REASON: An intelligent user assistant for interactive environments

by J. M. Prager
D. M. Lamberti
D. L. Gardner
S. R. Balzac

The provision of intelligent user assistance has been an ongoing problem in designing computer interfaces. Interactive computing environments must support expert as well as novice users when providing advice for error correction and answers to questions directed to a system. To address these issues, we have investigated the application of fairly well-understood artificial intelligence techniques in novel ways to provide intelligent help. This paper describes the design methodology used to build REASON (Real-time Explanation And SuggestiON), an intelligent user-assistant prototype for a windowed, multitasking environment. REASON's central component is an inference engine that solves problems arising from a user's activity. When the user makes one of several different kinds of errors, the inference engine offers dynamically generated suggestions about what the user might have intended. The user can also query REASON using natural language. In addition to providing suggestions of corrected input or answers to questions, REASON can provide two complementary types of explanations of these responses, derived from the inferences that led to them.

Much of the recent work in designing help systems for computer users has been influenced by the difficulties that people have in learning how to interact with computers.[1-4] However, one of the most common, yet arguably least successful, computer applications is on-line user assistance.

In studying new user interface technology, there is a considerable base of work in the areas of contextual assistance, user modeling, planning and problem solving, and natural-language processing. Much research work has been undertaken investigating the role that each plays in providing on-line assistance for computer users. The focus of most research projects on advisory systems has been on depth of investigation of a particular component, rather than integrating multiple components to address the problem of user assistance.[5-8] These projects fall primarily into two categories: (1) projects that have been the source of exciting speculation as opposed to useful technology, and (2) projects that have been tightly bound to real-world tasks or the laboratory, and thus rarely press forward the fundamental issues that comprise the central goals of artificial intelligence (AI). Furthermore, the effectiveness of most systems developed in research environments has not been studied empirically beyond prototyping functions to demonstrate research propositions. For the most part, no feedback on performance has been acquired from users. This situation contributes to the fact that there are essentially no commercially available systems that integrate these features.

Context-dependent help is a popular issue in the current literature. Context dependency usually refers

to a discourse-based assistance that is tailored to a user's plans for best accomplishing goals. Inferencing strategies are used to identify such plans and generate advice at the appropriate level of detail. For a more extensive discussion of context dependency refer to References 9 and 10. Prototype systems (e.g., the TOPS-20 operating system[11]) have been developed that incorporate contextual help, using plan recognition based on AI methods.[8,12,13] Information about the current state of the interaction can be represented by a plan or plan hypothesis that may be only partially developed and instantiated. For example, a system known as the system architect's apprentice (SARA) employs a technique whereby help is integrated into the grammar and then processed by a combined parser generator and an integral help generator.[14] The integral help generator has a concise representation of the user interface available to it, making contextual help easier to generate. This technique provides for consistency and accuracy of syntactic assistance at a lower level and more in-depth information at a higher level.

Despite the attention given to context-dependent assistance, it is not easy to see how the various techniques offered can solve the problems associated with reliably determining users' plans. This is partly because it is difficult to identify and map plan-driven behavior to context-dependent advice, and partly because most models of question-answering are more inclined towards database interrogation than requests for help or explanation.[9] Also, much of the recent work on contextual assistance identifies the importance of plans but fails to include in knowledge bases explicit discourse information needed to satisfy pedagogical goals.

Similarly, the development of predictive user models has been seen as critical for advisory systems.[15] Quinn and Russell[16] point out that the value of an intelligent interface is extremely limited if it is not based on a strong model of the user. To a large extent, the work on user modeling has presumed that users are homogeneous in relevant ways. Although it is true that, for the majority of users, a system built on such a principle is better than it would have been without homogeneity assumptions, it is not true that such a system is likely to be the best that could be produced.

Despite this emphasis, some researchers have attempted to individualize user models. In an interactive help facility for Scribe,[17] which is a document formatter, Rich[18] has incorporated a user model based on patterns of user commands. In Scribe, the appropriate level of a question response is a function of the level of the question itself and the level of knowledge of the user who asked the question. In presenting the correct level of explanation, the system maintains a dictionary that contains an entry for each of the things that can occur in a set of condition-action rules describing knowledge about Scribe. Associated with each entry is information that describes when it may be appropriate to mention the associated concepts in an explanation. In answering a specific question, the system locates the rule(s) that apply and compares what it knows about the concepts in the rule(s) to what it knows about the user, based on patterns of commands issued by the user during a period of time.

The feasibility of a more natural approach to user-computer interaction is usually shown by building and demonstrating a prototype system.

Scribe exemplifies a technique for user modeling that is based on inferring a user's skill level and specific problems and errors from actions and responses. This technique reflects a desire to place most of the burden of constructing the model on the system and thus raises concerns regarding the reliability of user classifications.

The feasibility of a more natural approach to user-computer interaction is usually shown by building and demonstrating a prototype system whose aim is to minimize the training required to interact effectively and efficiently with a computer. To most persons, this means supplying a system that allows the use of the words and syntax of a language used in common discourse, such as standard English.[19–22] Most natural-language systems exist as large-scale prototypes that can recognize and interpret fairly extensive vocabularies and sentence structures.[23–26] Unfortunately, very few of these prototype systems have been evaluated by actually measuring user per-

formance through extended system usage. Most of the commercially available systems have not been on the market long enough to have been thoroughly evaluated. Thus, it is nearly impossible to make empirically reliable conclusions for or against any particular commercially available natural-language technology.

In each of the aforementioned areas of research, the emphasis has been on developing state-of-the-art techniques to implement theoretical propositions regarding the type and amount of assistance users need when attempting to perform an action on the computer. Numerous programming methodologies, including AI techniques, have been applied to solving the problem of "intelligent help." Most of the focus has been on perfecting the various techniques to a level of depth that advances scientific inquiry. Although these aims have been well-appreciated and justified, little if any technology currently exists that effectively integrates these well-documented techniques into a modular system that addresses the needs of a spectrum of users ranging from the computer novice to experienced programmers. The opportunity for the practical implementation of these concepts and techniques to provide contextual user assistance currently exists and needs to be addressed.

In general, and especially in the AI realm, there has been a slow but pervasive recognition of the fact that the scientific advancement of programming techniques has overshadowed a realistic assessment of the need for enhanced performance and usability in computer systems. This reckoning leads to the realization that in the past much of the research work on designing intelligent help systems has focused on solving AI problems to the exclusion of practical concerns about implementing systems in a robust manner.

Our work should be viewed as an extension of prior work that has been done on contextual user assistance, using natural language as a means of user-computer discourse. The purpose of our work is to enhance the helpfulness of a computer system to users through the integration and application of several well-understood AI techniques in solving real problems stemming from the usage of commercial systems. An eminent outcome of this effort is the design of *intelligent command lines* that lend themselves to a practical implementation for commercial systems. It is our intention to ensure that this design be fully conforming to the IBM Systems Application Architecture (SAA) and Common User Access (CUA) standards.[27] These standards govern software interfaces, protocols, and conventions for human interaction with applications and system services, communication mechanisms that interconnect SAA systems, and interfaces for program development.[28]

A primary distinction between our work and that of past researchers is that we are focusing on breadth of system function. Specifically, we concentrate on an interactive computing environment in which there often is not only more than one way to explain something, but also more than one way to do something. We take a user's goal-centered approach to problem solving. Therefore, deducing the user's goals is intrinsic to our system. In this approach, the context of interaction is a determining factor in the generation process that produces the form and content of the system's suggestion(s) and explanation(s). Our suggestions are dynamically generated and automatically supplemented by complementary forms of explanation based on a model of the user. Our objective is to maximize the flexibility of the user's interaction with the help function through mixed interaction modes and to tailor advice to the specifics of error conditions. This objective leads to a second aspect of the work that focuses on natural language as an input medium. In a truly interactive computing environment, user assistance based on a human-advisor discourse model needs to be addressed, and natural language is a clear choice.

The theme of this paper is fourfold: to review the theoretical basis, design organization, functional components, and development process of the REASON (Real-time Explanation And SuggestiON) system. We first highlight some of the critical issues involved in identifying areas in which users need intelligent help when using an interactive system. We describe the basis for the approach we chose to implement our intelligent help system. An overview discusses the conceptual design model as implemented in REASON. This section presents a functional description of the components of the REASON system and gives a description of the operation of the working components, highlighting the implementation choices for our technology. We concentrate on the design methodology for building an intelligent user assistant for a command-oriented system, such as an operating system. The operations of system components are explained using a detailed example of a typical user interaction with an operating system. We conclude by summarizing the approach to online user assistance that we have chosen and present our plans for future enhancements.

Critical issues in addressing user errors and queries

In providing user assistance, there are essentially two domains for which computer users need support: errors in command usage and requests for help information. A conflict arises between creating an environment simple enough for a novice (i.e., a user with limited knowledge of computers in general or within a particular domain) and yet sophisticated enough to accommodate an expert. There is a wide continuum of skills between those of a novice, who knows only the rudiments of a system, and those of an expert, who has mastery over it. The novice is constantly learning about the purpose of specific functions and their interrelationships with other functions and is usually faced with the burden of what to learn and how to locate the necessary information needed to accomplish a task.

Computer users need support for errors in command usage and requests for help information.

One approach to addressing this problem has been to provide on-line tutorials or training manuals.[29] This approach is beneficial in allowing users to focus on their task activity and in providing specific reinforcement for tasks that are accomplished. Yet it is precisely this hand-holding mode of operation that often makes users unwilling to spend any length of time learning about a system on its own terms. When consulting on-line tutors, a user, in effect, ceases working. Consequently, there is a conflict between learning and working that encourages novice users to find ways of by-passing training in order to proceed with work, using trial-and-error methodologies.[10] Of equal importance is the fact that obtrusive tutorial systems cause expert users to become frustrated by the lack of freedom to accomplish tasks without being saddled with unnecessary details of system functions and explanations.

This paper presents a solution to these difficulties through the use of intelligent on-line user assistance that mitigates the learning-versus-working conflict by monitoring user activity to identify errors and provide advice for error correction that can be selectively viewed at the discretion of users. A natural-language mode of asking the system for help directly can also be of benefit. This user-assistance approach can help to better integrate the time and effort spent on learning with actual system usage.[30] This type of design can also help to counteract the sharp separation between learning and working that often reduces the motivation to use verbose training and help materials.

Commands can be erroneous for any number of reasons, and sometimes more than one form of error is present at a time. In an effort to identify categories of errors, we surveyed users for the most common and serious types of errors they make in using operating systems. Based on our findings, we constructed a taxonomy of error types. The taxonomy breaks possible errors into four categories:

- *Errors in execution* occur when the user knows the correct command to issue but does not carry it out correctly (e.g., a typographical error).
- *Errors in remembering* are situations in which the user has forgotten all or part of the syntax of a command.
- *Errors in understanding* occur when the user does not fully understand the meaning of a command and so uses it in an invalid way.
- *Errors in preparation* occur when commands are issued that look good superficially, but are invalid in the current environment (e.g., negative transference of commands across operating systems). This last situation includes errors that occur when the user has failed to make the necessary preparations so that a command will execute successfully.

Clearly, the first two kinds of errors do not really require an AI treatment. Typographical errors can be handled by a spelling checker (e.g., InterLisp DWIM[31]). Syntax help can be provided by improved on-line help or an input-completing parser. However, we feel that such components are generally not widely available as parts of operating systems. Thus it is necessary to offer assistance in these kinds of situations, along with the quite sophisticated help we are providing for understanding and preparation errors.

Our design also focuses on the subject of user queries to an operating system, which provides an appealing domain for the application of natural-language concepts. Given that the goal of our work is to develop an interactive environment that is both efficient and easily learned, a promising way to achieve this objective is to use natural language as an alternative to command input.

In designing a natural-language system, attention should be given to ways in which a user queries a system. There is rarely a direct correspondence between a precise statement or question representing a user's goal and a sequence of commands required to satisfy it. It is more likely that the user's query is vague, highlighting a poorly-defined goal, and it can be answered in multiple ways or by using a number of different sequences of commands. Thus there is some difficulty in validly mapping user queries to system answers. We have categorized user questions into several types in an attempt to reduce the complexity of the mapping problem. Based on observation of a sample of users ranging in skill level from novice to expert, we have created the following categories:

- *Procedural specification.* How do I perform a certain action?
- *Function specification.* What does a command do?
- *Goal or subgoal satisfaction.* How can a goal be accomplished? How can a specific subgoal be accomplished within the context of a higher-level goal?
- *Analysis of a process.* What is the easiest way to accomplish a goal?

To address the distinction between question types, we have constructed the following modified taxonomy of system responses as presented in Reference 13:

- *Introduce new information.* Present commands and actions that are new to a user.
- *Remind.* Present information about commands that the user may have forgotten.
- *Clarify alternatives.* Present information about the relationships (e.g., preconditions and postconditions) between commands to which the user has been exposed, and show alternative commands to achieve a task.
- *Elucidate goal hierarchies.* Present hierarchical information regarding the relationships between goals and subgoals.

These types of responses differ in their format and level of detail as well as in their emphasis and the amount of related information included. Clarifying and elucidating require a careful mixture of reminding and introducing new information. However, much of the knowledge needed to regard user plans in terms of current goals is incomplete. It is also not possible to predict with certainty what a user's goal might be. Hence, the responses must be provided as effectively as possible by system inferencing strategies within the constraints of incomplete knowledge.

In most systems the presentation format of help information does not parallel a user's view of the task.

From a system design perspective, emphasis must be placed on the inferencing used to attempt to identify a user's goal and the application of the appropriate knowledge to satisfy the context-dependent assistance provided to a user, on the basis of that goal. Similarly, the choice of the best presentation format for the information must be decided upon. Our solution to these problems is a goal-centered approach to user assistance, which we now describe.

Motivation for goal-centered user assistance

Rarely do on-line help systems marketed today take the context of a user's interaction with the computer into account when presenting canned help information.[32] Furthermore, in most systems the presentation format of help information does not parallel a user's view of the task. User assistance, in the form of suggestions and explanations of suggestions, should be presented in a format that coincides with a user's approach to accomplishing a goal. Similarly, help messages are often poorly understood by users. It is not sufficient to provide suggestions alone for accomplishing a goal. Rather, the system needs to make explicit its reasoning as to why the suggestions are offered and how suggestions can be implemented.

What a user needs to know about a system at any given time depends mostly upon the user's plans and goals. Even the most rudimentary advisor must take a user's goals into account, otherwise there is no guarantee that the advice given will be appropriate. Advice is appropriate only to the extent that it helps a user to derive and debug a plan of action for achieving his or her aims. The simplest way of responding to user queries is to anticipate the queries and store information to answer them as canned text. The simplest sort of canned explanations are error messages. However, providing predefined canned text as the basis of help information for all user queries fails to really satisfy learning needs in accomplishing a goal. Also, the system has no conceptual model of what it is saying. An advice-giving system needs to be able to reason about the current state of the interaction and give explanations at different levels of abstraction. This, in turn, implies the ability to present the necessary information in a scheme that supports the user's view of a plan for goal achievement. In addition to a meaningful presentation format, help messages need to be able to be explained at several levels of detail. Explanations must tell a user how to interpret suggestions given as options, as well as how to implement those suggestions.

REASON is an intelligent user-assistant prototype for a command-oriented system.

An objective of our work is to use one explanation paradigm to integrate dynamically generated explanations that parallel a user's view of a task with a precise explication of the system's reasoning for why a suggestion is offered. To achieve this objective, we propose a goal-centered approach to advice-giving. In this design, suggestions and justifications of the system's actions are given as a direct function of a user's needs within a current context and treated as discourse between a user and the computer. Suggestions are provided, based on an analysis of the match between the rules in the knowledge base and the user goals that the system sees as comprising a user model.

It is important to note that determining user goals is the focus of discourse-based systems that have been prototyped in the past.[12,33] However, our design can be differentiated by the assumption that the output of goal determination is taken as an input into an inferencing component and matched against knowledge to generate useful responses that are directly relevant to the context of the situation.

REASON design and implementation

Design considerations. REASON is an intelligent user-assistant prototype for a command-oriented system, such as the operating system OS/2™.[34] The purpose of REASON is to monitor user actions in order to (1) identify user errors and provide advice for correcting such errors, and (2) allow a user to ask for help directly through a natural-language front end. The operation of REASON is based on a user-centered systems design approach in which the identification of user goals or intentions is critical for accurate responses. Using this design, REASON operates in a mixed initiative mode, reacting to conditions in the environment as well as to explicit requests from users.

User interaction with an operating system provides an appealing domain for study and application of the AI techniques employed by REASON. Basically, all of the problems of language processing and reasoning (i.e., requirements for REASON to understand language, hypothesize user goals, access knowledge about goals, and form reasonable responses) are present in some fashion. The domain is complex enough to provide substantial subproblems, but not so unbounded that a useful working system must possess an extraordinary repertoire of knowledge.[12]

As discussed earlier, we see the major contribution of REASON to be not so much from a state-of-the-art solution to one or more very narrow fields of endeavor, but rather to be in the combination of advances from several fields over recent years into an effective, robust, working system. Consequently, REASON's performance in any one area is likely to be less than what might be found in any leading-edge laboratory, but we believe that the very provision of multiple features suitably integrated can lead to a working system that is able to solve real-world problems in real time. We, therefore, find ourselves frequently invoking an "80–20 rule" (or other variants

of the law of diminishing returns). If a component can provide a large percentage of the function of the state-of-the-art at a small percentage of the cost (memory, speed), we will happily settle for that.

Making the domain knowledge separate from the inferencing mechanism would allow easier porting to other domains.

Although we have not performed any quantitative measurements of those properties, we can usually tell when we are at the knee in the development-effort curve.

This philosophy was very important with respect to our goal of building a *working* system. We did not want to spend excessive effort in any one area, at the expense of the breadth of the system. Building REASON has caused us to get involved in the areas of problem solving, knowledge representation, natural-language understanding, planning, plan recognition, explanation, and text generation, each one of which is a substantial subfield of AI. We feel that limiting the effort in each area was the only way to build a complete system. However, we structured REASON in such a way that most components could be replaced by more advanced versions in the future.

We anticipate that the ways in which the REASON system will be used will generate new problems that can drive future research efforts in the areas of AI, intelligent user assistance, and interface design. For example, analysis of the structure of questions asked by users varying in skill level might lead to modifications in the interface supporting natural-language input, leading to a more formalized, constrained natural-language dialog as an alternative form of input. Overall, we are enthusiastic about the potential value of the design methods chosen to build REASON. We outline here some of the critical issues and decisions that have been guideposts in our design efforts.

Generalization of the REASON system. We have been developing REASON as an intelligent help system for OS/2. We do not believe that the concept of an intelligent help system is limited to OS/2 or to operating systems generally. Any environment in which the user issues commands via a command line should be amenable to incorporating the REASON technology—for example, text editors. Consequently, we have been seeking to develop REASON as far as is possible in a domain-independent manner.

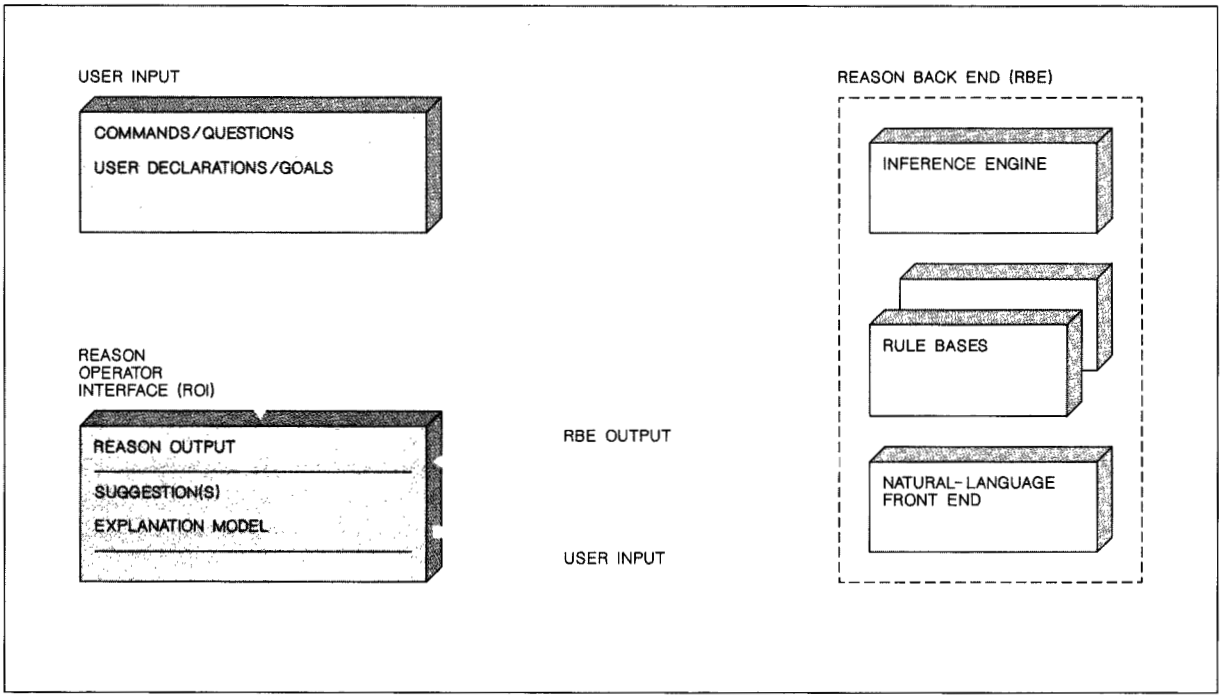
REASON as a rule-based system. REASON was conceived as a mechanism to solve two seemingly independent problems: (1) to respond to direct questions from the user about how to use the system, and (2) to intercede when it is noticed that the user is committing, or is about to commit, an error. Both situations require a problem-solving capability, although in the latter case an extra component is necessary to set up the problem to be solved. Both situations generally require the same body of knowledge about the subject domain. Thus, at the outset, it seemed to us that a single program could be built to achieve both of our major goals.

It seemed natural that this program would be a general-purpose problem solver combined with a knowledge base describing the domain in which it was to operate. Making the domain knowledge separate from the inferencing mechanism would allow easier porting to other domains, as well as easier debugging of the REASON system itself.

Inference mechanism

For an inferencing mechanism, we faced a choice of using backward-chaining or forward-chaining or a combination of both. The problems to be solved would generally be of the kind of determining a route (set of steps) to take the system from one state (the user's current state) to another state (the user's goal state). We felt that in addition to being largely goal-driven, there would be many cases for which we would want to find all solutions or at least a large subset of all solutions. Consequently, we decided on a depth-first, backward-chaining inferencing strategy much like the control strategy used by Prolog. We also saw that REASON would require a great deal of pattern-matching for which the Prolog unification mechanism would work well, which was another reason we decided to use Prolog. We built our own meta-interpreter in Prolog, because, although we wanted to take advantage of Prolog's search strategy as a control mechanism, we wanted some control

Figure 1 Model of REASON user-assistant design



over the search strategy, as well as the pattern-matching, ourselves.

Natural language

We wanted a natural-language front end to REASON, but we did not want to undergo an extensive research project in this area. As previously mentioned, we performed some preliminary studies with potential users to determine the kinds of questions they would ask an intelligent help system. While the totality of questions was spread over a large range—from short to long, simple to complex—the majority were of the "How do I . . ." or "What does . . . do" variety. Based on the 80–20 rule, we judged that a fairly simple grammar could be constructed to represent the majority of these questions.[35] We correspondingly decided to use a Definite Clause Grammar (DCG),[36] because it was easy to write a DCG that parsed most of these questions. This decision fitted in very well with the choice of programming language, because DCGs are very easily implemented in Prolog.

The components of REASON

We now present an overview of the conceptual model upon which REASON is based. This model, showing the functional components of REASON, is presented in Figure 1. It is seen that REASON consists of two major components: (1) the REASON operator interface (ROI, which we pronounce "roy"), and (2) the REASON back end (RBE, which we pronounce "ruby"). These components handle the user-interaction and inferencing processes, respectively, and as such require a relatively low bandwidth for communication between one another. The logical separation of the user interface from the inference engine, as depicted in Figure 1, is maintained in the physical implementation. ROI and RBE can be implemented either as separate processes in a multitasking operating system, such as OS/2, or on different physical machines on a local area network (LAN).

REASON operator interface. ROI handles REASON's communications with the user. This largely entails capturing the user's input and transmitting it to RBE and capturing RBE's response and displaying that to the user. In its present form, REASON requires the

existence of an operating system shell, because it must gain access to the commands that the user types in and to the return codes issued by the operating system. Under OS/2 Standard Edition 1.1, ROI is implemented as a Presentation Manager application.[34] It should be noted that REASON itself currently monitors only the user's input in the system's command line. The user, however, can view REASON's response (suggestions and explanations) by direct manipulation. Later in this paper there are several examples of ROI in execution, presenting windows of text containing REASON's suggestions and explanations.

REASON back end is the component of REASON that actually solves the user's problems.

In designing REASON, a primary aim has been to minimize the intrusiveness of the system on the user. Usually the user does not want to surrender control of the interaction with the computer and does not work efficiently and effectively with an intelligent agent that is continually interrupting to give advice. In an effort to address these concerns, REASON sits as a guardian monitoring the user's interaction, thereby allowing it to play as unobtrusive a role as possible. If an error is made or question is asked in a command-line environment, the user is given a visual cue (e.g., an icon appears), indicating that REASON is ready to give a response that can be viewed if so desired. In communicating this advice to the user, ROI does not take control over the user's interaction with the computer. Rather, it allows the user to have the option of viewing suggestions and explanations at the desired level of depth, depending upon the expertise of the user. REASON neither automatically corrects the command for the user on the command line nor executes it. Once the suggestions are reviewed, the user has control over their execution.

REASON back end. REASON back end (RBE) is the component of REASON that actually solves the user's problems. It does this by an inferencing process and as such can be thought of as being the AI component of REASON. RBE has access to a description of the user's current environment, and ROI passes the commands and questions that the user issues to RBE. RBE responds to all questions and to those commands that are in error. Its response is usually in the form of one or more suggestions, each suggestion being accompanied by an explanation. A suggestion may be anything from *do nothing* to a series of several actions to be carried out. These responses are passed back to ROI for display to the user.

RBE consists of a compiled Arity/Prolog® application, as well as an external file containing its rule/frame bases. Most of the domain-dependent knowledge for an implementation of REASON is located in this file, but those functions that are inextricably linked with the syntax of the commands in the domain are part of the compiled module. This means that changing domains generally entails recompiling the REASON system. This would be a drawback if REASON were thought of as a general-purpose, expert system shell like ESE.[37] For the reason that most users are not going to be porting REASON between domains, this lack of flexibility is not regarded as serious. Work is being done to extend the number of built-in functions that can be used to describe a domain, so as to minimize the domain-dependent code.

RBE functional components

The main components of RBE are the parser, the inference engine, and the knowledge bases. This section contains descriptions of these components. However, we introduce the subject with a discussion of the methods of knowledge representation used in REASON.

Goal expressions. The domain knowledge is represented in the form of *frames*, within which are *locations* or *slots* where information is necessarily represented with a finer granularity. The format used here is what we call *goal expression* and is used throughout REASON. For example, the output of the natural-language parser is a case frame, which is transformed into a goal expression.

A goal expression consists of a *predicate name*—representing an action—and arguments—representing the objects and attributes involved in the action. The objects are represented (depending on the level of detail required) either by atoms or by five-part lists that we call *object descriptors*. The components

of object descriptors are: (1) the object's generic name; (2) its given name or label; (3) its adjectival descriptor; (4) an all/one/none flag; and (5) possible containing object. The last field is used to talk about

A special variant of the goal expression is known as the condition.

files in directories as well as other instances of containment. For example, the goal expression representing *erase all files abc.* in directory def* would be

```
erase([file, abc.*, _, all,
       [directory, def, _, one, _]])
```

It can be seen that goal expressions are syntactically Prolog predicates, although they are never evaluated directly by the base Prolog system. We use the square brackets for lists, and the Prolog notation for identifiers. In this notation, identifiers beginning with a lowercase letter are atoms, those beginning with a capital letter or the underline character are variables. The underline character used by itself is the *anonymous variable.* Variables are initially unbound, that is, not matched to any atom or structure. When bound, all variables within a goal expression or one of our frames (described later in this paper) are bound to the same value. Each instance of an anonymous variable can *bind to* (or *match*) any value. Being unnamed, however, it can not be referred to. Therefore, this essentially represents a "don't care" condition.

Goal expressions may be mapped to a subset of Sowa's[38] Conceptual Graphs representation. We have considered using Conceptual Graphs directly, but have so far found the reduced expressive power of goal expressions to be sufficient to meet our needs.

Conditions. A special variant of the goal expression is known as the condition, which is a predicate with name *condition.* Condition represents a system state by means of an assertion that an attribute of a certain aspect of the system has a certain value. Specifically, the first argument is the property name, the second its value, the rest determine the aspect of the system being described. For example, consider the following condition:

```
condition(exists_file, t, c:,
          \dir1\dir2, myfile.txt)
```

This condition asserts that the file *c:\dir1\dir2 \myfile.txt* exists (where t stands for true). These conditions are not asserted into the Prolog clause space but are processed by our own meta-interpreter. Mentioning the truth value directly allows us to assert:

```
condition(exists_file, f, c:,
          \dir1\dir2, myfile.txt)
```

which would be difficult to do if we relied simply on the Prolog negation-by-failure. In addition, these conditions can take values other than true and false. Consider the following example:

```
condition(screen_mode, mono)
```

This condition says that the current screen mode is MONO, where other values include CO40 and CO80.

We have defined conditions so that if the name and the third argument and beyond of two conditions are equal, the second arguments (or values of the conditions) are equal. This effectively makes our conditions into functions. This has great value because it helps us determine what prior knowledge is invalidated when new knowledge is gained. For example, if we have the following condition,

```
condition(exists_directory, t, dir1)
```

then the assertion of the following condition

```
condition(exists_directory, t, dir2)
```

will not affect it, because directories can coexist. However, consider the following condition:

```
condition(exists_directory, f, dir1)
```

This condition causes the former assertion to be cancelled, because a directory cannot simultaneously exist and not exist. It should be noted that the values t and f are not special to REASON. If they were uniformly substituted by *red* and *blue*, say, the program would work the same way.

Figure 2 Command frame for the *erase* command

```
command_frame(
      command :: [erase, erase, [_Arg],
                  [drive::_Drive, path::_Path, filename::_Name]],
      parse_routine :: parse_erase,
      intent :: [erase([file,_Arg,_,_,_]),
                 increase(slots_in_directory,_Drive,_Path)],
      must :: [condition(exists_file,t,_Drive,_Path,_Name)],
      pre  :: [],
      post :: [condition(exists_file,f,_Drive,_Path,_Name)],
      tells :: [],
      prop :: [wild_card :: y, int_ext :: int]).
```

Knowledge bases. REASON maintains its domain-specific information in four different collections or *knowledge bases*. These knowledge bases resemble *rule bases*, even though not all of the information is strictly in the *antecedent-consequent* form that is typical of an expert system's rule bases. The knowledge bases represent knowledge of the commands and actions available to the user, certain relationships between goals and states, and ready-made solutions to anticipated goals. This information is in the form of frames, which—in combination with REASON's *generic rules* described later—form actual rules. These frames employ goal expressions heavily.

Command frames. The REASON command knowledge base consists of objects called *command frames* that represent the commands the user can issue in the current operating domain. Command frames have the following slots:

- A list containing the command name, its class, and two different representations of the argument list
- The *parse routine*, which is the name of the routine used to parse the arguments
- The *intents* of the command, namely, the possible user goals that this command accomplishes, represented as a list of goal expressions
- The *preconditions* of the command, namely, those conditions that must be true before the command can be executed: Two slots, denoted by the labels *must* and *pre*, contain these conditions. These slots contain lists of disjunctions of conditions. The difference between the *must* and *pre* slots is that if a command is tried whose *pre* conditions fail, REASON tries to create a state in which they succeed, whereas if the *must* conditions fail, REASON does not pursue this line of reasoning.
- The *postconditions* of the command, namely, those conditions that become newly true after the command is executed: This slot is a list of conditions.
- The *tells* of the command, namely, the information that running the command provides: This slot is a list of goal expressions.
- A property list for miscellaneous information, such as whether the command takes wild cards or is part of the operating system kernel.

For example, the command frame for the **erase** command in shown in Figure 2.

As described later, command frames such as this are used in conjunction with the REASON *generic rules* to determine corrections of the user's input. One of these generic rules makes an incorrect command valid by constructing one or more commands to be issued in advance to create a state in which the former command's preconditions hold. This can give rise to undesirable consequences if the allocation of preconditions to the *must* and *pre* slots are not carefully considered. In the case of the **erase** command, putting the condition that the named file must exist as a *must* rather than a *pre* condition prevents REASON from suggesting that the user **create** a file and then erase it, when trying to erase a nonexistent file.

Action frames. The REASON action knowledge base consists of objects called *action frames*, which represent the noncommand actions that the user can issue. Typical actions might be those of inserting or removing diskettes on pressing certain key combinations, etc. Actions are syntactically very similar to commands, except that they have no associated parse routines or property lists.

Figure 3 The action frame for inserting a diskette

```
action_frame(
      action :: ['insert diskette', [_Drive]],
      intent :: [insert([disk,_,floppy,_,_,_]),
                 insert([diskette,_,_,_,_])],
      must   :: [condition(exists,t,drive,_Drive)],
      pre    :: [condition(occupied,f,_Drive)],
      post   :: [condition(occupied,t,_Drive)]).
```

Figure 4 A consequence frame

```
conseq_frame(
       goal_or_state :: edit([file,_Name,_Adj,_,_Desc]),
       consequence   ::   [change([file,_Name,_Adj,_,_Desc]),
                           view([file,_Name,_Adj,_,_Desc]),
                           create([file,_Name,_Adj,_,_Desc])]).
```

For example, the action frame for inserting a diskette might look as shown in Figure 3. Note the two intents are used to cover the possibility that the user might refer to the object being inserted as either a diskette or floppy disk.

Consequence frames. Consequence frames are essentially if-then rules used to interrelate goals and states. The primary use of these objects is in cases where what the user wants to do is a subset or side-effect of a more major operation. For example, viewing a file can be accomplished by editing it, but that is not the primary purpose of the editor, so view is not among the intents of an editor, though edit is. A consequence frame (shown in Figure 4) is used to state that viewing is a possible consequence of editing.

Goal frames. Goal frames are used to tell REASON the algorithm for breaking certain goals into subgoals. The purpose is to relieve REASON of having to work out from first principles well-known techniques. A goal frame has the following components:

- The goal
- What to do if some or any of the parameters are unbound (i.e., the default slot)
- Conditions or other relations to test (i.e., the process slot)
- The subgoals which, if achieved, guarantee the goal will be satisfied

For example, if the system has been given the goal of formatting a diskette, it must determine the drive, if unspecified, and then have the user insert the diskette before running the format command. This is achieved by means of the goal frame shown in Figure 5.

Currently the frames must be coded by hand, but an aid to automate the process is being built. All the frames are maintained in an external file that REASON reads in at run time. Consequently, users are free to customize the file as they see fit. For example, they can add command frames to describe new applications they may import or goal frames to describe new procedures.

Natural-language parser

The REASON parser is based on a modified Definite Clause Grammar,[39] with built-in spelling corrector and semantic role analyzer. The analysis is case-based,[40] thereby producing a case frame that is later converted to a goal expression. The lexicon (vocabulary) is maintained in an external file, along with the semantic/syntactic role relations, thus allowing

Figure 5 Goal frame

```
goal_frame(
        goal :: format([diskette,_Drive,_,_,_]),
        default :: [ask_user_then_default(_Drive,'What drive?',a)],
        process :: [condition(exists,t,diskette,_Drive)],
        subgoal :: [insert([diskette,_,_,_,_])],
                    format([disk,_Drive,_,_,_])]).
```

Figure 6 Case frame from the question "How can I send a file to the printer?"

```
quest_frame(wh(nocase,manner,location_or_manner,[how]),
            case_frame(send,
                       subj(actor,agent,[I]),
                       obj(concrete_thing,patient,[a,file]),
                       pp(goall,[to,the,printer]),
                       vmod(location_or_manner,[how])))
```

them to be updated at run time without recompilation. At one time, the parser contained a built-in morphological parser; however, we found that because almost all questions issued by users in our studies were in the present tense, it was more efficient to include plural forms of nouns and third-person, singular forms of verbs in the lexicon.

The verbs in the lexicon are tagged with one or more different labels that are used both in the selection of the grammar rule and in the construction of the case frame. For example, the set of labels used includes **vi** (intransitive verb, such as *quit*) and **v_recipient_np** (verb taking a recipient, then a noun phrase, such as *send*).

As an example, the case frame that would be generated from the question "How can I send a file to the printer" is shown in Figure 6.

The components of a case frame are the verb and its various cases. For example, the subject (*subj*) of the verb send is in this case the word *I*, which is an example of an actor and plays the role of agent. For this question and others like it, the parser generates two nested case frames; the outer one is called a *question frame.*

When the parser succeeds in generating a case frame, it invokes a function we call the *case-frame filter* to produce a goal expression and an associated flag called the *question type.* For the previous example, the generated goal expression is:

```
send([file,_,_,_,_],
     [printer,_,_,_,_]).
```

with the associated question type of *how.*

Inference mechanism. REASON employs a depth-first, backward-chaining inference mechanism to solve problems. This mechanism employs a set of what we call *generic rules,* processed by a second-level interpreter. These rules specify different ways that REASON solves the current problem as follows: a given rule may completely solve the problem; it may solve part of it and generate one or more subgoals for the rest; or it may simply redefine the problem. Solving the problem is defined to mean either taking an incorrect command and converting it to a sequence of commands that do what the incorrect command is guessed to be attempting to do, or taking a goal and generating a sequence of commands which will achieve the goal. This sequence of commands becomes what is presented to the user as a *suggestion.*

A suggestion often consists merely of a single command, and occasionally it is empty if REASON determines the user is already in the desired goal state. Sometimes the suggestion contains actions, such as inserting a diskette, but for brevity in this section we assume all of the components of a suggestion are commands.

Overall strategy. Input to REASON back end (RBE) is either an incorrect command issued by the user or a question or command in English. RBE tests the input according to the following criteria in turn. If it meets success in any one test it deals with the input appropriately. This might result in multiple solutions, but RBE does not proceed with any further tests in the list. The input is tested for being:

- A correct command
- A command that is correct so far but incomplete
- An English question or command
- An incorrect command
- A goal expression
- An English question or command with spelling errors

If the input is an incorrect command, RBE tries to find possible ways to correct the input before terminating. This often results in several suggestions.

RBE accepts goal-expression inputs directly, primarily as a debugging tool. They are processed by the same mechanism that handles incorrect commands. The value of exposing this interface to the user must be evaluated in future testing of the REASON system.

When REASON seeks to interpret the input as an English question, it passes the input to our natural-language parser. If a parse is generated, the resulting case frame is processed by a *case-frame filter*, which produces a goal expression along with a flag indicating what kind of question was asked (i.e., what, how, etc.). This goal expression is then processed by RBE.

If the input is parsed as English with spelling errors, a goal expression representing the input (correctly spelled) is generated, along with a description of the corrections made. These corrections are viewable, along with other information about the parse process under the REASON operator interface (ROI).

Generic rules. Generic rules may be thought of as transformations that take a command or a goal and produce a set of subgoals. Secondary outputs of the application of a rule are commands to be recommended to the user, along with an explanation fragment and a set of rules to be tried further. The process comes to an end when a rule generates no subgoals. The totality of commands is collected and presented as a suggestion, and the explanation fragments are collected and converted into coherent English text.

The rules that pertain to the syntax of commands are not usually domain independent.

The REASON generic rules include the following:

- Correct the spelling of the command
- Correct the spelling of the arguments
- Correct the argument
- Complete the command
- Change the command to one with similar meaning
- Select a command to satisfy the given command's preconditions
- Select a command whose intention matches the given goal
- Transform the given goal into a more general one
- Break a goal into subgoals

Domain dependence of generic rules. The rules that pertain to the syntax of commands are not usually domain independent. They do in fact represent the bulk of the domain-dependent component of REASON that is not yet able to be extracted into an external file. Consequently, on changing domains, this part of REASON is recompiled.

To REASON, actions have a designated, predefined syntax. If the new domain consists entirely of actions, REASON is not recompiled, because command-oriented generic rules are not applicable. To demonstrate this, we implemented a "monkey-and-bananas" problem entirely as action frames and goal frames. The monkey-and-bananas problem places a monkey and a crate at opposite corners of a room. From the center of the ceiling hang some bananas.

To be successful, the monkey must move the crate under the bananas and then climb on the crate to reach the bananas. This problem is often used as a benchmark for comparing the performance of inference engines. We have used it to demonstrate that REASON is capable of solving such problems. We were able to merge the new knowledge bases with those we had to describe the operating system, and REASON could solve problems from either domain without being recompiled. Monkey-and-banana problems were posed to REASON by means of the goal-expression input mechanism, described earlier.

RBE operation

REASON solves problems by the repeated application of the process described in this section and depicted in Figure 7. This figure shows how one of the generic rules named in the input is selected, merged with a frame, and applied to the current problem to produce a new problem and a new set of rules to try, along with the suggestions and explanations produced to this point. The process takes as primary input what we call the *current problem* and a set of *rule names.* The current problem may be a command to be issued, a condition to be satisfied, or a goal to be solved. The rule names list those rules that either the starting conditions or the previous application of this procedure have determined are appropriate for the current problem. The secondary input to the process is the generic rule base, which is the collection of frames for the current domain and a description of the current environment.

The problem solver takes each rule named in its input, in turn, and tries to use it to solve the current problem. It does this by selecting an appropriate frame with which it *unifies* to form a specific rule. The problem solver then applies this specific rule to the current problem. If the application succeeds, it generates the following: a new problem, a new set of rule names (dictated by the rule that succeeded), a suggestion fragment, and an explanation fragment. The process terminates when no new problem is generated. In this case, the chain of suggestion and explanation fragments is processed and sent to ROI to be presented to the user.

More than one suggestion can be produced if either of the following situations occurs during the inference process:

- More than one rule name is listed in the input.
- A given rule unifies with more than one frame.

Whenever a sequence of rules ends, the suggestion and explanation is generated and the inference engine backtracks to the prior choice point, in order to try the next alternative.

In such domains as medical diagnosis, for which expert systems have been built, it is often advantageous to be able to deal in varying degrees of abstraction, particularly when it comes to giving explanations.[41] Depending on the nature of the domain and the domain expert's knowledge of it, abstract knowledge may or may not be available to the expert system. That is, it may not be possible to enumerate all the possible domain principles. Here, by contrast, because of the artificial and generally systematic nature of the problem domain, we can in a relatively simple way start with abstractions (our generic rules) and get to specifics in a straightforward manner.

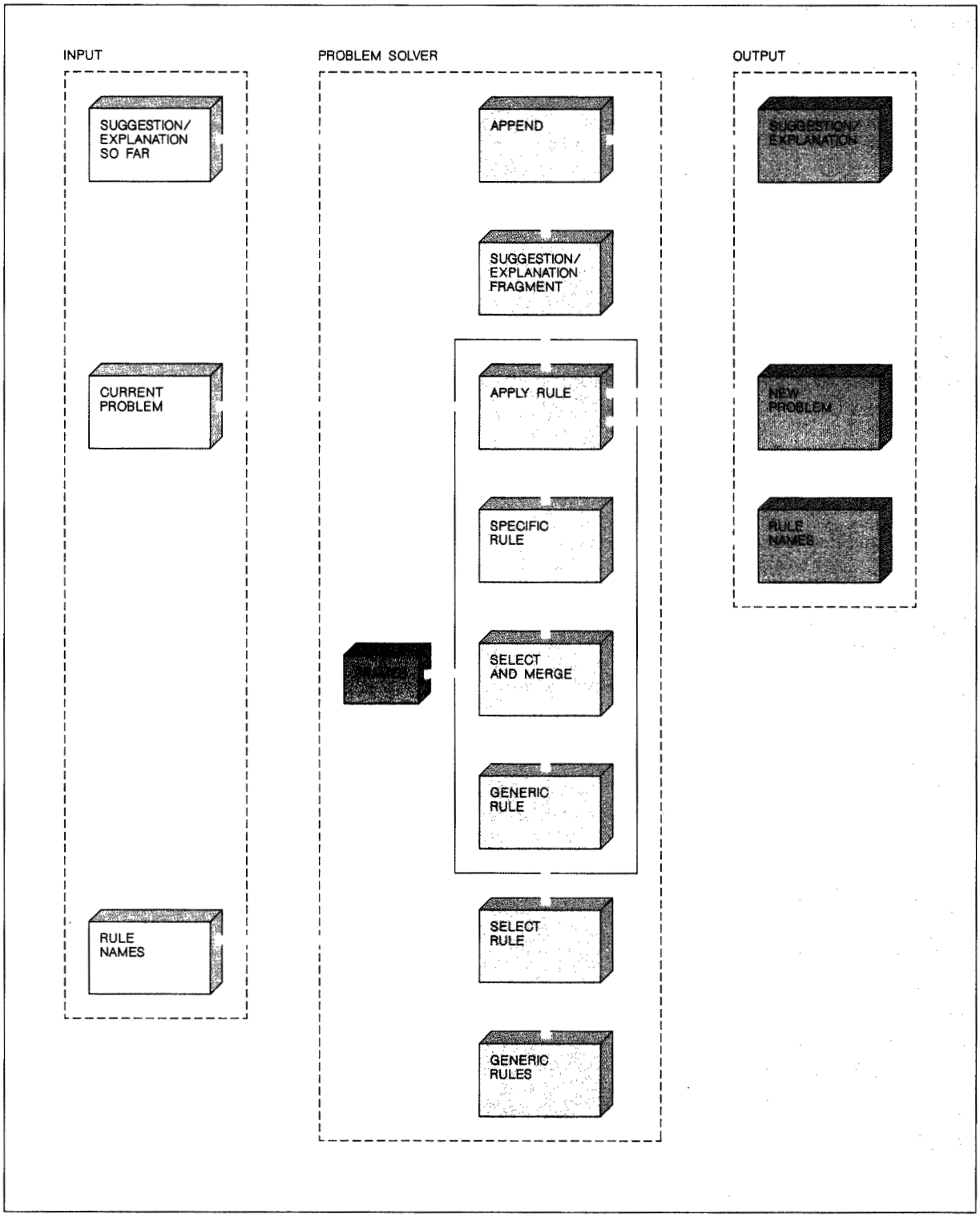
Unification of the rule is tried with all command frames in turn.

Example. This problem-solving process can be illustrated by means of an example. In this example, we see how one of REASON's responses to a user's failed attempt to issue a **cd mydir** command might be to issue a **md mydir** command first.

Suppose the user issues the command **cd mydir** (change to directory **mydir**), where directory **mydir** does not exist. The input to the process (the current problem) is the command **cd mydir**, along with a set of rule names that includes the rule **comm_prem** (command premature). This rule states, in effect, that when you wish to issue a command **C1** and cannot do so because its preconditions **P** are not met, find a command **C2** whose postconditions include **P**, solve for **C2**, then issue command **C1**. This rule is generic because **C1**, **C2**, and **P** are unbound.

Unification of the rule is tried with all command frames in turn. At some point the **md** command

Figure 7 REASON problem solver



(make directory) is attempted. In this case, the unified rule becomes the following: when you wish to issue a command **C1**, and cannot because its precondition—i.e., *directory D exists*—is not met, solve for command **md D**, then issue **C1**. This rule applies to the input **cd mydir**, in so doing binding **D** to **mydir**. The new problem to be solved becomes the viability of the command **md mydir**.

When the unified rule is applied to the input, it is in effect constructing the highly specific rule: when you wish to issue a command **cd mydir**, and cannot because its precondition—*directory mydir exists*—is not met, solve for command **md mydir**, then issue **cd mydir**. It is this rule, in a transformed state, that is used to construct the explanation of this step.

It should be noted that this fully specified rule represents a small chunk of specific domain knowledge and corresponds to the level of granularity of a rule in a typical expert system. It is by the regular nature of the artificial environment in which REASON operates that we are spared having to deduce all of the thousands (potentially) of similar rules. Instead, we deal with a few dozen command frames and about a dozen generic rules.

The application of the **comm_prem** rule was just described. It was mentioned that this was one of several rule names passed as input to the problem-solving process. All named rules are tried, and any others that succeed have generated alternative suggestions. For example, if there had existed a directory **mydir1**, say, then the **args_misspelled** rule would have succeeded.

Explanation paradigm paralleling user task activity

REASON offers suggestions when the user issues an incorrect command, as well as when the user asks the system directly for help using natural-language queries. Explanations of suggestions are necessary to aid the user in understanding the system's reasoning strategies for why commands are erroneous and how to recover from the errors.

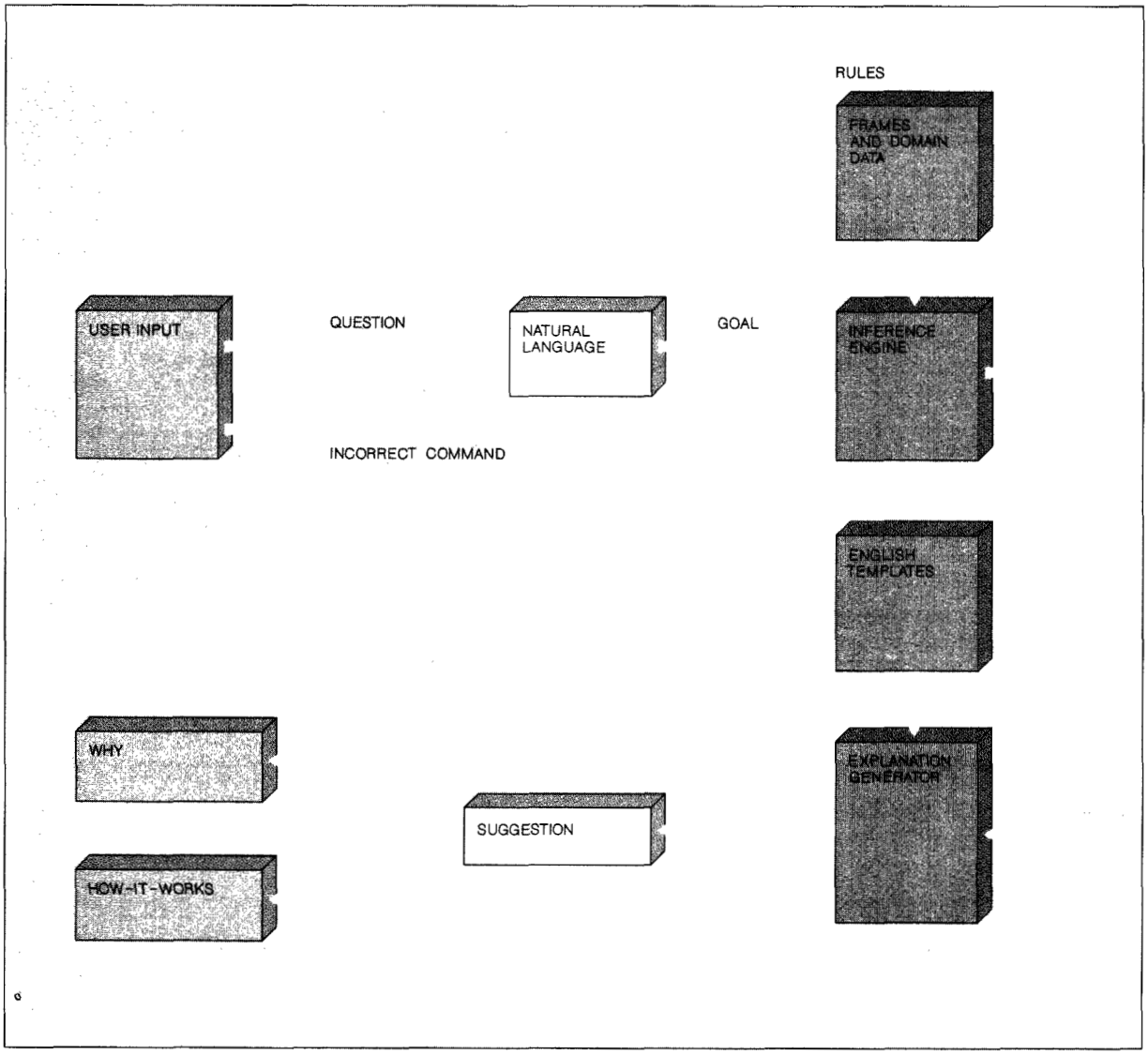
As noted earlier, historically, most explanation facilities for help systems have been merely static traces of system rules or canned text that resembles an online manual. A primary problem with using static traces as explanations is that the system can state what it does or did to arrive at a suggestion, but it cannot state why the system recommends a particular suggestion. These types of explanation do not take into account the context in which the error was made or that in which the question was asked. Also, the presentation format of help information usually does not parallel the user's view of the task. Suggestions and explanations must be presented in a format that coincides with the user's approach to accomplishing a goal. It might be mentioned that help messages are often poorly understood. It is not sufficient to provide a single suggestion for accomplishing a goal, but rather the system needs to make explicit its reasoning as to why the suggestion is offered and how a suggestion can be implemented.

In studying new user interface technology, intelligent help appears to be a way of providing context-dependent advice that can operate as a partner with the user by offering advice based on user intentions or goals. What the user needs to know about a system at any given time depends mostly upon those plans and goals. Providing predefined canned text as the basis of help information for all user queries fails to satisfy such needs in accomplishing a goal. An advice-giving program must be able to reason about the current state of the interaction.[9] This in turn implies the ability to present the necessary information in a scheme that supports the user's view of the plan for goal achievement. In other words, the explanations should be tailored to the specifics of the situation at hand. In addition to a meaningful presentation format, advice must be explained at several levels of detail. Explanations must tell the user how to interpret suggestions given as options, as well as how to implement these suggestions.

To accomplish these objectives, REASON implicitly interprets the user's statements and then adapts the advice accordingly. We have chosen to implement one explanation paradigm that can be used to integrate dynamically generated explanations that parallel the user's view of a task providing information on why a particular set of suggestions are recommended, as well as how to perform the steps required to achieve a goal.

The basic idea is to establish an explanation paradigm, based on stated or inferred goals of the user. Its components are generated dynamically through system inferencing and are presented using two complementary formats that are derived from a solution-tree trace converted into connected English sentences. This explanation paradigm parallels the hierarchical nature of the user's knowledge about a task, when attempting to achieve a goal using the

Figure 8 Explanation process



computer system. The explanation process is depicted in Figure 8. Suggestions and two kinds of explanations are generated when the user asks a question or types an incorrect command.

Before generating an explanation for the user, the system must first determine what the user wishes to accomplish and how it should be done. The aim of the system is to take the user's problem as expressed in terms of either an incorrect command or a natural-language question and produce one or more suggestions. Each suggestion consists of one or more steps for the user to take to achieve the inferred goal. In the event that the system finds that the user's desired goal state already exists, the suggestion will be to do nothing. In any case, the solution to the problem must take place in such a way as to be amenable to explanation.

All user queries to the system are represented internally as goal expressions, whether originating in incorrect commands or natural-language questions. All

parts of an explanation are generated dynamically on-the-fly by passing these goal expressions to the problem solver and applying the appropriate inference rules. In generating explanations, for each rule sequence that is applied, a trace is kept of the essential details of each rule, such as name and appropriate parameters. By applying suitable text templates to this solution tree, an English-like explanation of the inferencing process is generated.

The system offers one or more suggestions. The explanation paradigm includes two types of explanation of the steps for these suggestions. The first focuses on how the recommended steps fit together to solve the subgoals of the problem (i.e., how-it-works). The second type presents the reason why a particular suggestion is offered by the system, thereby providing a logical connection between each step of the suggestion and the original problem or question. Both types of explanation are used to support a logical mapping from the suggestions offered, to the achievement of user goals. Figure 9 shows a response to the question, *"How do I install and run program* TEST *from drive A to directory* MYDIR *on drive C?"*

As shown in Figure 10, the basic idea of the *how-it-works* explanation is to show the procedures or steps that can be implemented to satisfy a set of subgoals leading to an overall goal. This explanation shows how multiple steps in a suggestion fit together to solve all or part of the original problem. The how-it-works explanation is derived from the solution tree in a hierarchical top-down manner. Each high-level goal is successively broken down into lower-level subgoals and finally into leaf nodes representing system commands or actions that the user must issue in order to achieve his or her overall goal. How-it-works is available when the solution process involves matching users' goals or subgoals against a predefined goal hierarchy.

The explanations for the suggestions to a problem or question are intended to provide information on errors and why alternative suggestions are recommended. A *why* explanation is always provided for each step in a single or multistep suggestion. The *why* explanation is derived from the solution tree using a bottom-up trace from each single leaf node in the hierarchical tree structure to the highest goal (e.g., *Install and run program* TEST). The sequence of rule-firings in the trace is converted into a sequence of English sentences that read fluently and explain the logic of the suggestion. If the user asks for an explanation of why md mydir is given as part

Figure 9 Example of a REASON suggestion to a natural-language question

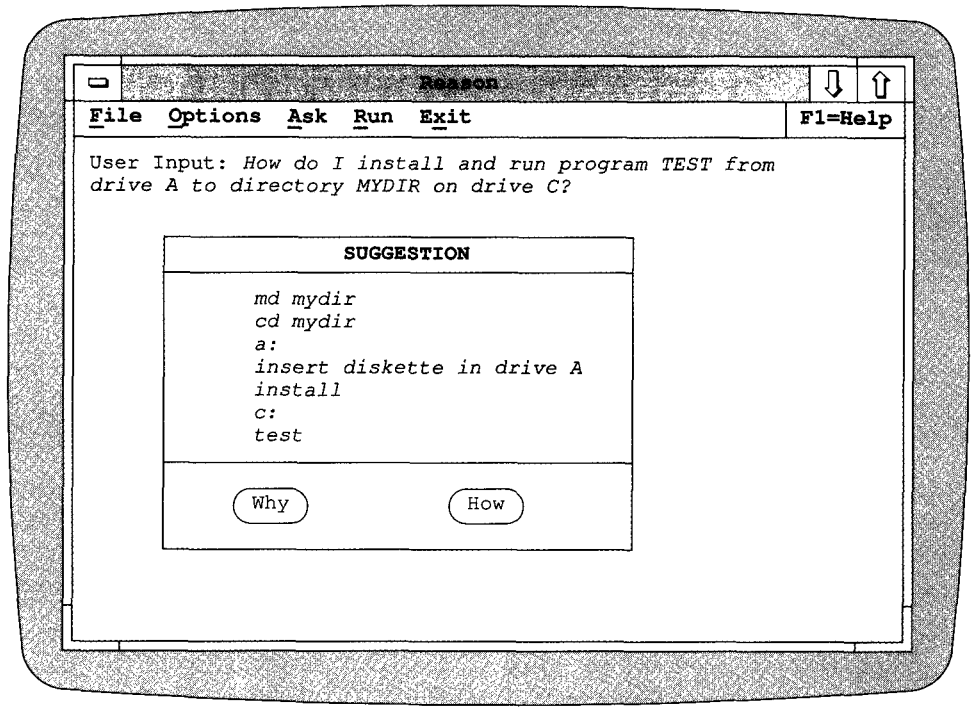
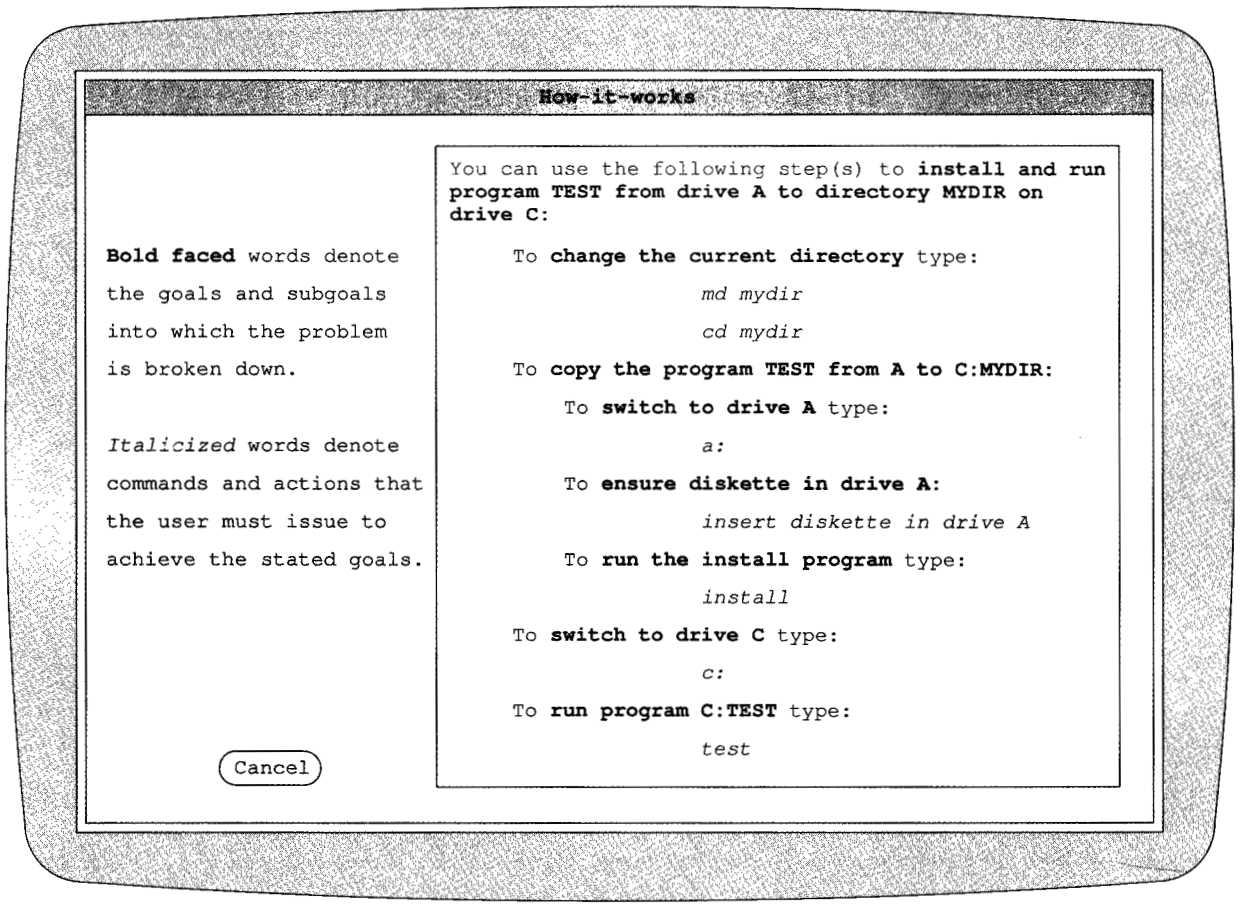


Figure 10 Example HOW-IT-WORKS explanation



of the suggestion to install and run program TEST, the help shown in Figure 11 is offered.

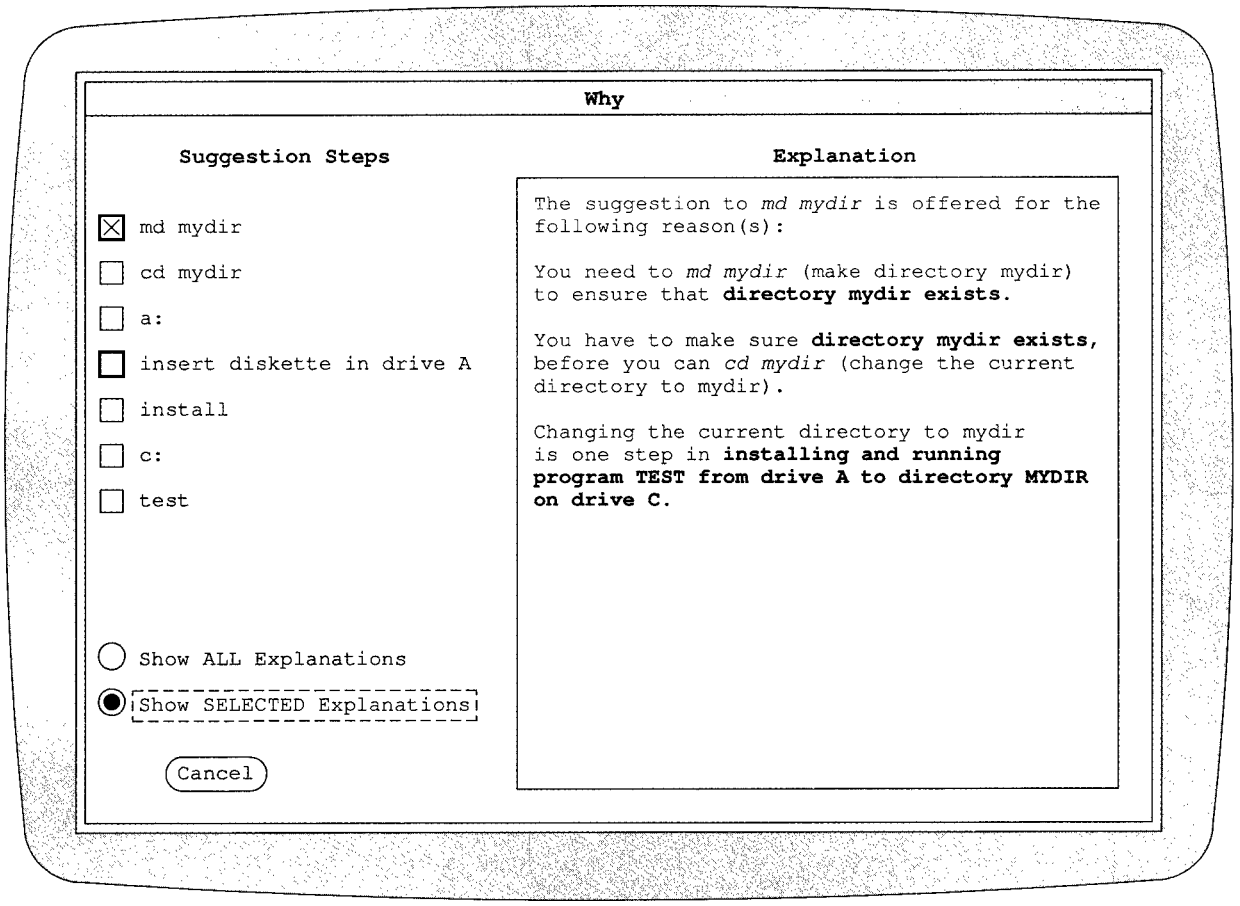
This explanation paradigm addresses the stated problem by dynamically generating explanations that take the context of the user-computer interaction into account. In addition, the explanation content, which consists of a goal-based rationale, parallels the user's view of a task, and complementary explanation types (i.e., why and how-it-works) are used to enhance the user's understanding of the information presented.[30]

Summary and conclusions

This paper has concentrated on our attempt to design and implement a prototype for a commercially feasible advisory system that is based on the integration of AI-based advances from several fields. It describes our extensions of previous research on designing advisory systems. Our aim is to enhance the helpfulness of computers through the use of an intelligent command line designed to support a mixed-initiative mode of interaction for correcting command errors and responding to natural-language questions. On the basis of our experience in this endeavor, it appears that leverage can be obtained by providing the following:

- The integration of several AI techniques to build a robust, commercially feasible intelligent advisory system
- The need for context-sensitive advice, based on a goal-centered approach to user assistance
- The development of taxonomies to enhance an understanding of user errors and typical requests

Figure 11 Example WHY explanation



for help, which can serve as a means to ensure that the advice provided is compatible with the user's requirements
- The right balance of function to effort required for implementation
- Development in a domain-independent manner, with the aim of porting the technology across operating system environments as well as task domains

These considerations have led us to concentrate on the following features of the REASON system:

- The definition of a goal language as a knowledge-representation medium for user goals and intermediate subgoals
- A mixed-initiative input mode to accommodate varying levels of user expertise
- Unobtrusive interaction with the user, allowing for the selective viewing of suggestions and explanations
- The development and presentation of an explanation model to integrate dynamically generated suggestion explanations that parallel the user's view of a task
- An emphasis on making the system user extendable with respect to the various rule bases

Although our experience with the general problem-solving capability of REASON is limited, we are optimistic about the prospects for porting the rule bases and inference mechanism beyond the operating system domain and into other domains. In its current form, REASON is not ready for general use. Nevertheless, it has demonstrated a capability for inferring user goals, processing natural-language queries, and

formulating a set of suggestions to achieve desired goal states in a robust manner.

Future directions

At present, we are working on several fronts toward giving REASON the ability to build a more refined model of the user. Our main objective is to deduce the user's plan by observing command and question patterns, which is the task of plan recognition. We also intend to use traces of user errors and the types of help previously sought as criteria for inferring user expertise. It is our hope that by using individualized user models, based on expertise and prior history, the system will be able to reason more deeply about the user's actions and plans. This information will allow REASON both to deduce plans that might otherwise be missed and to determine which of many equally plausible plans is most appropriate in a given context.

Currently, the REASON knowledge base must be coded by hand. We are building an interactive development aid to ease the developer's task considerably. Not only will it cut down on the possibility of syntax errors in the knowledge base, but also it will be able to show the developer all possible matches of the rules and frames being entered. This will help ensure that the rules and frames are used as the developer expects. In addition, this extension will enable partial precompilation of the knowledge base, which will improve REASON's performance.

An empirical investigation of REASON with users in varying working environments is also being planned. We are quite eager to evaluate REASON's inferencing power across a range of users with differing needs for advice from the system. Included within this evaluation is an assessment of the validity and effectiveness of the explanation model incorporated in the REASON design. This usability testing will also provide us with valuable information as to the robustness of our natural-language capabilities, thereby helping us to determine the feasibility of the language restrictions built into the REASON natural-language parser. Such an evaluation will indicate whether the natural-language front end is complete enough to be used as intended.

In the longer term, we plan to address whatever deficiencies are found by experimental use. We also plan to extend our natural-language parser to be able to differentiate between hypothetical questions and questions referring to the current state of the system. Similarly, we plan to implement multiple explanation modes characterized by different depths of help and tutorial information that is present in on-line documentation, possibly stored as hypertext/hypermedia.[42] Finally, we would like to do experimental research to find a way to incorporate REASON into noncommand-driven systems, such as those with direct manipulation interfaces, with the hope of continuing to broaden the scope of potential applications of intelligent help.

Cited references

1. J. L. Alty and M. J. Coombs, "Communicating with University Computer Users: A Case Study," in *Computing Skills and the User Interface*, M. J. Coombs and J. L. Alty, Editors, Academic Press, London (1981), pp. 7–71.
2. R. Burton and J. S. Brown, "An Investigation of Computer Coaching for Informal Learning Activities," in *Intelligent Tutoring Systems*, D. Sleeman and J. S. Brown, Editors, Academic Press, New York (1982), pp. 79–98.
3. N. K. Sondheimer and N. Relles, "Human Factors and User Assistance in Interactive Computing Systems: An Introduction," *IEEE Transactions on Systems, Man, and Cybernetics* **12**, No. 2, 102–107 (March/April 1982).
4. C. Lewis and D. A. Norman, "Designing for Error," in *User Centered System Design*, D. A. Norman and S. W. Draper, Editors, Lawrence Erlbaum Associates, Hillsdale, NJ (1986), pp. 411–432.
5. E. Rich, "User Modeling Via Stereotypes," *Cognitive Science* **3**, 329–354 (1979).
6. J. Faletti, "PANDORA—A Program for Doing Commonsense Planning in Complex Situations," *Proceedings of the National Conference on Artificial Intelligence (AAAI-82)*, Carnegie-Mellon University, Pittsburgh, PA (1982), pp. 185–188.
7. J. F. Kelley, "An Iterative Design Methodology for User-friendly Natural-language Office Information Applications," *ACM Transactions on Office Information Systems* **2**, 26–41 (1984).
8. K. R. McKeown, M. Wish, and K. Matthews, "Tailoring Explanations for the User," *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (1985), pp. 794–798.
9. P. Jackson and P. Lefrere, "On the Application of Rule-based Techniques to the Design of Advice-giving Systems," in *Developments in Expert Systems*, M. J. Coombs, Editor, Academic Press, London (1984), pp. 177–200.
10. J. M. Carroll and J. McKendree, *Interface Design Issues for Advice-Giving Expert Systems*, Research Report RC 11984, IBM Thomas J. Watson Research Center, Yorktown Heights, NY (1986).
11. C. Holg, *The Joy of TENEX and TOPS-20 ... in Two Parts*, Technical Report ISI/TM 79-15, USC Information Sciences Institute, Marina del Rey, CA (January 1979).
12. R. Wilensky, Y. Arens, and D. Chin, "Talking to UNIX in English: An Overview of UC," *Communications of the ACM* **27**, No. 6 (June 1984), pp. 574–593.
13. U. Wolz and G. E. Kaiser, "A Discourse-based Consultant for

Interactive Environments," *IEEE Proceedings of the Fourth Conference on Artificial Intelligence Applications* (March 14–18, 1988), pp. 28–33.
14. R. S. Fenchel and G. Estrin, "Self-describing Systems Using Integral Help," *IEEE Transactions on Systems, Man, and Cybernetics* **12,** No. 2, 162–167 (March/April 1982).
15. D. Sleeman and J. S. Brown, *Intelligent Tutoring Systems*, Academic Press, New York (1982).
16. L. Quinn and D. M. Russell, "Intelligent Interfaces: User Models and Planner," *Proceedings of CHI '86: Human Factors in Computing Systems*, Boston, MA (April 13–17, 1986), pp. 314–320.
17. B. K. Reid, *Scribe: A Document Specification Language and Its Compiler*, Ph.D. thesis, Carnegie-Mellon University (1980).
18. E. Rich, "Users Are Individuals: Individualizing User Models," *International Journal of Man-Machine Studies* **18,** 199–214 (1983).
19. L. A. Miller, "Natural Language Programming: Styles, Strategies, and Contrasts," *IBM Systems Journal* **20,** No. 2, 184–215 (1981).
20. A. W. Bierman, B. W. Ballard, and A. H. Sigmon, "An Experimental Study of Natural Language Programming," *International Journal of Man-Machine Studies* **18,** 71–87 (1983).
21. J. A. Hendler and P. R. Michaelis, "The Effects of Limited Grammar on Interactive Natural Language," *Proceedings of CHI '83: Human Factors in Computing Systems*, New York (1983), pp. 190–192.
22. T. Winograd and C. F. Flores, *Understanding Computers and Cognition*, Ablex Publishers, Norwood, NJ (1986).
23. E. F. Codd, "HOW ABOUT RECENTLY?" (English dialog with relational databases using RENDEZVOUS Version 1), in *Databases: Improving Usability and Responsiveness*, Academic Press, New York (1978), pp. 3–8.
24. D. Waltz, "An English Language Question Answering System for a Large Relational Database," *Communications of the ACM* **21,** No. 7, 526–539 (July 1978).
25. M. Templeton, "EUFID: A Friendly and Flexible Frontend for Data Management Systems," *Proceedings of the 1979 National Conference of the Association for Computational Linguistics* (August 1979).
26. J. J. Robinson, *DIAGRAM: A Grammar for Dialogues*, AI Center Technical Note 205, SRI International, Menlo Park, CA (February 1980).
27. *Systems Application Architecture—Common User Access Panel Design and User Interaction*, SC26-4351-0, IBM Corporation (1987); available through IBM branch offices.
28. E. F. Wheeler and A. G. Ganek, "Introduction to Systems Application Architecture," *IBM Systems Journal* **27,** No. 3, 250–263 (1988).
29. R. Kimball, "A Self-improving Tutor for Symbolic Integration," in *Intelligent Tutoring Systems*, D. Sleeman and J. S. Brown, Editors, Academic Press, New York (1982), pp. 283–307.
30. A. P. Aaronson and J. M. Carroll, *The Answer Is in the Question: A Protocol Study of Intelligent Help*, Research Report RC 12034, IBM Thomas J. Watson Research Center, Yorktown Heights, NY (1986).
31. W. Teitelman and L. Masinter, "The InterLisp Programming Environment," *Computer* **14,** No. 4 (April 1981), pp. 25–34.
32. R. C. Houghton, "Online Help Systems: A Conspectus," *Communications of the ACM* **27,** No. 2 (February 1984), pp. 126–133.
33. T. Finin, "Providing Help and Advice in Task Oriented Systems," *Proceedings of the Eighth International Conference on Artificial Intelligence*, Karlsruhe, West Germany (1983), pp. 176–178.
34. *Operating System/2 Standard Edition 1.1, Volume 1*, 90X7934, IBM Corporation (1988); available through IBM branch offices.
35. M. D. Ringle and R. Halstead-Nussloch, "Shaping User Input: A Strategy for Natural Language Dialog Design," to be published in *Interacting with Computers* (1990).
36. F. C. Pereira and D. H. Warren, "Definite Clause Grammars for Language Analysis: A Survey of the Formalism and a Comparison with Augmented Transition Networks," *Artificial Intelligence* **13,** 231–278 (1980).
37. *Expert System Consultation Environment/VM and Expert System Development Environment/VM*, IBM IPS Service Support Center, Irving, TX (1985).
38. J. F. Sowa, *Conceptual Structures: Information Processing in Mind and Machine*, Addison-Wesley Publishing Company, Reading, MA (1984).
39. M. McCord, "Natural Language Processing in Prolog," in *Knowledge Systems and Prolog*, A. Walker, Editor, Addison-Wesley Publishing Company, Reading, MA (1987), pp. 316–324.
40. C. J. Fillmore, "The Case for Case," in *Universals in Language*, E. Bach and R. T. Harms, Editors, Holt, Rinehart & Winston, New York (1968).
41. W. R. Swartout, "Explaining and Justifying Expert Consulting Programs," *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI)*, Vancouver, BC (1981), pp. 815–822.
42. B. R. Gaines and J. N. Vickers, "Design Considerations for Hypermedia Systems," *Microcomputers for Information Management* **5,** No. 1, 1–27 (March, 1988).

John M. Prager *IBM Cambridge Scientific Center, 101 Main Street, Cambridge, Massachusetts 02142.* Dr. Prager is a project leader at the IBM Cambridge Scientific Center, which he joined in 1979. He worked initially on office systems, in particular on the POLITE project. His research contributions from that effort have earned him an Outstanding Innovation Award and two Invention Achievement Awards. His current activities include the development of user interfaces for powerful personal workstations, using techniques from artificial intelligence. He has published several papers and technical reports and is a member of the Association for Computing Machinery and the Institute of Electrical and Electronic Engineers Computer Society. Dr. Prager received a B.A., a Diploma in Computer Science (with Distinction), and an M.A., all from the University of Cambridge, and a Ph.D. in computer science from the University of Massachusetts at Amherst.

Donna M. Lamberti *IBM Cambridge Scientific Center, 101 Main Street, Cambridge, Massachusetts 02142.* Dr. Lamberti is a research staff member in the artificial intelligence (AI) and user interface area. She received a B.A. in Experimental Psychology from Vassar College, Poughkeepsie, New York, in 1982 and an M.S. degree in Cognitive Psychology, as well as a Ph.D. in Management Information Systems/Decision Sciences from Rensselaer Polytechnic Institute, Troy, New York, in 1987. Her thesis research was the development and empirical evaluation of an intelligent interface for a diagnostic expert system. This work was sponsored by an IBM Fellowship for research in information systems. She joined the Cambridge Scientific Center in 1987. Her current research interests include intelligent interface design for decision support technology, the design of AI-based advisory systems for organizations, and the implementation of decision support/knowledge-based systems.

David L. Gardner *IBM Systems Integration Division, 6300 Diagonal Highway, Boulder, Colorado 80301.* Mr. Gardner was research staff member on the REASON project. He received a bachelor's degree in electrical engineering from Brigham Young University in 1982 and a high technology M.B.A. degree from Northeastern University, Boston, Massachusetts, in 1989. From 1982 to 1986, he was involved in test engineering at the IBM Boca Raton, Florida, facility. From 1986 to 1989, he was a staff member at the IBM Cambridge Scientific Center. His interests include personal computers, workstations, computer-human interfaces, artificial intelligence, and the application of computing systems in the high-technology industry.

Stephen R. Balzac *IBM Palo Alto Scientific Center, 1530 Page Mill Road, Palo Alto, California 94304.* Mr. Balzac is a research staff member at the Palo Alto Scientific Center. He first worked for IBM as a co-op at the Thomas J. Watson Research Center. He received his S.B. and S.M. degrees in computer science from the Massachusetts Institute of Technology, Cambridge, Massachusetts, in 1987. His thesis work was the development of an interactive knowledge classification system using a semantic inheritance network. His current interests include artificial intelligence and knowledge representation.

Reprint Order No. G321-5391.