their own postconditions that may make true the preconditions of other commands, so forward chaining repeats until Marvel runs out of things to do—all preconditions in the entire rule base are now unsatisfied (false).

For example, the compile rule in Fig. 7-14 states that the compiler can be applied to a MODULE object only after checking that there exists a PROCEDURE object that is a component of this module and that has been edited and analyzed successfully since the last time the module was compiled. The result of applying the compile command could be either errors or successful compilation into machine code.

3.2.3 STRATEGIES. *Strategies* are metadescriptions of a project or a family of similar projects. A strategy encapsulates information about classes of objects, their capabilities, the external relations among them, and the rules guiding their manipulation as part of software development activities. Each strategy consists of four parts. The first describes the interface between this and other strategies by means of imports and exports. The second part describes a view of the objectbase by specifying classes of objects with their operations and syntactic relations. The third part defines the external relations among instances of these classes by declaring the name of each relation and the domain and range classes. The last part of a strategy is the rules that model the software development process.

Rules:

```
edit[?p:PROCEDURE]:

        ( EDITOR edit ?p )

        (?p.edited = Edited)

analyze[?p:PROCEDURE]:
        IF (?p.edited = Edited)

        ( ANALYZER analyze ?p )

           (?p.analyzed = Analyzed) AND
           (?p.state = Changed) AND
           (?p.edited = NotEdited)
        OR
           (?p.analyzed = NotAnalyzed) AND
           (?p.edited = NotEdited)

compile[?m:MODULE]:
        (forall PROCEDURE ?p) such that
            (member (?m.procs ?p))
        :
        (?p.analyzed = Analyzed) AND
        (?m.status = ModNoComp)

        ( COMPILER compile ?m )

        (?m.status = ModIsComp)
        OR
        (?m.status = ModNoComp)

profile[?proj:PROJECT]:

        ( PROFILER type_info ?p )

build[?proj:PROJECT]:
        (forall MODULE ?m) such that
            (member (?proj.mods ?m))
        :
        (?m.status = ModIsComp)

        ( BUILDER build ?proj )

        (?proj.status = Linked)
        OR
        (?proj.status = NotLinked)

  /* A rule consists of its name, formal parameters with their types, and
then its body.  The body consists of a precondition indicated in a variant of
first order predicate logic, an activity, and one or more postconditions
separated by "OR".  The activity consists of the name of the tool, the
particular operation to be carried out by the tool, and its arguments.  "="
and similar operators are provided by MSL. */
```

FIGURE 7-14
Rules.

Strategies combine three major concepts:

1. Classes of objects and multiple inheritance similar to object-oriented languages
2. Rules that are similar but not identical to rules in production systems
3. Modularization and information hiding

A single strategy might provide only a partial view of the project. Then the complete description of the project is captured in a collection of interacting strategies in a way similar to modules or packages in a conventional programming language. Modularization has become a standard concept in large-scale programming, the basic idea being that programmers work on one piece at a time and then put together all the pieces. The same concept was followed in designing strategies. Different superusers, or the same superuser at different times, develop a set of strategies where each strategy encapsulates a single role of a class of team member, part of the development process, or subpart of the software system.

Each strategy requires other strategies and uses their exported *facilities*. These facilities include both class definitions and specific objects—commands—such as a performance profiler object. Consider a set of two strategies: a programming language strategy and a programming environment strategy. The Clanguage strategy defines the syntax and semantics of an extension of C, as depicted in Fig. 7-15. A C module consists of a set of procedures. Each procedure has a source code segment and an object code segment. Statements, expressions, and such are ignored for simplicity.

The Cenvironment strategy of Fig. 7-16 combines the MODULE class imported from the Clanguage strategy with the separate MODULE class defined locally. The result is the internal representation of a single MODULE class defined according to the *union* of the Cenvironment and Clanguage definitions of MODULE. Objects in the combined class may appear as elements of the mods attribute of PROJECT objects and any other places in the objectbase where instances of the MODULE class are expected. Both EDITOR and COMPILER specialize the TOOL class by giving a command string for invoking themselves. Note that both EDITOR and COMPILER are specific objects rather than classes of objects.

Strategies can be reused by other strategies, due to the information hiding and strict interfaces. The ease of reusability is enhanced by building a library of strategies, where each strategy is categorized according to the task(s) it performs. For example, strategies that define programming languages are all grouped together while those describing memory management techniques are also grouped together, leading to a lattice of strategies. The interface to the library includes methods for depositing a new strategy, retrieving an existing strategy, and searching for strategies that perform a specific task.


```
STRATEGY: Clanguage;

Interface:
    Imports: none;
    Exports: all;

Objectbase:

    MODULE::= superclasses: ENTITY:
        printname : string ;
        procs : set_of PROCEDURE ;
        exportlist : set_of EXPITEM ;
        importlist : set_of IMPITEM ;
    END

    PROCEDURE::= superclasses: ENTITY:
        printname : string ;
        code_c : text ;
        code_o : binary ;
    END

    EXPITEM::= superclasses: ENTITY:
        expelem : PROCEDURE ;
    END

    IMPITEM::= superclasses: ENTITY:
        impelem : PROCEDURE ;
        module : MODULE ;
    END

END Clanguage

  /* None and all have the obvious meaning.  In the general case, Imports
lists the names of imported strategies while Exports lists the names of
exported classes, relations, tools and rules. */
```

FIGURE 7-15
Strategy.

3.2.4 MERGING STRATEGIES. When two or more strategies are loaded into Marvel, they are merged into a single internal representation that is treated as if it were based on a single strategy. As mentioned earlier, a strategy provides a view of the objectbase. When several strategies are active (i.e., loaded), there are several views of the same objectbase. These views have to be merged into one unified, composite view. Merging of strategies implies merging of classes of objects, merging of relations, and merging of rules. If the strategies being merged do not overlap, their merger is simply the aggregate of all parts of all strategies. Two strategies are said to be overlapping if they contain

- Classes with the same name
- Relations with the same name
- Rules with the same name, same preconditions, or same activity

Merging overlapping strategies involves checking the consistency of overlapping class definitions, overlapping relation definitions, and potentially

```
STRATEGY: CEnvironment;

Interface:
    Imports: Clanguage;
    Exports: all;

Objectbase:

    PROJECT::= superclasses: ENTITY:
        printname : string ;
        status : (Linked,NotLinked) = "NotLinked";
        timestamp : real = "0.0";
        mods : set_of MODULE ;
        executable : binary ;
    END

    MODULE::= superclasses: ENTITY:
        status : (ModIsComp,ModInComp,ModNoComp) =
                    "ModNoComp";
    END

    PROCEDURE::= superclasses: ENTITY:
        analyzed : (Analyzed,InAnalysis,NotAnalyzed) =
                    "NotAnalyzed";
        edited : (Edited,NotEdited) = "NotEdited";
        state : (Changed,NotChanged) = "NotChanged";
    END

Relations:

    inproject: PROJECT MODULE
    exp: MODULE PROCEDURE
    imp: MODULE MODULE
    contains: MODULE PROCEDURE

    COMPILER:: superclasses: TOOL:
        operations:
                compile : string = "compile";
    END

    ANALYZER:: superclasses: TOOL:
        operations:
                analyze : string = "analyze";
    END

    BUILDER:: superclasses: TOOL:
        operations:
                build : string = "build";
    END

    EDITOR:: superclasses: TOOL:
        operations:
                edit : string = "runeditor";
    END

    PROFILER:: superclasses: TOOL:
        operations:
```

FIGURE 7-16
Another strategy.


```
                type_info : string = "profiler";
    END

Rules:

    edit[?p:PROCEDURE]:

            ( EDITOR edit ?p )

            (?p.edited = Edited)

    analyze[?p:PROCEDURE]:
            IF (?p.edited = Edited)

            ( ANALYZER analyze ?p )

               (?p.analyzed = Analyzed) AND
               (?p.state = Changed) AND
               (?p.edited = NotEdited)
            OR
               (?p.analyzed = NotAnalyzed) AND
               (?p.edited = NotEdited)


    compile[?m:MODULE]:
            (forall PROCEDURE ?p) such that
                (member (?m.procs ?p))
            :
            (?p.analyzed = Analyzed) AND
            (?m.status = ModNoComp)

            ( COMPILER compile ?m )

            (?m.status = ModIsComp)
            OR
            (?m.status = ModNoComp)

    profile[?proj:PROJECT]:

            ( PROFILER type_info ?p )

    build[?proj:PROJECT]:
            (forall MODULE ?m) such that
                (member (?proj.mods ?m))
            :
            (?m.status = ModIsComp)

            ( BUILDER build ?proj )

            (?proj.status = Linked)
            OR
            (?proj.status = NotLinked)

END CEnvironment

  /* TOOL is a built-in class.  Each tool lists its operations (only one each
shown here), each with an indication of what the user has to do to invoke the
operation. */
```

FIGURE 7-16 (*cont.*)

contradictory rules. Unloading of strategies, on the other hand, may cause portions of the objectbase to become inaccessible. If a strategy was loaded in the first place only because it was imported by the strategy now being unloaded, it will be unloaded also.

When two overlapping strategies are merged, a consistency checker verifies that the overlapping items can be unified. If two object classes in two strategies have the same name, they can be unified if their sets of attributes are disjoint or if attributes having the same name are identical. Two attributes are identical if they have the same name and the same type. Similarly, two relations are identical if they have identical domain and range classes. Checking the consistency of rules is much harder, since multiple rules with the same activity but different preconditions and postconditions are not necessarily conflicting. In particular, Marvel checks only that their preconditions and postconditions, respectively, are not obviously contradictory.

For example, consider three strategies A, B, and C. B imports some facilities from C, and both A and B define a class called X. A defines X as having two attributes *att1* and *att2*, while B defines X as having only one attribute, *att3*. If the user loads only strategy A, any instance of class X will have only two attributes, *att1* and *att2*. However, if she later loads strategy B, all instances of X are updated to have a third attribute *att3*. Also, strategy C is loaded automatically because it is used by B. The rules available at this point on instances of X assume that X has three attributes. However, rules that were defined in strategy A operate only on *att1* and *att2*, while rules defined in B assume only the existence of *att3*. Now if strategy B is unloaded, C is also. Furthermore, *att3* will no longer be accessible. This does not mean that *att3* is deleted from all instances of class X, but rather that the current rules (i.e., those defined in strategy A) cannot access *att3*. Thus, *att3* will be "unused" until B is reloaded; *att3* retains its previous value.

Merging rules from different strategies is more complicated. There are several issues. First, if two or more rules invoke the same activity, how does one combine their preconditions and their postconditions? Second, if several rules invoke distinct activities but they have the same preconditions, which of them is invoked if the precondition becomes true? Third, if several rules have the same postcondition but invoke distinct activities, which does one invoke during backward chaining?

The first problem has two solutions: Marvel can either AND all the preconditions or it can OR them. The example in Fig. 7-17 depicts two rules in two strategies. Both rules have the same activity (build system). When Marvel merges these two rules, it can build the system either when all the procedures are compiled AND analyzed or when all the procedures are either compiled OR analyzed. Since the former is probably what is intended, the default interpretation ANDs all the preconditions for the same activity; Marvel allows the user to change the interpretation to OR.

In summary, Marvel defines a methodology for acquiring the knowledge required by knowledge-based programming environments. The methodology


```
STRATEGY: A;
   ...

   Rules:

    build[?proj:PROJECT]:
            (forall MODULE ?m) such that
                (member (?proj.mods ?m))
            :
            (?m.status = ModIsComp)

            ( BUILDER build ?proj )

            (?proj.status = Linked)
            OR
            (?proj.status = NotLinked)

END A


STRATEGY: B;
   ...

   Rules:

    build[?proj:PROJECT]:
            IF (?proj = ?current_focus)

            ( BUILDER build ?proj )

            (?proj.status = Linked)
            OR
            (?proj.status = NotLinked)

END B
```

FIGURE 7-17
Overlapping rules in two strategies.

consists of a notation for describing families of software projects and a kernel that is instantiated by the description. The modular units of the language is strategies, which describe the classes of objects making up the objectbase, the relations among these objects, and the rules for manipulating objects and applying commands to them in the process of developing and maintaining the system. The kernel tailors its behavior according to the description and thus provides the user with a specialized environment that provides more intelligent assistance than previous programming environments.

Strategies are merged to give a complete description of the objectbase and the development process at any time. Marvel adjusts dynamically to the loading and unloading of strategies, which may cause the perceived state of the objects to change according to the role of the user or the phase in the project's life cycle. Changes may be made to capabilities of objects, types of attributes of objects, and interactions among objects (most notably, automated application of command objects to other objects). Changes are propagated throughout the objectbase to ensure that all the objects possess a consistent view of the objectbase. In particular, only those parts of a class definition that are defined in the currently loaded strategies are visible.

The approach is novel in that Marvel handles the incorporation of new commands without modifying the kernel or the tool executed by the command and without physically moving the tool into the objectbase. A strategy simply describes how to locate the tool externally and how it interacts with existing objects before incorporating it in the objectbase.

Smile (Kaiser and Feiler, 1987) and the Common Lisp Framework (CLF) (Balzer, 1987) are the immediate ancestors of Marvel. Smile is a hand-coded programming environment for C that behaves as an intelligent assistant, actively participating in the development process, but it cannot be modified without recoding. CLF is a rule-based programming environment for Common Lisp that supports forward chaining but not (directly) backward chaining. The rules are written in AP5 (Cohen, 1986), a logical specification notation that can be efficiently compiled into Lisp. It is possible to simulate Marvel-like behavior in CLF, but CLF is limited by its Lisp orientation and implementation and cannot integrate external tools.

Darwin (Minsky, 1985) is another rule-based programming environment that supports backward chaining. Darwin does not actively participate in software development in the sense of automatically carrying out activities on behalf of the user, but instead it monitors the user's activities and will not permit him to do anything against the policies dictated by the project manager and encoded in the rules. This kind of behavior can be simulated in Marvel using restrictive preconditions as constraints but is outside its normal scope of behavior. It would be interesting to combine Darwin and Marvel capabilities in a more general intelligent assistant.

Finally, Inscape (Perry, 1989) is a programming environment that combines a form of automatic programming with a form of intelligent assistance. Inscape supports the creative labors of its users—programming—but uses facilities more in line with intelligent assistance rather than automatic programming. It operates in a monitoring mode similar to Marvel, always looking over the programmer's shoulder and joining in when appropriate.

Inscape provides a special notation for specifying, for each subroutine, its preconditions (things that must be true before executing the subroutine), postconditions (things that become true by virtue of executing the subroutine), and obligations (things that must be done later on because the subroutine has been executed). As a program is written and modified, Inscape checks whether all preconditions and obligations have been satisfied by previous and subsequent postconditions, respectively. This is done using a relaxed form of theorem proving. This checking is similar to but much more significant than the symbol resolution, type checking, anomaly detection, and so on of typical language-based editors such as the Cornell Program Synthesizer (Teitelbaum and Reps, 1981), since semantic as well as syntactic errors can be detected. Inscape has been implemented for the C programming language, but the ideas could be applied to any procedural language.

4 CONCLUSIONS

As discussed in the introduction, applications of artificial intelligence to software engineering have tended in two directions, automatic programming and intelligent assistance. Automatic programming in its full glory is a very long term goal, although at least one commercial product is already available. Much additional work is needed in this field, and the notion of reverse engineering of existing programs seems particularly important.

Intelligent assistance is a more immediately practical approach for improving the productivity of individual programmers as well as the quality of their programs. The next major research problem is to apply intelligent assistance to coordinating a full software development team rather than interacting with just one user at a time. In the meantime, commercial exploitation of intelligent assistance seems imminent for single-user programming environments.

There is one catch, however: user acceptance of intelligent assistance. Many programmers consider themselves pioneers fighting a rugged frontier, making the computer do what they want it to do. The notion of the computer telling the programmers what to do, as in Genie, or the computer just going ahead and doing things for them, as in Marvel, is likely to meet with initial resistance. Marketers of intelligent assistance products must be highly concerned with human–computer interaction, and managers of software engineers must be extremely careful in the introduction of intelligent assistance tools into their workplace. A technology that is not used will never achieve its promise for higher programmer productivity and higher program quality.

ACKNOWLEDGMENTS

The Genie examples were developed by Ursula Wolz and the Marvel examples by Nasser Barghouti and Mike Sokolsky. Professor Kaiser's Programming Systems group is supported by National Science Foundation grants CCR-8858029 and CCR-8802741, by grants from AT&T, DEC, IBM, Siemens, Sun, and Xerox, by the Center for Advanced Technology, and by the Center for Telecommunications Research.

REFERENCES

Balzer, R. (1985). A 15 Year Perspective on Automatic Programming. *IEEE Trans. Software Eng.* SE-11(11):1257–1268.

Balzer, R. M. (1987). Living in the Next Generation Operating System. *IEEE Software*, November, pp. 77–85.

Bisiani, R., Lecouat, F., and Ambriola, V. (1988). A Planner for the Automation of Programming Environment Tasks. In: 21st Annual Hawaii Int. Conf. on System Sciences (Bruce D. Shriver, Ed.), pp. 64–72.

Broy, M., and Pepper, P. (1981). Program Development as a Formal Activity. *IEEE Trans. Software Eng.* SE-7(1):14–22.

Cheatham, T., Townley, J., and Holloway, G. (1979). A System for Program Refinement. 4th Int. Conf. on Software Engineering, pp. 53–62.
Cohen, D. (1986). Automatic Compilation of Logical Specifications into Efficient Programs. 5th Nat. Conf. on Artificial Intelligence, pp. 20–25.
Freeman, P. (1987). A Conceptual Analysis of the Draco Approach to Constructing Software Systems. *IEEE Trans. Software Eng.* SE-13(7):830–844.
Goldberg, A. T. (1986). Knowledge-Based Programming: A Survey of Program Design and Construction Techniques. *IEEE Trans. Software Eng.* SE-12(7):752–768.
Huff, K. E., and Lesser, V. R. (1988). A Plan-Based Intelligent Assistant That Supports the Software Development Process. In: ACM SIGSoft/SIGPlan Software Engineering Symposium on Practical Software Development Environments (Peter Henderson, Ed.), pp. 97–106.
Kaiser, G. E., and Feiler, P. H. (1987). Intelligent Assistance Without Artificial Intelligence. 32nd IEEE Computer Society Int. Conf., pp. 236–241.
Kaiser, G. E., Barghouti, N. S., Feiler, P. H., and Schwanke, R. W. (1988a). Database Support for Knowledge-Based Engineering Environments. *IEEE Expert* 3(2):18–32.
Kaiser, G. E., Feiler, P. H., and Popovich, S. S. (1988b). Intelligent Assistance for Software Development and Maintenance. *IEEE Software*, May, pp. 40–49.
Kedzierski, B. L. (1984). Knowledge-Based Project Management and Communication Support in a System Development Environment. 4th Jerusalem Conf. on Information Technology.
Minsky, N. H. (1985). Controlling the Evolution of Large Scale Software Systems. Conf. on Software Maintenance—1985, pp. 50–58.
Perry, D. E. (1989). The Inscape Environment. 11th Int. Conf. on Software Engineering, pp. 2–9.
Rich, C., and Waters, R. C. (1988a). The Programmer's Apprentice: A Research Overview. *Computer* 21(11):10–25.
Rich, C., and Waters, R. C. (1988b). Automatic Programming: Myths and Prospects. *Computer* 21(8):40–51.
Smith, D. R., Kotik, G. B., and Westfold, S. J. (1985). Research on Knowledge-Based Software Environments at Kestrel Institute. *IEEE Trans. Software Eng.* SE-11(11):1278–1295.
Stallman, R. M. (1981). Emacs—The Extensible, Customizable, Self-Documenting Display Editor. SIGPlan SIGOA Symp. on Text Manipulation, pp. 147–156.
Teitelbaum, T., and Reps, T. (1981). The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Commun. ACM* 24(9):563–573.
Waters, R. C. (1986). KBEmacs: Where's the AI? *AI Mag.* VII(1):47–56.
Wolz, U. (1988). Automated Consulting for Extending User Expertise in Interactive Environments: A Task Centered Approach. Columbia Univ. Department of Computer Science, Tech. Rep. CUCS-393-88.
Wolz, U., and Kaiser, G. E. (1988). A Discourse-Based Consultant for Interactive Environments. 4th IEEE Conf. on Artificial Intelligence Applications, pp. 28–33.

CHAPTER 8

KNOWLEDGE-BASED VISION SYSTEMS

M. G. RODD

1 INTRODUCTION

Computer vision systems set out to replicate, to some extent, our powerful human ability to recognize and classify visually acquired images, or scenes. Inherently they form a major component in the general category of scientific endeavor referred to as artificial intelligence. Also, inherently, one accepts them to be knowledge-based—although it has been argued that early computer vision systems, like classic, sequentially programmed computers, only made use of implicit knowledge, essentially that of the designer! This chapter takes a pragmatic, user-oriented view of so-called knowledge-based computer vision systems—a viewpoint highly biased toward the practical applications of the technology as a component in a manufacturing, or process-plant, control system. It investigates the characteristics of this area of application and defines the potential role of vision-based systems. On the basis of this, it reviews current progress in fulfilling these roles. It highlights critical areas, especially the need to recognize that the industrial environment requires solutions that can coexist in those very environments and can react at speeds that are acceptable in the closed-loop control situation of which they form a part. It is argued strongly that simply gluing together well-understood low-level processing systems and currently available expert-systems-based high-level processing is bound to end in disaster! In practice, a total systems approach must be made, recognizing the realities of the application, including the still-unresolved problems relating to knowledge acquisition. Low- and high-level processing are integrated into highly flexible, explicit knowledge-based solutions: industrially

KNOWLEDGE ENGINEERING

Volume II APPLICATIONS

Hojjat Adeli, Editor

Department of Civil Engineering
The Ohio State University

McGraw-Hill Publishing Company

New York St. Louis San Francisco Auckland Bogotá Caracas
Hamburg Lisbon London Madrid Mexico Milan Montreal New Delhi
Oklahoma City Paris San Juan São Paulo Singapore Sydney Tokyo Toronto