Intelligent Agents as a Basis for Natural Language Interfaces

By

David Ngi Chin

This dissertation has been submitted in partial satisfaction
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
JAN 1988		**00-00-1988 to 00-00-1988**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Intelligent Agents as a Basis for Natural Language Interfaces	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Typical natural language interfaces respond passively to the user's commands and queries. They cannot volunteer information, correct user misconceptions, or reject unethical requests. In order to do these things, a system must be an intelligent agent. UC (UNIX Consultant), a natural language system that helps the user solve problems in using the UNIX operating system, is such an intelligent agent. The agent component of UC is UCEgo. UCEgo provides UC with its own goals and plans. By adopting different goals in different situations, UCEgo creates and executes different plans, enabling it to interact appropriately with the user. UCEgo adopts goals from its themes, adopts sub-goals during planning, and adopts meta-goals for dealing with goal interactions. It also adopts goals when it notices that the user either lacks necessary knowledge, or has incorrect beliefs. In these cases, UCEgo plans to volunteer information or correct the user's misconception as appropriate. These plans are prestored skeletal plans that are indexed under the types of situations in which they are typically useful. Plan suggestion situations include the goal which the plan is used to achieve, the preconditions of the plan, and appropriateness conditions for the plan. Indexing plans by situations improves efficiency and allows UC to respond appropriately to the user in real time. Detecting situations in which a plan should be suggested or a goal adopted is implemented using if-detected daemons. The user's knowledge and beliefs are modeled by the KNOME (KNOwledge Model of Expertise) component of UC. KNOME is a double-stereotype system which categorizes users by expertise and categorizes UNIX facts by difficulty. KNOME deduces the user's level of expertise during the dialog with the user. After UCEgo has selected a plan, it is refined through the process of answer expression by the UCExpress component. UCExpress first prunes the answer to avoid telling the user something that the user already knows, and to mark where to use anaphora or ellipsis in generation. UCExpress also uses specialized expository formats to express different types of information in a clear, concise manner. The result is ready for generation into English.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**200**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

Intelligent Agents as a Basis for Natural Language Interfaces

David Ngi Chin

Intelligent Agents as a Basis for Natural Language Interfaces

David Ngi Chin

ABSTRACT

Typical natural language interfaces respond passively to the user's commands and queries. They cannot volunteer information, correct user misconceptions, or reject unethical requests. In order to do these things, a system must be an intelligent agent. UC (UNIX Consultant), a natural language system that helps the user solve problems in using the UNIX operating system, is such an intelligent agent.

The agent component of UC is UCEgo. UCEgo provides UC with its own goals and plans. By adopting different goals in different situations, UCEgo creates and executes different plans, enabling it to interact appropriately with the user. UCEgo adopts goals from its *themes*, adopts sub-goals during planning, and adopts *meta-goals* for dealing with goal interactions. It also adopts goals when it notices that the user either lacks necessary knowledge, or has incorrect beliefs. In these cases, UCEgo plans to volunteer information or correct the user's misconception as appropriate. These plans are prestored skeletal plans that are indexed under the types of situations in which they are typically useful. Plan suggestion situations include the goal which the plan is used to achieve, the preconditions of the plan, and *appropriateness conditions* for the plan. Indexing plans by situations improves efficiency and allows UC to respond appropriately to the user in real time. Detecting situations in which a plan should be suggested or a goal adopted is implemented using *if-detected daemons*.

The user's knowledge and beliefs are modeled by the KNOME (KNOwledge Model of Expertise) component of UC. KNOME is a *double-stereotype* system which categorizes users by expertise and categorizes UNIX facts by difficulty. KNOME deduces the user's level of expertise during the dialog with the user.

After UCEgo has selected a plan, it is refined through the process of *answer expression* by the UCExpress component. UCExpress first *prunes* the answer to avoid telling the user something that the user already knows, and to mark where to use anaphora or ellipsis in generation. UCExpress also uses specialized expository *formats* to express different types of information in a clear, concise manner. The result is ready for generation into English.

Preface

A Short History of UC

The original idea and motivation for UC, the UNIX Consultant, came from my advisor, Robert Wilensky, without whom this thesis and UC would not exist. The very first version of UC was written in early 1982 by myself. This version of UC used Yigal Arens's PHRAN parser/understander and an early version of his CLUSTER Context Modeller ([Arens, 1986]). It used a Conceptual Dependency ([Schank, 1975]) style representation, and produced canned output.

After the success of the initial version of UC, Yigal Arens (the senior graduate student at the time) was appointed by Robert Wilensky to head implementation of the UC project, and other students were brought in to work on various aspects of UC. Joe Faletti attempted to adapt his PANDORA commonsense planner ([Faletti, 1982]) for UC, Lisa Rau wrote an ellipsis understanding component for UC ([Rau, 1985]), Paul Jacobs wrote the PHRED generator for UC ([Jacobs, 1983]), Jim Mayfield started work on a goal analysis mechanism ([Mayfield, forthcoming]), and Jim Martin started work on the UCTeacher knowledge acquisition component ([Martin, 1985]). During this time, UC was adapted to run on the PEARL AI database system ([Deering et al., 1982]). My part of the project centered on the knowledge representation and inference mechanisms of UC ([Chin, 1983a] and [Chin, 1983b]).

This was an exciting period for working on UC. Transcripts of electronic mail to actual UNIX Consultants were collected and analyzed to see how UC might be improved. The group even performed a small experiment to see if users would behave differently when interacting with a human than when interacting with a program (simulated by a human unbeknownst to the subjects). This was described in [Chin, 1984]. All of this culminated in a version of UC that was described in the CACM article, [Wilensky et al., 1984].

Eventually some of the senior graduate students left (Yigal Arens and Joe Faletti), and new students joined the project. Charley Cox took over the task of building a parser/understander for UC ([Cox, 1986]), Marc Luria wrote a planner for doing things in UNIX ([Luria, 1985] and [Luria, forthcoming]), and Dekai Wu wrote a Concretion Mechanism for doing certain low level inferences. At this time, the KODIAK representation language ([Wilensky, 1987]) was developed at Berkeley in large part to address inadequacies in the representation scheme of the previous versions of UC. The initial implementation of KODIAK was written by Peter Norvig for his FAUSTUS story understander ([Norvig, 1986]) and this was adapted for use in UC. Also during this period, Lisa finished her Master's thesis and left; Paul Jacobs finished his Ph. D. thesis on the KING natural language generator ([Jacobs, 1986]) and left, leaving Anthony Albert to continue work on generation in UC. The implementation of UC was headed at various times by Rick Alterman (a postdoctoral fellow at Berkeley), Marc Luria, or myself. My work on UC shifted to the areas presented in this thesis (e. g. [Chin, 1986]). This period of UC's development culminated in the Berkeley Technical Report, [Wilensky et al., 1986].

Acknowledgements

I would like to thank my advisor Robert Wilensky not only for his support and guidance, but also for shaping my ideas on artificial intelligence. I wish to thank the other members of my thesis committee, Lotfi Zadeh and Eleanor Rosch, for their advice, and thank all of the above and the other members of my qualifying committee, Michael Stonebraker and Charles Fillmore, for pushing me toward excellence.

I would also like to thank all of the members of the BAIR (Berkeley Artificial Intelligence Research) group that were involved with UC, both for their help with UC and for their advice on the work described in this thesis: Anthony Albert, Richard Alterman, Yigal Arens, Charley Cox, Joe Faletti, Paul Jacobs, Marc Luria, Jim Martin, Jim Mayfield, Peter Norvig, and Lisa Rau. I would also like to thank the other members of BAIR not directly involved with UC: Nigel Ward, Terry Regier, Shigeo Omori, Eric Karlson, and Michael Braverman. I also wish to thank Feng Gao and Fred Douglis for helping to read drafts of this thesis. Additionally, I would like to thank Sharon Tague for being a friend as well as a good secretary.

I would like to thank the sponsors of this research, which include: the Defense Advanced Research Projects Agency (DoD), under Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract N00039-84-C-0089; the Office of Naval Research, under grant N0014-48-C-0732; the National Science Foundation under grant #85-14890; and Hughes Aircraft grant #442427-59868.

I would like to thank my sister Patti C. Sagastegui and my brother-in-law David Sagastegui for their emotional support and many home-cooked meals during these years. Also, I would like to thank my brother Stanley Chin for his emotional support at home with my parents. Finally, most importantly, I wish to thank my parents, Norman and Lucinda Chin for everything!

Table of Contents

Chapter I

Introduction

1. Consultation Programs

Consider the problem of building a program that simulates a human consultant. A user would be able to come up to such a program and obtain advice in the program's domain of expertise by entering queries in English (or some other natural language). The consultant program would then provide solutions in English. A user might ask for advice about how to do things, for definitions of terminology, or for advice in solving problems. In short, this program would behave like a real human consultant.

In order to build such a system, one needs to satisfy at least three requirements. First, the computer system needs to be able to understand the user's queries. Next, the program must be able to solve the user's problems and formulate an answer. Finally, the system must be able to convey the solution to the user in a clear, concise manner. Needless to say, there are many difficult and unsolved problems in each of these areas. The first requirement, understanding the user, involves the whole of natural language understanding, a difficult area of artificial intelligence research. The second requirement, problem solving, has a long continuing history of research in AI. The last requirement, communicating the answer to the user, has a shorter history of research in AI, but is no less difficult a problem.

However, even if all of the problems in each of the three areas were to be solved, and one could build a natural language consultation system that did each of the three tasks perfectly, that would still not be enough for a good natural language consultation system. A good consultation system also needs to be able to take the initiative in a dialog with the user, rather than always responding passively to the user. For instance, consider the following user interaction with a hypothetical program that provides advice on using the UNIX[1] operating system:

User: What does ls -v do?

Program: It lists the contents of your current directory.

The hypothetical program gives an answer that is literally correct, since the ls command actually ignores inappropriate flags such as -v. However, a consultant that provides only the above answer has failed to correct the user's incorrect preconception that the ls command has a -v flag. So, although the user did not specifically ask whether ls has a -v flag, a good consultant would not fail to provide the information that in fact ls does not have such a flag. Such a response is shown in the next dialog:

---

[1] UNIX is a trademark of Bell Laboratories.

User: What does ls -v do?

Program: There is no -v option for ls.

This is a better answer even though it literally does not answer the user's question. In deciding to ignore the user's direct question and shift its attention to the user's misconception, the program has taken the initiative in the dialog. This shows that a good consultant cannot just passively respond to the user; rather, it must have its own agenda.

In the previous dialog, the better answer requires that the program realize that the user has a misconception. To do this, the system must first infer that the user believes that ls has a -v option and then realize that the user's belief conflicts with the program's knowledge. So, in general, a good consultant system must also model the user's knowledge and beliefs.

Even after a consultant system realizes that the user has a misconception, it must decide how to deal with the misconception. In the above example, the system decides that it should inform the user of the facts in order to correct the user's misconception. In other cases, the system may choose to ignore the misconception, as in the following scenario:

User: How can I delete someone else's file when I don't have write permission on the file?

Program: I will not help you delete someone else's file because that is unethical.

In the user's statement above, the user has assumed that one needs write permission on the file to delete it. This is not true. Rather, one needs write permission on the parent directory to delete the file. Regardless of what is the correct precondition, the program decides not to help the user because of ethical considerations. This also means that the program decides not to correct the user's misconception, so as to avoid helping the user delete someone else's file. This is an example of a decision by a consultant program to be uncooperative.

Of course a good consultant program cannot arbitrarily decide to be uncooperative. In the previous case, the decision to be uncooperative was based on the fact that the user's goal of deleting someone else's file conflicts with the program's goal of preserving all users' files. In this case, the program's goal of preserving files wins out over the program's desire to help the user who asked the question. These sorts of goals and goal interactions are needed to guide a consultant system properly.

After a course of action has been determined, it must be carried out. Even in the simplest case where the program just answers the user's question, there are still many ways to express the answer to the user. In the following scenario, F1 and F2 represent two possible formats for expressing the answer to the user's question.

User: How can I change the group protection of a file?

F1: To change the permission of a file, type 'chmod' followed by one or more spaces or tabs followed by 'g', followed by either '+' for add permission or '-' for remove permission, followed by the type of permission, which is either 'r' for read permission, 'w' for write permission, or 'x' for execute permission, followed by or more spaces or tabs followed by the name of the file to be changed, followed by a carriage return.

F2: Use chmod.
For example, to add group read permission to the file foo,
type 'chmod g+r foo'.

The first form of the answer, F1, is correct and quite general, but it is also so verbose that it is undecipherable. On the other hand, the second form of the answer, F2, is succinct and gives the user information in an easily readable form, but it is considerably less general. In fact the second format is somewhat inaccurate, since the example strictly applies only to adding group read permission to the file foo. It is up to the reader to use analogous reasoning to apply this to other cases. Despite this lack of generality, the second answer form is clearly superior to the first. Note that the second form requires additional computation to transform the general solution of F1 into an *example*. A natural language system needs to incorporate knowledge about when and how to use special presentation formats like examples to convey information more clearly to the user.

2. Intelligent Agents

The examples in the previous section have shown that a good computer consultation system cannot be just a passive question-answering system. Rather, the consultant system must often take the initiative. This is because consultant systems generally have greater knowledge in their field of expertise than users. As in the example in the previous section, the consultant may sometimes need to take the initiative in order to correct a user's misconceptions. Also, the consultant may need to take the initiative in order to provide needed information that the user did not explicitly ask for. In fact, the user often does not even realize that such information is pertinent, so will never ask for it. A computer consultant system needs to have the human-like capability of taking the initiative in a dialog rather than always responding to the user passively.

Previous efforts in natural language systems that take the initiative have resulted in programs capable of "mixed-initiative" dialogs. Among the first of these, the SCHOLAR system ([Carbonell, 1970a&b]) for CAI (Computer-Aided Instruction), could take the initiative to test the user on facts in its knowledge base. It also allowed the user to query the system about facts in its knowledge base. This type of "mixed-initiative" works only for limited situations such as mutual quizzing. A system based on answering or generating quiz questions cannot be adapted to help the user solve problems, volunteer information pertinent to the user's problems, or detect misconceptions evident from the

user's questions (as opposed to wrong answers from the user).

More recent efforts have followed one of two approaches to the problem of taking the initiative in a dialog: script-based or frame-based. In the script-based approach, the system follows a set script of interchanges with the user. For example, the hotel reservation application of the HAM-ANS system ([Hoeppner et al., 1984]) used a fixed series of exchanges in which the system has the initiative part of the time. The other approach, frame-based initiative, is exemplified by the GUS system ([Bobrow et al., 1977]). GUS took the initiative in the dialog when it needed to fill in information for a slot in a frame. Typically, the frame would represent information that GUS needed in order to address the user's problem. Each of these approaches works to provide programs with the capability to take the initiative in limited situations. However, none is general enough to cover other types of situations where a program should take the initiative. For example, neither approach would allow a program to take the initiative to correct a user misconception.

The approach that I take is to view the program as an *agent*. That is, a consultation system should be viewed as a system that can perform actions. For a natural language consultation system, acting consists of mostly *speech acts* ([Austin, 1962] and [Searle, 1969]), i. e. acting by communicating with the user. Within this paradigm, taking the initiative in a dialog translates into acting without the guidance of the user. An agent that has this capability of taking the initiative is called an *autonomous agent*.

A *rational agent* is an agent that behaves rationally. In AI programs, much as in popular psychology, this usually means attributing reasonable plans and goals to the agent in question. For example, PAM ([Wilensky, 1978]) understood stories involving rational agents by analyzing the goals and plans of the characters. Likewise, TALE-SPIN ([Meehan, 1976]) used plans and goals to create simple stories with rational agents as characters. In the problem solving domain, robot planning programs (e. g. [Fikes and Nilsson, 1971] and [Sacerdoti, 1974]) have shown that planning is a good paradigm for programming robots as rational agents. In the realm of conversation, [Hobbs and Evans, 1980] have argued that human conversation fits such a paradigm. So, a program that contains reasonable plans and goals and that is guided by those plans and goals can be considered a rational agent.

Within the planning paradigm, a *rational autonomous agent* is one that contains plans and goals that allow the agent to take the initiative in appropriate situations. The central problem in building autonomous agents is determining which situations require the agent to take the initiative. For a rational autonomous agent that is based on the planning paradigm, this problem translates to the problem of determining appropriate goals for the planner. Such a process is called *goal detection* ([Wilensky, 1983]).

After a rational autonomous agent, henceforth called an *intelligent agent*, has detected appropriate goals, it is up to the planner of the intelligent agent to formulate a plan to satisfy these goals and then carry out the plan. Much work has been done in AI in the area of planning where the goals of the planner are provided by the operator. For example, [Newell and Simon, 1972] formulated "means-ends" analysis as a general strategy for achieving given "ends" or goals. However, the robot planning programs all assumed that the goals are given by the users. Likewise in TALE-SPIN, the programmer provided the initial goals of the characters. For story understanding, PAM was able to

recognize a character's goals when directly stated in the narrative, or when the goal could be inferred from the characters' stated actions based on the assumption that the actions were part of the character's plan. As [Carbonell, 1982] points out, none of these systems have systematically addressed the problem of goal detection, which is essential for building intelligent agents.

The PANDORA planner ([Faletti, 1982]), which implemented some of the commonsense planning ideas presented in [Wilensky, 1983], did perform some goal detection. PANDORA detected goals when actual or projected states conflicted with goals or plans. Also, certain goals were attached to the frames describing situations. For example, the goal of "find out about the world" was attached to the "morning" frame, which meant that PANDORA would try to read a newspaper in the morning. However, except for very simple frames, PANDORA did not address the problem of when is it proper to invoke frames and their associated goals. Also, because PANDORA existed in a self-contained simulated world, it did not address the problem of detecting goals when the system must interact with real users.

In order to interact intelligently with its environment, an agent needs knowledge about its environment. Such knowledge is needed for the agent to respond properly to stimuli, which for a rational agent is equivalent to detecting the right goals and carrying out the right plans. In the case of an agent that is a computer consultation system, the environment consists of a discourse with the user on the system's domain of expertise. Thus, the intelligent agent needs to have information about its domain of expertise and about its user. A computer consultation system could not be built without some information about its domain; however, information about its user is less commonly found in such systems. Yet, a model of the user is essential for a computer consultation system that attempts to embody an intelligent agent.

This thesis addresses the problem of building a natural language computer consultation system that behaves as an intelligent agent. The ideas presented in this thesis are implemented in the UC (UNIX Consultant) system, which embodies an intelligent agent.

3. UC, the UNIX Consultant

UC (UNIX Consultant) is an interactive natural language consultant system for the UNIX operating system. UC is able to provide information about how to do things in UNIX, provide definitions about UNIX or general operating system terminology, and provide help in debugging problems with using UNIX.

The main purpose of the UC project is to investigate fundamental issues of AI in natural language processing, knowledge representation, planning and problem solving, and building integrated natural language interfaces. The UC program serves as a test-bed for ideas on how to approach these problems. Successful ideas are carried over to succeeding versions of UC, while unsuccessful ideas are rethought. Although UC is completely implemented, it is still only an extendible prototype and does not contain enough knowledge to be usable in the real world. Indeed, producing a usable product is an inappropriate pursuit for a research institution; so, UC is only implemented in enough breadth to validate its research ideas and to show that a real system might actually be

constructed along the lines suggested by the research.

A short overview of UC follows. For more details on other aspects of UC not discussed in this thesis, the reader is referred to [Wilensky et al., 1986] and [Wilensky et al., 1984].

In a typical UC session, the user types questions in English to UC, and UC responds to the user in English. A schematic diagram of the flow of information among UC's various components is shown in Figure 1.1. The input to UC is analyzed by the ALANA language analysis component of UC, which produces a semantic representation of the input. This representation is in the form of a KODIAK network (see Appendix A). Next, UC's Concretion Mechanism performs concretion inferences ([Wilensky, 1983] and [Norvig, 1983]) based on the semantic network. Concretion is the process of inferring more specific interpretations of the user's input than might strictly be correct on a logical basis. Such inferences might be motivated by the context of the utterance or by culturally accepted usage biases. After concretion, the modified KODIAK network is passed to PAGAN, UC's goal analysis component. PAGAN deduces the user's actual goals. This includes inferring the user's high-level goals as well as the user's immediate goals. PAGAN also handles phenomena such as indirect speech acts.

After the initial analysis of the user's input, the UCEgo component of UC decides how UC should respond. UCEgo is the component of UC that implements an intelligent agent. It first determines what UC's own goals should be, then formulates a plan to achieve these goals, and finally carries out this plan. UCEgo detects its own goals based on the present situation, which may include such varied factors as the user's goals, the user's utterances, as well as UC's own goals, knowledge, and internal state.

Part of UCEgo's response may involve calling on the services of the UCPlanner component of UC. UCPlanner is a UNIX domain planner that creates plans for doing things in UNIX. Another component of UC that may be called by UCEgo is UCExpress. UCExpress uses the process of answer expression to refine the communicative plans produced by UCEgo. First, UCExpress prunes extraneous concepts from the answer, either when the user already knows the concepts, or when the concepts either are already part of the conversational context. Next UCExpress uses specialized formats such as similes and examples to express information to the user in a clear and concise manner. The result of UCExpress' processing is an annotated KODIAK network that is ready for generation into English by the UCGen tactical level generator.

Another important part of UC is KNOME, the user modeling component. KNOME encodes the knowledge and beliefs of the user. It also deduces what the user knows and believes based on UC's conversation with the user. KNOME also models the extent of UC's own knowledge of UNIX. This is useful in differentiating between actual user misconceptions and cases in which UC's knowledge base is incomplete.

This thesis is mainly concerned with the UCEgo, UCExpress, and KNOME components of UC. UCEgo is composed of two parts, a goal detector and a plan selector/executor. The goal detector detects appropriate goals for UC according to the situation. The plan selector/executor then takes these goals and plans for them, eventually executing the plans. Goal detection in UCEgo is described in Chapter III, and plan

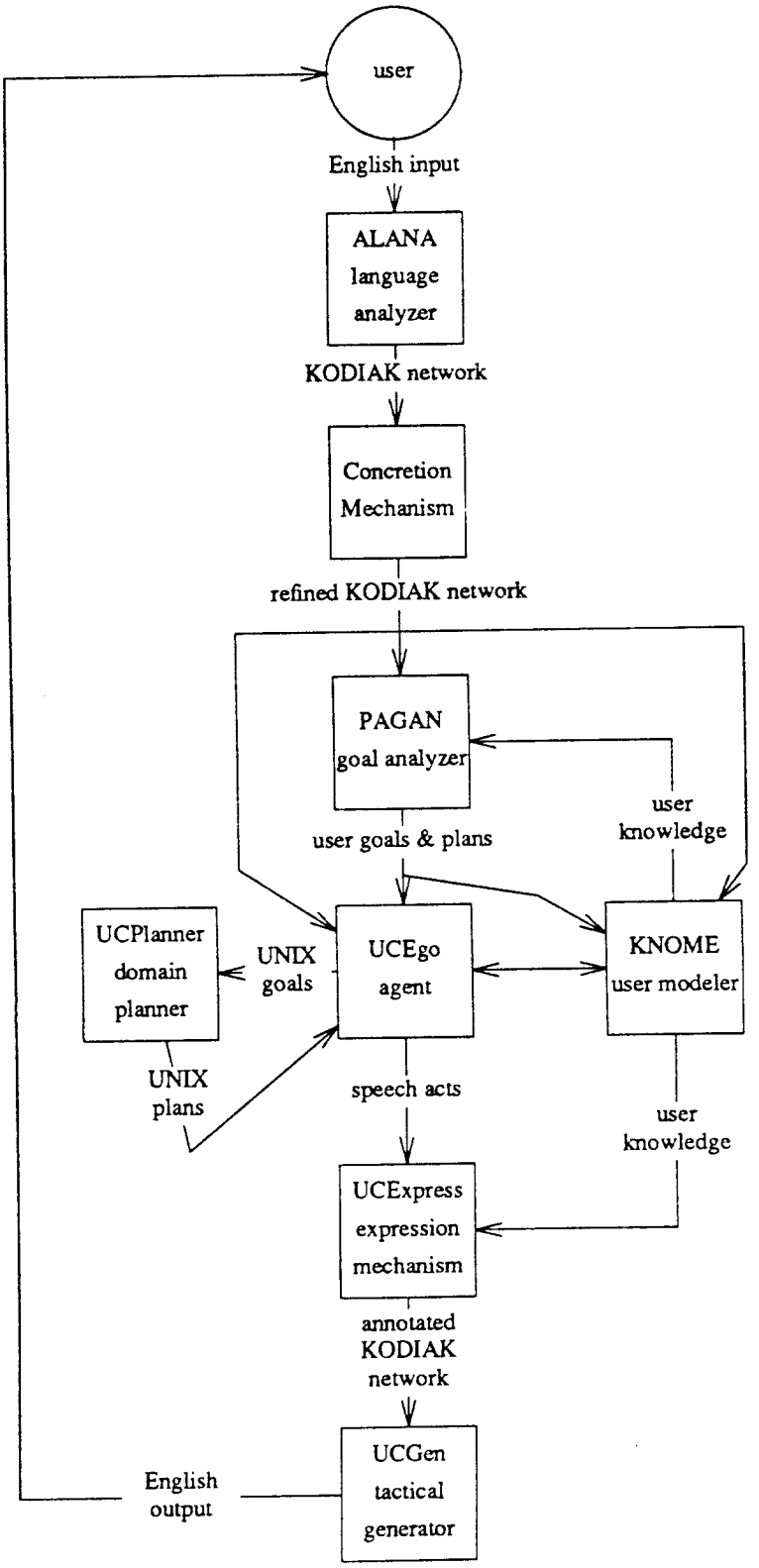

Figure 1.1. Flow of information among UC components.

selection/execution is discussed in Chapter IV. Refinement of the plans by UCExpress is described in Chapter V, and KNOME, the user modeling component, is described in Chapter II.

3.1. A Session with UC

To see how UCEgo works in more detail, consider the following trace of a UC session. In the trace, actual output from UC to the user is shown in **Bold Courier font**, trace output from UC is shown in Courier font, and input from the user to UC is shown in *Oblique Courier font*. Explanations of the processing are in normal font and are separated by half lines from the actual trace.

In the trace output, an "&" represents an abbreviation by UC of a concept that was completely specified earlier in the trace (UC marks previously printed concepts so that it will know when it can abbreviate). Where parts of the trace are left out for brevity, this is indicated by three dots. In order to fit the trace on the page, the indentation of some of the LISP forms were adjusted by hand. These conventions will be used throughout the rest of this thesis for trace output from UC.

The following trace demonstrates what UC can do, concentrating on the capabilities of the UCEgo, KNOME, and UCExpress components. In the trace, the user asks UC six questions. The somewhat lengthy trace of processing the user's first question illustrates some of the more basic capabilities of the system, including: how KNOME deduces what the user knows from the user's query; how UCEgo detects goals, plans for those goals, and executes the resultant plans; and how UCExpress refines those plans through the process of answer expression.

The next five examples demonstrate some of the more exotic capabilities of UC. The second query shows UC correcting a user misconception, and the third example illustrates the use of an example format by UCExpress to more clearly convey information to the user. UC's reply to the user's fourth question shows UC volunteering information to the user, and the user's followup question elicits an answer in UCExpress' simile format. Finally, the last example shows UCEgo refusing to help the user with an unethical request.

```
Welcome to UC (Unix Consultant) version 3.23
To a '#' prompt, please type in your questions
about the UNIX file system in English.
To leave, just type '^D' or '(exit)'.

Hi.
How can I help you?
```

This first user query is used to show some of the more basic features of UC.

```
# How can I find out the protection of a file?

The parser produces:
(ASK10 (listener10 = UC)
       (speaker10 = *USER*)
       (asked-for10 =
         (QUESTION10 (what-is10 = (ACTION14? (actor14 = *USER*))))))
(INFORMATION-ACTION0?
  (actor0-1 = *USER*)
  (INFORMATION-ACTION-effect0 =
    (INFORMATION-EFFECT1?
      (desired-info1 =
        (STATE-OF-FILE0 (file-state0 = FILE-PROTECTION1)
                        (state-file0-0 = FILE4?)))
      (informant1 = (actor0-1 = aspectual-of
                                (INFORMATION-ACTION0? &)))))
  (cause0-0 = (ACTION14? &)))
```

The first step in UC's processing is done by UC's parser/understander component which produces a KODIAK semantic network representation of the meaning of the user's utterances. The KODIAK representation system used in UC is described in Appendix A. The above linearized form of KODIAK represents the fact that the user (*USER*) has asked (ASK10) UC a question about what is the referent of ACTION14. ACTION14 is the cause of INFORMATION-ACTION0, which represents the user finding out the file-protection (FILE-PROTECTION1) state (STATE-OF-FILE0) of some file (FILE4).

```
UCEgo detects the following concepts:
FILE-PROTECTION1
and asserts the following concept into the database:
(user-knows6 (uk-fact6 = FILE-PROTECTION1)
             (uk-user6 = *USER*)
             (uk-truth-val6 = TRUE))

KNOME: Asserting *USER* knows FILE-PROTECTION1
KNOME: Since FILE-PROTECTION1 is a FILE-PROTECTION,
       asserting *USER* knows FILE-PROTECTION
KNOME: FILE-PROTECTION has difficulty MUNDANE, so deducing:
KNOME: ruling out *USER* = NOVICE
KNOME: *USER* is SOMEWHAT-UNLIKELY to be BEGINNER
       => likelihood(*USER* = BEGINNER) = UNCERTAIN
KNOME: *USER* is SOMEWHAT-LIKELY to be INTERMEDIATE
       => likelihood(*USER* = INTERMEDIATE) = SOMEWHAT-LIKELY
KNOME: *USER* is LIKELY to be EXPERT
       => likelihood(*USER* = EXPERT) = LIKELY
```

KNOME, the user modeling component of UC, is described in Chapter II. KNOME models the user's knowledge in the UNIX domain. It infers individual facts about what the user does or does not know from what the user says, and then combine this evidence to figure out the user's level of expertise in using UNIX. Here, from the fact that the user correctly uses the concept of FILE-PROTECTION in a sentence, KNOME infers that the user must know that concept. This deduction allows KNOME to make further inferences about the user's level of expertise in the UNIX domain. Since FILE-PROTECTION is a concept of a mundane difficulty level, KNOME infers that the user could not be a novice, is somewhat unlikely to be a beginner, is somewhat likely to be an intermediate, and is likely to be an expert (actually, although an expert is likely to know about file protection, an expert would also know how to change a file's protection and would not ask such a question; this latter inference appears later in the trace).

Ordinarily, the inferences that KNOME makes about the user's level of expertise are immediately useful to UC in forming its answer. However, in this particular case, these inferences will not be used until UC processes the user's next query.

```
The goal analyzer produces:
((HAS-GOAL-ga0
   (planner-ga0 = *USER*)
   (goal-ga0 = (KNOW-ga0? (knower-ga0 = *USER*)
                          (fact-ga0 = (ACTION14? &))))))
```

Based on the fact the user asked UC a question about how to do something, UC's goal analysis component infers that the user's goal is to know ACTION14, which represents how to find out the protection of a file.

```
UCEgo: suggesting the plan:
(PLANFOR73 (goals73 = (HELP5 (helpee5 = *USER*)
                             (helper5 = UC)))
           (plan73 = (SATISFY6 (need6 = (KNOW-ga0? &))
                               (actor6 = UC))))
based on the situation:
(UC-HAS-GOAL63 (status63 = ACTIVE) (goal63 = (HELP5 &)))
(HAS-GOAL-ga0 &)
```

UCEgo suggests specific plans when it encounters certain specialized *situations*. In this case, UCEgo suggests that a plan (PLANFOR73) for helping the user is to satisfy the user's goal of knowing (KNOW-ga0) how to find out the protection of a file. In general, UCEgo suggests the plan of helping the user by satisfying the user's goal of knowing, whenever UCEgo encounters a situation in which UC has the goal of helping the user, and the user has the goal of knowing something. UC's goal of helping the user

originated at the start of the session from UC's *role theme* of being a UNIX consultant. Themes and the goals which arise from themes are discussed in Chapter III, Section 2. The suggestion of plans for carrying out these goals is described in Chapter IV, Section 2. The actual detection of situations is performed by *if-detected daemons*, which are described in Chapter VI.

```
UCEgo detects the following concepts:
(HAS-GOAL-ga0 &)
and asserts the following concept into the database:
(user-knows7 (uk-user7 = *USER*)
             (uk-truth-val7 = FALSE)
             (uk-fact7 = (ACTION14? &)))

KNOME: Asserting *USER* does not know ACTION13?
```

The fact that the user wants to know how to find out the protection of a file allows KNOME to infer that the user does not know how. Since "how to find out the protection of a file" is a description of an object rather than an actual object, KNOME does not immediately use this fact to infer the user's level of expertise as it did when it inferred that the user knew the concept of file protection. Later, after UC identifies the referent of the description, KNOME will then make additional inferences about the user's level of expertise.

```
UCEgo detects the following concepts:
(UC-HAS-GOAL63 &)
(PLANFOR73 &)
and asserts the following concept into the database:
(UC-HAS-INTENTION9 (intention9 = (SATISFY6 &)) (status9 = ACTIVE))
```

Here, UCEgo adopts the *intention* of carrying out the plan for helping the user (intentions are described in Chapter IV, Section 3.1). An intention has two possible meanings depending upon the type of plan. If the plan is a series of actions, then the intention means that UCEgo has scheduled the plan for execution during its plan execution stage (see Chapter IV, Section 3). On the other hand, if the plan is to satisfy some state (as in this case), then the intention means that UCEgo will adopt that state as a sub-goal and continue its planning process. This is shown below: UCEgo adopts the sub-goal of having the user know how to find out the protection of a file.

```
UCEgo: detected the goal:
(UC-HAS-GOAL66 (goal66 = (KNOW-ga0? &)))
from the situation:
```

```
(UC-HAS-INTENTION9 &)

UCEgo detects the following concepts:
(PLANFOR-ga1 (goals-ga1 = (INFORMATION-EFFECT1? &))
             (plan-ga1 = (ACTION14? &)))
(UC-HAS-GOAL66 &)
and asserts the following concept into the database:
(UNIX-planner2 (user-goals2 = (INFORMATION-EFFECT1? &)))
```

Since UC has the goal of knowing a plan for something, it calls the UNIX domain planner component of UC in case the domain planner can produce a plan for UC. This type of reasoning is described in Chapter IV, Section 3.2.

```
The planner is passed:
((INFORMATION-EFFECT1? &))

The planner produces:
(PLANFOR460 (goals460 = (INFORMATION-EFFECT1? &))
            (plan460 = UNIX-LS-1-COMMAND0))
(LS-1-HAS-FORMAT0
  (LS-1-HAS-FORMAT-command0 = UNIX-LS-1-COMMAND0)
  (LS-1-HAS-FORMAT-format0 =
    (LS-1-FORMAT1 (LS-1-FORMAT-step1 =
                    (SEQUENCE30 (next30 = -1)
                                (step30 = ls))))))
(HAS-EFFECT120
  (command-of-effect120 = UNIX-LS-1-COMMAND0)
  (effect-of-command120 = (LIST-EFFECT10
                            (list-object10 = FILE-PROTECTION0?))))
(HAS-OPTION20 (command-of-option20 = UNIX-LS-1-COMMAND0)
              (option20 = -1-OPTION0))
(HAS-COMMAND-NAME180
  (HAS-COMMAND-NAME-named-obj180 = UNIX-LS-1-COMMAND0)
  (HAS-COMMAND-NAME-name180 = (SEQUENCE30 &)))
(LS-1-FORMAT0 (LS-1-FORMAT-step0 = (SEQUENCE30 &)))
```

The UNIX domain planner produces the plan of using the ls -l command.

```
UCEgo: suggesting the plan:
(PLANFOR74
  (goals74 = (KNOW-ga0? &))
  (plan74 = (TELL6 (effect6 = (STATE-CHANGE1
                                (final-state1 = (KNOW-ga0? &))))
                   (listener6-0 = *USER*)
                   (speaker6-0 = UC)
```

```
                    (proposition6 = (PLANFOR460 &)))))
based on the situation:
(ANSWER-FOR4 (answer4 = (PLANFOR460 &))
             (query4 = (ACTION14? &)))
(UC-HAS-GOAL66 &)
```

Based on the fact that PLANFOR460 (representing using the ls -l command) is an answer for the query of how to find out the protection of a file, UCEgo suggests the plan of telling the user the answer in order to achieve the sub-goal of having the user know how to find out the protection of a file.

```
UCEgo detects the following concepts:
(ANSWER-FOR4 &)
(KNOW62 (knower62 = *USER*)
        (fact62 = (ACTION14? &))
        (truth-val62 = FALSE))
and asserts the following concept into the database:
(user-knows8 (uk-fact8 = (PLANFOR420 &))
             (uk-user8 = *USER*)
             (uk-truth-val8 = FALSE))

KNOME: Asserting *USER* does not know PLANFOR460
```

Based on the previous inference that the user does not know how to find out the protection of a file and the fact that using ls -l is a plan for finding that out, KNOME infers that the user does not know that ls -l is a plan for finding out the protection of a file (the meaning of PLANFOR460).

```
UCEgo detects the following concepts:
(UC-HAS-GOAL66 &)
(PLANFOR74 &)
and asserts the following concept into the database:
(UC-HAS-INTENTION10 (intention10 = (TELL6 &))
                    (status10 = ACTIVE))
```

UC adopts the intention of carrying out the plan of telling the user the answer.

```
UCEgo detects the following concepts:
(UC-HAS-INTENTION11 &)
and asserts the following concept into the database:
(UCexpress3 (gen-prop3 = (TELL6 &)))
```

Telling the user involves passing the proposition to the UCExpress component, which does answer expression. UCExpress is described in Chapter V.

```
UCEgo detects the following concepts:
(KNOW63 (knower63 = *USER*)
        (fact63 = (PLANFOR460 &))
        (truth-val63 = FALSE))
and asserts the following concept into the database:
(user-knows9 (uk-fact9 = UNIX-LS-1-COMMAND0)
             (uk-user9 = *USER*)
             (uk-truth-val9 = FALSE))

KNOME: Asserting *USER* does not know UNIX-LS-1-COMMAND0
KNOME: Since UNIX-LS-1-COMMAND0 is a UNIX-LS-1-COMMAND,
       asserting *USER* does not know UNIX-LS-1-COMMAND
KNOME: UNIX-LS-1-COMMAND has difficulty MUNDANE, so deducing:
KNOME: ruling out *USER* = EXPERT
KNOME: *USER* is SOMEWHAT-LIKELY to be BEGINNER
       => likelihood(*USER* = BEGINNER) = SOMEWHAT-LIKELY
KNOME: *USER* is SOMEWHAT-UNLIKELY to be INTERMEDIATE
       => likelihood(*USER* = INTERMEDIATE) = UNCERTAIN
```

Since the user does not know the main usage of ls -l (the meaning of a PLANFOR), KNOME can infer that the user must not be familiar with ls -l. Based on this, KNOME makes further inferences about the user's level of expertise. Since the user does not know ls -l, KNOME infers that the user cannot be an expert, is somewhat unlikely to be an intermediate, and is somewhat likely to be a beginner.

```
Express:  now expressing the PLANFOR:
(PLANFOR460 &)

Express:  expressing a complete plan, so expanding the action
PLANFOR460 into its subcomponents:
TYPE-ACTION0
```

In expressing the answer, UCExpress chooses to compress the information in the plan of "use the UNIX ls -l command which has the format of ls followed by -l" into the more succinct subaction, "type 'ls -l'," since the shorter answer is easier to read and understand.

```
Express:  not expressing INFORMATION-EFFECT1?,
```

```
since it is already in the context.
```

UCExpress concludes that it does not have to express the concept, INFORMATION-EFFECT1, which represents the phrase "find out the protection of a file," since this is already part of the context of the dialog.

```
The generator is passed:
(TELL6 &)
```

The actual translation of concepts into English is done by the tactical level generator component of UC.

Type 'ls -l'.

The next query illustrates how UC corrects user misconceptions.

```
# What does ls -p do?

The parser produces:
(ASK11 (listener11 = UC)
       (speaker11 = *USER*)
       (asked-for11 = (QUESTION11 (what-is11 = STATE13?))))
(HAS-EFFECT21? (effect-of-command21 = STATE13?)
               (command-of-effect21 = UNIX-LS-COMMAND0))
(HAS-OPTION3 (option3 = -p-OPTION0)
             (command-of-option3 = UNIX-LS-COMMAND0))
```

The parser/understander interprets the user's question as a query about the effects (STATE13) of UNIX-LS-COMMAND0 which has a -p option.

-
-
-

The processing up to this point is similar to the previous example: the goal analysis

component infers that the user's goal is to know the effects of ls -p, and UCEgo adopts the user's goal of knowing as a sub-goal of helping the user (UC-HAS-GOAL67).

```
UCEgo detects the following concepts:
(HAS-EFFECT21? &)
(UC-HAS-GOAL67 &)
and asserts the following concept into the database:
(UC-find-effects1 (unknown-command1 = UNIX-LS-COMMAND0))
```

UC-find-effects is a procedure which finds the effects of UNIX commands.

```
UCEgo:  trying to find effects for UNIX-LS-COMMAND0

UCEgo:  unknown relation:
(HAS-OPTION3 &)
UCEgo:  User has the misconception:
(HAS-MISCONCEPTION1 (confused1 = *USER*)
                    (misconception1 = (HAS-OPTION3 &)))
since
(KNOW42 (fact42 = (ALL3 (such-that3 =
                          (HAS-OPTION0? (option0 = (ALL3 &))
                                        (command-of-option0 =
                                          SIMPLE-COMMAND0?)))
                        (all-type3 = OPTION0?)))
        (knower42 = UC))
and since the user believes:
(HAS-OPTION3 (option3 = -p-OPTION0)
             (command-of-option3 = UNIX-LS-COMMAND0))
which involves an unknown OPTION
```

In the process of finding the effects of the UNIX-LS-COMMAND0, UCEgo notices that it has an option (HAS-OPTION3) which does not have a analog in UC's knowledge base. This, in conjunction with the fact that UC knows all of the options of simple commands (of which ls is a member), tells UC that the user has a misconception. What UC knows is represented by KNOME using *meta-knowledge* , such as the above fact to model the limitations of UC's own knowledge. Meta-knowledge is described in Chapter II, Section 4. Correcting user misconceptions is described in Chapter III, Section 4.

```
UCEgo: suggesting the plan:
(PLANFOR76
  (goals76 = (HELP5 &))
  (plan76 =
    (SATISFY8 (need8 =
                (KNOW67? (knower67 = *USER*)
```

```
                        (fact67 = (NEGATE1 (negative1 =
                                            (HAS-OPTION3 &))))))
              (actor8 = UC))))
based on the situation:
(UC-HAS-GOAL63 &)
(HAS-MISCONCEPTION1 &)
```

To satisfy the goal of helping the user, UCEgo suggests the plan of having the user know that ls does not have a -p option.

-
-
-

The processing here follows the same pattern as in the previous example: UCEgo adopts the intention of carrying out the plan, passes the proposition to UCExpress which passes the concepts to the tactical generator.

---

ls does not have a -p option.

UC's response to the next user question shows how UCExpress uses an example format to more clearly express information to the user.

```
# How can I add write permission to a file?

The parser produces:
(ASK12
  (listener12 = UC)
  (speaker12 = *USER*)
  (asked-for12 =
    (QUESTION12 (what-is12 = (ACTION15? (actor15 = *USER*))))))
(CHANGE-PROT-FILE-ACTION0?
  (ch-prot-effect0 = (CHANGE-PROT-FILE-EFFECT0?
                       (change-prot0 = FILE-PROTECTION2)
                       (change-file0 = FILE5?)))
  (actor0-2 = *USER*)
  (cause0-1 = (ACTION15? &))))
(HAS-PROT-VALUE1 (value-protection-type1 = ADD-STATUS)
                 (prot-type-arg1-0 = FILE-PROTECTION2))
(HAS-FILE-PROTECTION2 (prot-file2 = FILE5?)
                      (file-prot2 = FILE-PROTECTION2))
```

```
(HAS-ACCESS-TYPE1 (access-protection-type1 = WRITE-PROT)
                  (prot-type-arg1 = FILE-PROTECTION2))

UCEgo detects the following concepts:
FILE-PROTECTION2
and asserts the following concept into the database:
(user-knows11 (uk-fact11 = FILE-PROTECTION2)
              (uk-user11 = *USER*)
              (uk-truth-val11 = TRUE))

KNOME: Asserting *USER* knows FILE-PROTECTION2
KNOME: FILE-PROTECTION2 is a FILE-PROTECTION, but KNOME
       already knows that *USER* knows FILE-PROTECTION,
       so not making any more deductions.
```

KNOME already inferred from the user's first query that the user knows the concept of FILE-PROTECTION, so KNOME does not make any additional inferences about the user's level of expertise this time.

-
-
-

The processing here is standard: the goal analyzer determines that the user wants to know how to add write permission to a file, the domain planner is called and returns with the plan of using the chmod command, and UCEgo implements the plan of telling the user this information.

```
KNOME: Asserting *USER* does not know UNIX-CHMOD-COMMAND0
KNOME: Since UNIX-CHMOD-COMMAND0 is a UNIX-CHMOD-COMMAND,
       asserting *USER* does not know UNIX-CHMOD-COMMAND
KNOME: UNIX-CHMOD-COMMAND has difficulty COMPLEX, so deducing:
KNOME: *USER* is LIKELY to be BEGINNER
       => likelihood(*USER* = BEGINNER) = VERY-LIKELY
KNOME: *USER* is SOMEWHAT-LIKELY to be INTERMEDIATE
       => likelihood(*USER* = INTERMEDIATE) = SOMEWHAT-LIKELY
```

Based on the fact that the user does not know how to add write permission to a file, and hence does not know that the UNIX chmod command is a plan for doing this, KNOME can infer that the user is not familiar with the chmod command. This allows KNOME to infer that the user is likely to be a beginner and somewhat likely to be an intermediate.

```
Express:  now expressing the PLANFOR:
(PLANFOR330 &)

Express:  creating an example for the incomplete plan, CHMOD-FORMAT0

Express:  choosing a name, foo, for an example file.

Express:  selecting USER-PROT -- print name, u,
to fill in a parameter of the example.
```

UCExpress is passed PLANFOR330 to express. PLANFOR330 is an incomplete plan, because the user did not specify either the user type (either user, group, or other) to which chmod should add write permission or the name of the file argument. By consulting KNOME, UCExpress can infer that the user does not know the command format of chmod. In order to communicate the format to the user, UCExpress uses an *example*. In order to create an example, UCExpress must fill in the unspecified parts of the plan by choosing a name for the file and a user type for the protection. When and how to use specialized formats such as examples to inform the user is described in Chapter V, Section 3.

```
Express:  created the example(s):
((TELL10
   (speaker10-0 = UC)
   (listener10-0 = *USER*)
   (proposition10 =
     (EXAMPLE0
       (example0 = (PLANFOR330-0
                    (goals330-0 =
                     (CHANGE-PROT-FILE-EFFECT0-0? ...))
                    (plan330-0 = (TYPE-ACTION1
                                  (speaker1-4 = *USER*)
                                  (type-string1 =
                                   (CHMOD-FORMAT0-0 ...)))))))))))

Express:  not expressing CHANGE-PROT-FILE-EFFECT0?,
          since it is already in the context.
```

This concept corresponds to the phrase "add write permission to a file." Since UCExpress has pruned this concept (i. e. marked the concept as not needing generation), the tactical generator, upon checking that this concept is unambiguous (i. e. there are not other EFFECTs being generated), is able to omit the concept. This shortens "To add write permission to a file, use chmod" to just "Use chmod." Pruning by UCExpress is described in Chapter V, Section 2.

```
The generator is passed:
(TELL9 &)

The generator is passed:
(TELL10 &)
Use chmod.
For example, to add individual write permission to the file named
foo, type 'chmod u+w foo'.
```

In the next example, UCEgo takes the initiative in the conversion to volunteer information to the user.

```
# Is chin logged into the network?

The parser produces:
(ASK13 (asked-for13 =
         (QUESTION13
           (what-is13 =
             (STATE-OF-USER0?
               (user-state0 = (LOGGED-INTO-NETWORK1
                                (onto-network1 = NETWORK2)))
               (state-user0 = USER9)))))
       (listener13 = UC)
       (speaker13 = *USER*))
(HAS-USER-NAME2 (user-name2 = chin) (named-user2 = USER9))

  •
  •
  •

KNOME: Asserting *USER* does not know STATE-OF-USER0?

  •
  •
  •

UCEgo detects the following concepts:
(UC-HAS-GOAL70 &)
and asserts the following concept into the database:
(UC-is-state1 (is-state1 = (STATE-OF-USER0? &)))
```

UC-is-state is a procedure for determining state information for users. Since access to UNIX was not implemented for this version of UC, UC-is-state always claims that UC does not know the information.

```
UCEgo:  UC does not know STATE-OF-USER0?

UCEgo: detected the goal:
(UC-HAS-GOAL71 (goal71 = (KNOW75? (knower75 = UC)
                                  (fact75 = (STATE-OF-USER0? &)))))
from the situation:
(KNOW74 (knower74 = UC)
        (truth-val74 = FALSE)
        (fact74 = (STATE-OF-USER0? &)))
(UC-HAS-GOAL70 &)
```

Since UC has the goal of having the user know STATE-OF-USER0, and UC does not know this information, UCEgo adopts the *meta-goal* of UC knowing this information. Meta-goals are described in Chapter III, Section 3.

```
UCEgo detects the following concepts:
(UC-HAS-GOAL71 &)
and asserts the following concept into the database:
(UC-is-state2 (is-state2 = (STATE-OF-USER0? &)))
```

UC-is-state is called whenever UC has the goal of knowing state information for a user. Previously UC-is-state was called, since UC had the goal of having the user know. Now UC has the goal of having UC know, so UC-is-state is called again.

```
UCEgo: suggesting the plan:
(PLANFOR81 (plan81 = (APOLOGIZE2 (apology2 = (KNOW74 &))
                                 (listener2-3 = *USER*)
                                 (speaker2-3 = UC)))
           (goals81 = (BE-POLITE5 (polite-to5 = *USER*)
                                  (is-polite5 = UC))))
based on the situation:
(KNOW74 &)
(HAS-GOAL-ga3 &)
(UC-HAS-GOAL61 (status61 = ACTIVE) (goal61 = (BE-POLITE5 &)))
```

Since UC does not know what the user wants to know, and UC wants to be polite to the user, UCEgo suggests the plan of apologizing to the user for not knowing.

```
UCEgo detects the following concepts:
(UC-HAS-GOAL61 &)
(PLANFOR81 &)
```

```
and asserts the following concept into the database:
(UC-HAS-INTENTION17 (intention17 = (APOLOGIZE2 &))
                    (status17 = ACTIVE))

UCEgo detects the following concepts:
(UC-HAS-INTENTION17 &)
and asserts the following concept into the database:
(UCexpress6 (gen-prop6 = (APOLOGIZE2 &)))

Express:  not expressing STATE-OF-USER0?,
          since it is already in the context.

The generator is passed:
(APOLOGIZE2 &)
I'm sorry, I do not know that.
```

Since UCExpress marked STATE-OF-USER0 as not needing generation, the tactical generator is able to consider using the pronoun "that" in place of STATE-OF-USER0. Since UCExpress has not actually altered the conceptual network, but only marked STATE-OF-USER0 as not needing generation, the generator still has enough information to generate the full sentence, if the use of a pronoun were to be ambiguous.

At this point, UC still has the goal of knowing whether chin is logged into the network, although UCEgo does not know of any plans for achieving this goal. So UCEgo adopts the meta-goal (shown below) of finding out a plan for achieving this goal. This will eventually lead UC to find a plan for finding out whether chin is logged into the network and then to volunteer this information to the user.

```
UCEgo: do not know a single planfor the foreground goal:
(UC-HAS-GOAL71 &)
so adding the meta-goal:
(UC-HAS-GOAL72 (goal72 = (KNOW76? (knower76 = UC)
                                  (fact76 = ACTION16?))))
(PLANFOR82? (goals82 = (INFORMATION-EFFECT2?
                         (desired-info2 = (STATE-OF-USER0? &))
                         (informant2 = UC)))
            (plan82 = ACTION16?))

UCEgo detects the following concepts:
(PLANFOR82? &)
(UC-HAS-GOAL72 &)
and asserts the following concept into the database:
(UNIX-planner4 (user-goals4 = (INFORMATION-EFFECT2? &)))
```

UCEgo calls the domain planner to produce a plan for achieving UC's goal.

```
The planner is passed:
((INFORMATION-EFFECT2? &))

The planner produces:
(PLANFOR490 (goals490 = (INFORMATION-EFFECT2? &))
            (plan490 = (UNIX-RWHO-COMMAND0 (rwho-actor0 = UC))))
(RWHO-HAS-FORMAT0
  (RWHO-HAS-FORMAT-command0 = (UNIX-RWHO-COMMAND0 &))
  (RWHO-HAS-FORMAT-format0 =
    (RWHO-FORMAT0 (RWHO-FORMAT-step0 = rwho))))
(HAS-EFFECT180
  (command-of-effect180 = (UNIX-RWHO-COMMAND0 &))
  (effect-of-command180 = (LIST-ACTION110
                            (list-loc110 = TERMINAL3)
                            (list-objs110 = USER-LOGIN-TIME1?))))
(HAS-COMMAND-NAME200
  (HAS-COMMAND-NAME-named-obj200 = (UNIX-RWHO-COMMAND0 &))
  (HAS-COMMAND-NAME-name200 = rwho))
```

The domain planner returns the plan of using the rwho command to find out if chin is logged into the network.

```
UCEgo detects the following concepts:
(UC-HAS-GOAL66 &)
(PLANFOR450 &)
and asserts the following concept into the database:
(UC-HAS-INTENTION21 (intention21 = (UNIX-RWHO-COMMAND0 &))
                    (status21 = ACTIVE))
```

UCEgo would like to carry out the plan of using the rwho command, but cannot, since UC does not presently have a UNIX interface.

```
UCEgo: suggesting the plan:
(PLANFOR84
  (goals84 = (HELP5 &))
  (plan84 = (SATISFY12 (actor12 = UC)
                       (need12 = (KNOW69? (fact69 = (PLANFOR490 &))
                                          (knower69 = *USER*))))))

based on the situation:
(UC-HAS-GOAL63 &)
(PLANFOR490 &)
(HAS-GOAL-ga3 &)
```

Based on the fact that UC wants to help the user (UC-HAS-GOAL63), the user wants to know something (HAS-GOAL-ga3), there is a plan for finding this out (PLANFOR490), and the user does not know the plan (not shown), UC suggests that a plan (PLANFOR84) for helping the user is for UC to let the user know the plan. This is an example of when UCEgo decides to *volunteer* information to the user. The user did not specifically ask UC how to find out if chin is logged into the network; rather the user only asked UC whether chin is logged into the network. UCEgo takes this opportunity to teach the user something helpful. Note that UCEgo only decides to volunteer this information if it believes that the user does not already know the information (as modeled by KNOME). Different situations in which a system might want to volunteer information are described in Chapter III, Section 5.

•
•
•

```
To find out, type 'rwho'.
```

This followup query by the user shows the use of a simile format by UCExpress. The simile format takes advantage of the user's prior knowledge as modeled by KNOME in order to convey information to the user more succinctly.

```
# What exactly does rwho do?

The parser produces:
(ASK14 (listener14 = UC)
       (speaker14 = *USER*)
       (asked-for14 = (QUESTION14 (what-is14 = STATE14?))))
(HAS-EFFECT22? (effect-of-command22 = STATE14?)
               (command-of-effect22 = UNIX-RWHO-COMMAND1))
```

•
•
•

```
UCEgo:  trying to find effects for UNIX-RWHO-COMMAND1
       the effects are:
((HAS-EFFECT16-0
   (command-of-effect16-0 = (UNIX-RWHO-COMMAND1 &))
   (effect-of-command16-0 =
     (LIST-ACTION9-0
       (list-loc9-0 = TERMINAL3-0)
       (list-objs9-0 =
         (ALL8-0 (such-that8-0 =
```

```
                    (LOGGED-INTO-NETWORK0-0?
                      (network-user0-0 =
                        (ALL8-0 &))
                      (via-login0-0 =
                        (LOGGED-INTO-MACHINE7-0?
                          (logged-on-user7-0 = (ALL8-0 &))
                          (login-tty7-0 = TTY5-0?)))
                      (onto-network0-0 = NETWORK1-0)))
                  (all-type8-0 = USER8-0?))))))
(HAS-EFFECT17-0
  (command-of-effect17-0 = (UNIX-RWHO-COMMAND1 &))
  (effect-of-command17-0 =
    (LIST-ACTION10-0 (list-loc10-0 = TERMINAL3-0)
                     (list-objs10-0 = TTY5-0?))))
(HAS-EFFECT18-0
  (command-of-effect18-0 = (UNIX-RWHO-COMMAND1 &))
  (effect-of-command18-0 =
    (LIST-ACTION11-0 (list-objs11-0 = USER-LOGIN-TIME1-0?)
                     (list-loc11-0 = TERMINAL3-0)))))
```

```
UCExpress: Found a related command, so creating a comparison
between UNIX-RWHO-COMMAND1 and UNIX-WHO-COMMAND0
```

In describing the effects of a command, UCExpress checks to see if there is a related command in the command hierarchy which the user already knows (as modeled by KNOME) and which can be used to form a comparison with the command about which the user asked. If resultant simile is more succinct than simply listing the effects of the command directly, then UCExpress uses the simile to convey the knowledge to the user. Similes and other explanatory formats are described in Chapter V, Section 3.

```
KNOME: Asserting *USER* does not know HAS-EFFECT26
```

•
•
•

```
KNOME: Asserting *USER* does not know UNIX-RWHO-COMMAND1
KNOME: Since UNIX-RWHO-COMMAND1 is a UNIX-RWHO-COMMAND,
       asserting *USER* does not know UNIX-RWHO-COMMAND
KNOME: UNIX-RWHO-COMMAND has difficulty MUNDANE, so deducing:
KNOME: asserting *USER* = BEGINNER
```

The fact that the user does not know the rwho command, in conjunction with previous facts, allows UC to infer that the user is a beginner.

•
•
•

rwho is like who, except rwho is for all users on the network.

The last example, below, shows UC refusing to help the user delete someone else's files.

```
# How can I delete Chin's files?

The parser produces:
(ASK15
  (listener15 = UC)
  (speaker15 = *USER*)
  (asked-for15 = (QUESTION15
                   (what-is15 = (ACTION17? (actor17 = *USER*))))))
(DELETE-ACTION0?
  (del-effect0 = (DELETE-EFFECT0?
                   (DELETE-EFFECT-final-state0 =
                     (EXISTS0 (exist-object0 = FILE6)
                              (existence0 = FALSE)))
                   (DELETE-EFFECT-initial-state0 =
                     (EXISTS3 (exist-object3 = FILE6)
                              (existence3 = TRUE)))
                   (del-object0 = FILE6)))
  (actor0-3 = *USER*)
  (cause0-2 = (ACTION17? &)))))
(HAS-OWNER1 (owner1 = PERSON39) (owned-obj1 = FILE6))
(HAS-NAME27 (name27 = chin) (named-obj27 = PERSON39))

  •
  •
  •

UCEgo: suggesting the plan:
(PLANFOR89 (goals89 = (HELP5 &))
           (plan89 = (SATISFY14 (need14 = (KNOW-ga5? &))
                                (actor14-1 = UC))))
based on the situation:
(UC-HAS-GOAL63 &)
(HAS-GOAL-ga5 &)
```

As in normal situations, UCEgo suggests the plan of having the user know how to delete Chin's files in order to help the user.

•
•
•

```
UCEgo: suggesting the plan:
(PLANFOR90
  (goals90 = (ACT-ETHICALLY0 (actor0-0 = UC)))
  (plan90 = (SATISFY15 (actor15 = UC)
                       (need15 = (PREVENT5 (preventer5 = UC)
                                           (prevent-state5 =
                                             (KNOW-ga5? &)))))))

based on the situation:
(HAS-OWNER1 &)
(UC-HAS-GOAL62 (status62 = ACTIVE) (goal62 = (ACT-ETHICALLY0 &)))
(PLANFOR-ga8 (goals-ga8 = (DELETE-EFFECT0? &))
             (plan-ga8 = (ACTION17? &)))
(HAS-GOAL-ga5 &)
```

However, in this particular situation (in which someone wants to know how to delete a file owned by someone else and UC has the goal of acting ethically), UCEgo suggests the plan of preventing the user from knowing how in order to act ethically.

```
UCEgo detects the following concepts:
(UC-HAS-GOAL62 &)
(PLANFOR90 &)
and asserts the following concept into the database:
(UC-HAS-INTENTION23 (intention23 = (SATISFY15 &))
                    (status23 = ACTIVE))

UCEgo detects the following concepts:
(UC-HAS-GOAL63 &)
(PLANFOR89 &)
and asserts the following concept into the database:
(UC-HAS-INTENTION24 (intention24 = (SATISFY14 &))
                    (status24 = ACTIVE))
```

UCEgo adopts the intention of carrying out the two plans.

```
UCEgo: detected the goal:
(UC-HAS-GOAL75 (goal75 = (KNOW-ga5? &)))
from the situation:
(UC-HAS-INTENTION24 &)

UCEgo: detected the goal:
(UC-HAS-GOAL76 (goal76 = (PREVENT5 &)))
```

```
from the situation:
(UC-HAS-INTENTION23 &)
```

UCEgo adopts as sub-goals: the goal of having the user know, and the goal of preventing the user from knowing.

```
UCEgo detects the following concepts:
(PLANFOR-ga8 &)
(UC-HAS-GOAL75 &)
and asserts the following concept into the database:
(UNIX-planner5 (user-goals5 = (DELETE-EFFECT0? &)))

UCEgo detects the following concepts:
(UC-HAS-GOAL75 &)
(UC-HAS-GOAL76 &)
and asserts the following concept into the database:
(HAS-PRECEDENCE7
  (greater7 = (UC-HAS-GOAL71
                (goal71 =
                  (RESOLVE-GOAL-CONFLICT2
                    (conflict-goal-A2 = (UC-HAS-GOAL76 &))
                    (conflict-goal-B2 = (UC-HAS-GOAL75 &))))))
  (lesser7 = (UC-HAS-GOAL76 &)))
```

Since UC both wants something and wants to prevent it, UCEgo detects a goal conflict. So it adopts the meta-goal (UC-HAS-GOAL71) of resolving the conflict (RESOLVE-GOAL-CONFLICT2). The HAS-PRECEDENCE7 relation states that the goal of resolving the conflict should take precedence over one of the conflicting goals. The HAS-PRECEDENCE8 relation below states that resolving the conflict also has precedence over the other conflicting goal.

```
UCEgo detects the following concepts:
(UC-HAS-GOAL75 &)
(UC-HAS-GOAL76 &)
and asserts the following concept into the database:
(HAS-PRECEDENCE8 (lesser8 = (UC-HAS-GOAL75 &))
                 (greater8 = (UC-HAS-GOAL71 &)))

UCEgo: detected the goal:
(UC-HAS-GOAL77 &)
from the situation:
(UC-HAS-GOAL75 &)
(UC-HAS-GOAL76 &)

UCEgo: suggesting the plan:
```

```
(PLANFOR91 (goals91 = (RESOLVE-GOAL-CONFLICT2 &))
           (plan91 = (UC-resolve-conflict1
                       (goal-A1 = (UC-HAS-GOAL76 &))
                       (goal-B1 = (UC-HAS-GOAL75 &)))))
based on the situation:
(UC-HAS-GOAL77 &)
```

UC-resolve-conflict is a procedure which tries to resolve the goal conflict.

```
UCEgo: suggesting the plan:
(PLANFOR92
  (plan92 = (APOLOGIZE3
              (speaker3-3 = UC)
              (apology3 =
                (HAS-ABILITY1 (ability1 =
                                (TELL15 (speaker15-0 = UC)
                                        (listener15-0 = *USER*)))
                              (truth-val1-1 = FALSE)
                              (doer1 = UC)))
              (listener3-3 = *USER*)))
  (goals92 = (BE-POLITE5 &)))
based on the situation:
(ASK15 &)
(UC-HAS-GOAL61 &)
(UC-HAS-GOAL76 &)
```

Since UC does not want the user to know something which the user asked about, yet UC still wants to be polite to the user, UCEgo suggests the plan of apologizing to the user for not be able to tell the user.

```
UCEgo detects the following concepts:
(UC-HAS-GOAL61 &)
(PLANFOR92 &)
and asserts the following concept into the database:
(UC-HAS-INTENTION25 (intention25 = (APOLOGIZE3 &))
                    (status25 = ACTIVE))

UCEgo detects the following concepts:
(UC-HAS-GOAL71 &)
(PLANFOR91 &)
and asserts the following concept into the database:
(UC-HAS-INTENTION26 (intention26 = (UC-resolve-conflict1 &))
                    (status26 = ACTIVE))

UCEgo detects the following concepts:
```

```
(UC-HAS-INTENTION25 &)
and asserts the following concept into the database:
(UCexpress10 (gen-prop10 = (APOLOGIZE3 &)))

UCEgo detects the following concepts:
(UC-HAS-INTENTION26 &)
and asserts the following concept into the database:
(UC-resolve-conflict1 &)

UCEgo: detected conflicting goals:
(UC-HAS-GOAL76
  (goal76 =
    (PREVENT5 (preventer5 = UC)
              (prevent-state5 =
                (KNOW-ga5? (knower-ga5 = *USER*)
                           (fact-ga5 = (ACTION17?
                                         (actor17 = *USER*)))))))))

(UC-HAS-GOAL75
  (goal75 =
    (KNOW-ga5? (knower-ga5 = *USER*)
               (fact-ga5 = (ACTION17? (actor16 = *USER*))))))

UCEgo: goal, UC-HAS-GOAL76, has higher precedence,
so resolving goal conflict by discarding the goal, UC-HAS-GOAL75
```

The goal conflict is finally resolved by abandoning one of the conflicting goals. Resolving goal conflicts is described in Chapter IV, Section 2.5.

```
The planner is passed:
((DELETE-EFFECT0? &))

The planner produces:
(PLANFOR250
  (goals250 = (DELETE-EFFECT0? &))
  (plan250 = (UNIX-RM-COMMAND0 (rm-file0 = FILE6)
                               (UNIX-RM-COMMAND-effect0 =
                                 (DELETE-EFFECT0? &)))))
(HAS-FILE-NAME20 (named-file20 = FILE6)
                 (file-name20 = (lisp= nil)))
(RM-HAS-FORMAT0
  (RM-HAS-FORMAT-command0 = (UNIX-RM-COMMAND0 &))
  (RM-HAS-FORMAT-format0 =
    (RM-FORMAT0 (rm-file-arg0 =
                  (file-name20 = aspectual-of
                                 (HAS-FILE-NAME20 &)))
                (RM-FORMAT-step0 = rm))))
(HAS-COMMAND-NAME00
  (HAS-COMMAND-NAME-named-obj00 = (UNIX-RM-COMMAND0 &))
  (HAS-COMMAND-NAME-name00 = rm))
```

```
UCEgo detects the following concepts:
(ANSWER-FOR9 (answer9 = (PLANFOR250 &)) (query9 = (ACTION17? &)))
(KNOW83 (knower83 = *USER*)
        (fact83 = (ACTION17? &))
        (truth-val83 = FALSE))
and asserts the following concept into the database:
(user-knows20 (uk-fact20 = (PLANFOR250 &))
              (uk-user20 = *USER*)
              (uk-truth-val20 = FALSE))

KNOME: Asserting *USER* does not know PLANFOR250

UCEgo detects the following concepts:
(KNOW84 (knower84 = *USER*)
        (fact84 = (PLANFOR250 &))
        (truth-val84 = FALSE))
and asserts the following concept into the database:
(user-knows21 (uk-fact21 = (UNIX-RM-COMMAND0 &))
              (uk-user21 = *USER*)
              (uk-truth-val21 = FALSE))

KNOME: Asserting *USER* does not know UNIX-RM-COMMAND0
KNOME: Since UNIX-RM-COMMAND0 is a UNIX-RM-COMMAND,
       asserting *USER* does not know UNIX-RM-COMMAND

The generator is passed:
(APOLOGIZE3 &)
I'm sorry, I cannot tell you.

UCEgo: do not know a single planfor the foreground goal:
(UC-HAS-GOAL76 &)
so adding the meta-goal:
(UC-HAS-GOAL78 (goal78 = (KNOW87? (knower87 = UC)
                                  (fact87 = ACTION18?))))
(PLANFOR93? (goals93 = (PREVENT5 &)) (plan93 = ACTION18?))
```

Since UCEgo does not know how to prevent the user from knowing how to delete Chin's files, it adopts the meta-goal of finding out a plan for achieving that goal.

```
UCEgo detects the following concepts:
(PLANFOR93? &)
(UC-HAS-GOAL78 &)
and asserts the following concept into the database:
(UNIX-planner6 (user-goals6 = (PREVENT5 &)))

The planner is passed:
((PREVENT5 &))

The planner produces:
```

```
nil
```

The domain planner does not know of any plans either.

```
# Good-bye
```

•
•
•

```
Good-bye.
```

3.2. Other Approaches to Building Natural Language Consultants

At about the same time as UC, The UCC system ([Douglass and Hegner, 1982]) was built to answer questions about UNIX. UCC stored information about UNIX in a conventional data-base, and translated user queries into a data-base query language, which was then used to retrieve information about UNIX from the data-base. This work was continued in the Yucca-II system ([Hegner, 1988] and [McKevitt and Wilks, 1987]. Yucca-II is also designed as a data-base of UNIX facts with a separate natural language front end.

More recently, the AQUA system ([Quilici et al., 1985], [Quilici et al., 1986], and [Quilici et al., 1987]) uses the paradigm of *reminding* to recognize user problems as being similar to AQUA's experiences prestored in a dynamic episodic memory. Based on the similarities, AQUA can classify the user's problem and then construct different advice based on the type of problem.

In a similar vein, SUSI ([Jerrams-Smith, 1986]), USCSH ([Matthews and Biswas, 1986]), and the SINIX Consultant SC ([Kemke, 1987]), provide direct interfaces to UNIX (or SINIX). These systems concentrate on detecting user mistakes or inefficiencies as users interact with UNIX (or SINIX), and then either correcting the user's mistakes or suggesting more efficient plans. The PASSIVIST and ACTIVIST systems ([Fischer et al., 1985]) attack the same problem in the domain of using the BISY editor.

The EUROHELP project ([Breuker et al., 1987] and [Breuker, 1987]) has concentrated on coaching strategies for teaching the user once the system has detected a user problem from monitoring the user in action. A prototype EUROHELP system has been implemented for the domain of UNIX-Mail.

In the domain of the VAX/VMS operating system, the WIZARD system ([Shrager, 1981], [Shrager and Finin, 1982], and [Finin, 1983]) monitors users, and pops up a help window when it detects inefficient usage of VMS commands by the user.

There are many differences between these systems and UC, both in their approach to the domain and in their implementations. For example, the Yucca-II system concentrates more on complex domain planning and less on natural language understanding than UC. The AQUA system concentrates on a particular aspect of consultation in the UNIX domain, plan-oriented user misconceptions. The SUSI, USCSH, SC, ACTIVIST, and WIZARD systems concentrate on active help in which the systems look over the shoulder of the user. Finally, the EUROHELP project has concentrated on coaching strategies for teaching the user.

In terms of this thesis, the most important difference between these other systems and UC is that the other systems are not implemented as intelligent agents. Although some of these systems address some of the same issues as the intelligent agent part of UC, none of these systems address all of the issues. So, AQUA and many of the active help systems can volunteer advice and/or handle some user misconceptions, but none of these systems would ever refuse to help the user when it is inappropriate to do so. Of course, one can always put in ad-hoc rules to look for special cases, but such systems would have difficulty detecting interactions (both positive and negative) among rules. To address these issues in a principled fashion, a system should have its own goals, plans, and meta-plans to deal with interactions among its plans and goals. In short, a system should be structured as an intelligent agent.

Chapter II

KNOME: UCEgo's User Modeler

1. Introduction

An intelligent agent needs to have a model of its environment in order to act intelligently. In the case of an intelligent agent that plays the role of a consultant, a large part of the agent's environment is the client of the consultant. Since the main job of a consultant is to impart knowledge to its client, a consultant system that is an intelligent agent needs to have a model of its user's knowledge and beliefs. In particular, an intelligent consultant system needs a user model in order to detect situations in which the user's knowledge is incorrect (misconceptions) and situations in which the user lacks needed information. In such case the intelligent consultant should take the initiative to correct the user's misconception or inform the user of the needed information, respectively.

Besides being useful in volunteering information and correcting misconceptions, a model of the user's knowledge allows a natural language interface to:

1) avoid telling the user something that the user already knows
2) tailor explanations for the user's level of understanding
3) utilize the user's background knowledge in interpreting what the user says

The KNOME (KNOwledge Model of Expertise) component provides UC with a model of what its users know. KNOME infers specific facts about what a user does or does not know from a dialog with the user, and combines this evidence to infer the user's level of expertise. Then, KNOME uses this judgment of the user's expertise to *predict* the likelihood that the user might know other facts. KNOME's model of the user's knowledge is made available to other components of UC, including UCEgo, UCExpress, and PAGAN (the goal analyzer). These components utilize KNOME's model of the user's knowledge to respond more intelligently to the user.

1.1. Some UC Examples

To see how KNOME can infer a user's level of expertise from a conversation with the user, and how this information can be used advantageously by UC in helping the user, consider the two UC sessions shown in Figures 2.1 and Figure 2.2.

```
Welcome to UC (Unix Consultant) version 3.23
To a '#' prompt, please type in your questions
about the UNIX file system in English.
To leave, just type '^D' or '(exit)'.

Hi.
How can I help you?

# How can I delete a file?

Use rm.
For example, to delete the file named foo, type `rm foo'.

# What does ruptime do?

ruptime is used to list the uptime of all machines on the network,
list the number of all users on them, and list their load average.
```

Figure 2.1. UC session with User1.

```
Welcome to UC (Unix Consultant) version 3.23
To a '#' prompt, please type in your questions
about the UNIX file system in English.
To leave, just type '^D' or '(exit)'.

Hi.
How can I help you?

# How can I find out the inode of a file?

Use ls -i.

# What does ruptime do?

ruptime is like uptime, except ruptime is for all machines on the
network.
```

Figure 2.2. UC session with user2.

The user of the first session, User1, is apparently a novice who knows very little about UNIX. After all, User1 does not even know something as simple as how to delete a file (the subject of User1's first question). On the other hand, the user in the second session, User2, knows what an inode is, a relatively complex UNIX concept. So User2 is probably a much more advanced user of UNIX than User1. KNOME uses this type of evidence about particular facts that users know or do not know in order to deduce their level of expertise.

Based on the different levels of expertise of the two users as deduced by KNOME, UC is able to provide qualitatively different answers to the two users. In the first answer to User1, KNOME gives an example of the format of the rm command (simply "rm" followed by the name of the file to be deleted). UC gives an example of the format to User1, since a user as unsophisticated as User1 probably would not know the format of rm. On the other hand, User2 is a more advanced user who undoubtedly already knows the format of ls. Hence UC is able to give a very concise answer to User2's first query, "Use ls -i," that does not include an example of the format of ls. This illustrates how KNOME is used to avoid telling the user something that the user already knows.

The second query of both users is the same. For the more advanced user, UC was able to explain the ruptime command in terms of a similar command, uptime, that the more advanced user presumably already knows. Such a simile would not work for User1, since a novice user probably would not know the uptime command, and so could not exploit the simile to understand ruptime. KNOME's modeling of the user's knowledge allows UC to choose whether or not to use different forms of presentation like similes based on whether or not the user will understand them.

1.2. Problems in Modeling

There are many difficulties in building and maintaining a model of what the user knows. The system must first determine what the user knows, and then represent this information internally. To determine the user's knowledge of the domain, the system could quiz the user exhaustively. However for most domains, such a quiz would be extremely time consuming and most users would not consent to such extensive quizzing. Fortunately, users tend to learn domain knowledge in a predictable order, so the system can make additional inferences about what a user is likely to know or not know based on a partial user model. That is, the user modeling system should be able to *predict* a user's knowledge based on partial information about what the user knows or does not know. Exactly which predictions/inferences are allowable is problematic. This is the key problem for any user modeling system that wishes to take advantage of the order in which users typically gain knowledge in a domain.

Besides the basic question of which inferences are allowable, there is also the problem of uncertainty. Making inferences about the user based on limited prior knowledge about the user is a form of default reasoning. Such default reasoning is not always valid, so the system needs to be able to handle the fact that such inferences are uncertain. Usually this requires that the user modeler be able to judge the certainty of its default inferences. Also, as the system obtains more concrete information, it must be able to update its user model. The new information may contradict previous predictions, so

maintenance is not simple.

Another problem concerns the organization of the inferences. Many inferences tend to be grouped together, because people tend to go through distinct stages of learning (see [Kay and Black, 1985]). People at a particular stage of learning tend to have similar knowledge. Once a system has determined its user's level, it can assume that this user knows about as much as other users at that level. A good user modeling system needs to group such inferences together in a simple and efficient manner.

Another problem in user modeling is how to represent the model internally. The simplest method is the *overlay* model ([Carr and Goldstein, 1977]) wherein the user's knowledge is represented as a subset of the system's knowledge. However, a simple overlay model cannot predict what a user might know based on partial information. An overlay model does not represent the order in which users typically learn information in a domain. What is needed is a model that represents such ordering information in a manner useful for prediction of what a user might know based on partial information.

1.3. Other User Models

User models have appeared frequently in Intelligent Tutoring Systems (e. g. [Burton and Brown, 1976], [Sleeman and Brown, 1982]). [Kass, 1987] provides a survey of user models in ITS. Typically, ITS use some variant of the simple overlay model ([Carr and Goldstein, 1977]). Overlay models encode the user's knowledge as a subset of the system's expert knowledge base. One variant is to encode users as a collection of differences from the system's built-in expert. Differences may include a lack of user knowledge relative to an expert. This is frequently augmented by a library of buggy procedures/rules that are common among users (e. g. [Brown and Burton, 1978], [Stevens et al., 1979], [Sleeman and Smith, 1981], [Johnson and Soloway, 1984], [Anderson et al., 1985], and [Reiser et al., 1985]). These types of user models are not designed to *predict* a user's knowledge based on partial information about what the user knows or does not know. This sort of default inference is not so appropriate for ITS where the system needs to keep an accurate user model. On the other hand, consultation systems usually do not interact with a single user for as much time as tutoring systems, so they do not have the time to build a complete user model. In such cases, default reasoning is needed. Therefore, consultation systems need a different type of user model than is commonly found in ITS.

An exception is the hierarchical overlay model of USCSH ([Matthews and Biswas, 1986]) which does some limited prediction. USCSH predicts the user's proficiency for parent nodes in the hierarchy by taking a weighted sum of the proficiency ratings of the children nodes. This approach allows prediction of highly related topics, but does not extend to prediction of the user's knowledge of less related topics. For example, such a system could predict that a user has high proficiency in process monitoring when the system knows that the user has high proficiency in using the `ps`, `who` and `kill` commands. However such a system could not predict the proficiency of the user in an unrelated topic, such as using the `cat` command, based on the system's knowledge of the user's proficiency in process monitoring commands.

Other user models have been built to model a number of aspects of the user (see [Wahlster and Kobsa, 1986] for a survey). For example, Grundy ([Rich, 1979], [Rich, 1983], and [Rich, 1987]) modeled user personality traits. Grundy used user *stereotypes* to group default inferences about user traits, then used these traits for suggesting books to the user for reading. To determine which stereotypes were applicable to a particular user, Grundy asked the user for a self description at the start of a session. Individual phrases (e. g. self descriptions such as athletic or feminist) would *trigger* stereotypes that were likely to apply to that user. Such an approach worked well for modeling personality traits, but does not carry over to modeling domain knowledge. A user's self description as a "beginner" often will not coincide with the system's idea of a "beginner." Also, stereotype triggers cannot be used to make inferences such as inferring that the user does not know something when the user asks about it. Stereotype triggers are not sufficient for inferring what users know.

Another use of user models is for representing user preferences. The hotel reservation system of HAM-ANS ([Hoeppner et al., 1984] and [Morik, 1987]) and the Real Estate Agent ([Morik and Rollinger, 1985]) chose different hotel rooms or apartments based on users' needs and preferences. These systems expect users to mention at the start of their session their requirements and preferences for a room/apartment. Such an approach works well in domains where users tend to specify their needs and preferences at the start of a session. However this is not the case in consultation domains where users are unlikely to mention their level of expertise in the domain at the start of a session.

Many systems including UC have modeled the user's plans and goals (e. g. [Allen and Perrault, 1980], [Carberry, 1983] and [Carberry, 1987], [Litman and Allen, 1984], [Johnson and Soloway, 1984], [Wilensky, 1986]). Such systems infer the user's plans and goals by matching the user's actions to steps of prestored plans. This type of inference works well for detecting the user's plans, but does not extend to determining what the user does or does not know.

2. Internal Representation of Users

KNOME represents what UC believes users know about UNIX. In this sense, KNOME's model of the user is not necessarily meant to accurately model actual users, but is meant to conform to the way human UNIX consultants model their clients. So in developing KNOME's user models, no attempt was made to psychologically profile what different users know. Instead, former UNIX consultants (UC's implementors) were informally surveyed to determine how they viewed users.

2.1. Double-Stereotypes

Humans use categorization to organize inferences (see [Rosch, 1977], [Rosch, 1978], and [Rosch, 1983]). These categories serve as reference points or prototypes for judging objects. Once an object has been identified as belonging to a particular category, default inferences about the object can be made based on the object's membership in the category.

An approach similar to human categorization is used in KNOME to organize inferences concerning users. In KNOME, users are grouped according to their level of expertise in using UNIX. Each group is represented by a prototype-style category. There are four such categories in UC: *novice, beginner, intermediate,* and *expert.* Each of the categories represents an increasing mastery of UNIX information.

Individual users are represented as members of one of these categories and inherit properties from their category. Specific information about individual users overrides inherited information. These user categories are also termed stereotypes after the stereotypes of Grundy ([Rich, 1979], [Rich, 1983], and [Rich, 1987]), which pioneered categorization of users in computer programs. However, unlike the stereotypes in Grundy, which consist of collections of attributes, the stereotypes in KNOME are implemented as categories within the KODIAK representation system (see Appendix A). As such, KNOME's stereotypes are integrated into KODIAK's multiple inheritance hierarchy and its default inheritance mechanisms.

Besides categorizing users, KNOME also categorizes information into levels of difficulty. Commands, command-formats, terminology, and other relevant information are categorized according to their level of difficulty. These stereotype categories include *simple, mundane,* and *complex.* Information is grouped into categories based on their typical location on the learning curve, in other words, based on when the typical user would learn the information. For example, some simple concepts include the rm, ls, and cat commands, the technical term "file," and the simple file command-format (the name of the command followed by the name of the file to be operated upon). These simple concepts are learned early in the experience of a typical user. Somewhat more advanced are mundane level concepts. Some examples are the vi, diff and spell commands, the technical term "working directory," and the -l option of ls. Examples of complex concepts are the grep, chmod, and tset commands, the term "inode," and the fact that write permission on the containing directory is a precondition for using the rm command for deleting a file. Complex concepts are learned relatively late by the typical user.

In addition to the three previously mentioned levels of difficulty, KNOME has the category *esoteric* that includes information that is not typically learned at any particular stage of experience. Such concepts are learned only when users have special needs. Some may be familiar to beginners, yet unfamiliar to many experts. An example of an esoteric command is spice, a program used for electronic circuit simulation. Only people who need to perform circuit simulations would know how to use spice. This group would likely include beginning users as well as intermediate and expert users. On the other hand, there are many expert UNIX users who do not know about spice, because they have never needed to perform semiconductor circuit simulations.

Once information has been classified into levels of difficulty, inferences based on user stereotypes can easily be represented as relations between the various user stereotypes and difficulty levels. For example, Figure 2.3 shows a graphical KODIAK representation of one such relation. This Figure shows the KODIAK representation of the fact that intermediate users know most facts that have the difficulty level of mundane. The predicate of MOST is used, because there is uncertainty as to whether any particular intermediate user will know any particular mundane level fact. However, any intermediate level user will know most mundane level facts. Table 2.1 summarizes the relation

between different user stereotypes and different knowledge difficulty levels.

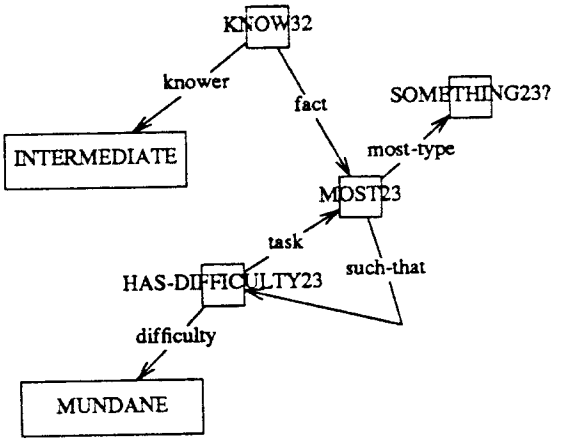

Figure 2.3. KODIAK representation of: intermediates know most mundane facts.

User Stereotype	Knowledge Difficulty Level			
	simple	mundane	complex	esoteric
expert	ALL	ALL	MOST	—
intermediate	ALL	MOST	AFEW	—
beginner	MOST	AFEW	NONE	—
novice	AFEW	NONE	NONE	NONE

Table 2.1. Relation between user stereotypes and knowledge difficulty levels.

The use of stereotype categories for both users and information is an example of a *double-stereotype* system. Single stereotype systems like Grundy used stereotypes for users (e. g., FEMINIST, INTELLECTUAL, SPORTS-PERSON) but not for information (i. e., Grundy did not have stereotypes for books such as GOTHIC-ROMANCE, DETECTIVE-STORY, or SCIENCE-FICTION). KNOME is the first user modeling system to utilize double-stereotypes.

The use of stereotypes allows KNOME to *predict* what users are likely to know or not know based on a partial user model. Once KNOME has accumulated a small number of facts (typically ~3)[2] about what users do or do not know during the course of a UC session, it can infer the user's level of expertise (see Section 3). Then based on the user's level of expertise, KNOME can predict whether or not the user is likely to know other facts. The small set of facts that allows KNOME to infer the user's level of expertise constitutes a partial user model, and the inference that the user belongs to a particular

[2] The actual number varies, because facts may mutually reinforce KNOME's evaluation of the user's level of expertise, in which case fewer facts will be needed; or facts may provide contradictory evidence, in which case more facts will be needed to evaluate the user's level of expertise.

stereotype represents a set of default inferences that are made on the basis of the partial user model.

The use of stereotypes allows KNOME to efficiently group these default inferences together. A system that did not use stereotypes would need to represent individually every partial user model and all the default inferences from that partial user model. Since a partial user model is simply a set of facts about a user, there is a combinatorically large number of partial user models. So, representing all such partial user models and their default inferences would be extremely inefficient. However many partial user models have similar default inferences. These can be grouped together efficiently using user stereotypes. A double-stereotype system is even more efficient, since it allows the system to encode for each fact only its level of difficulty rather than having to encode its relationship to every user stereotype. The rest of a double-stereotype system can then be encoded in a small number of statements relating the two levels of stereotypes. This relationship in KNOME between user stereotypes and knowledge difficulty levels is summarized in Table 2.1.

KNOME currently has only one double-stereotype range for users. This range represents user's knowledge of UNIX file manipulation commands, UNIX information gathering commands, and UNIX concepts and terminology. This corresponds to UC's domain of expertise within UNIX. If UC were extended to cover more of UNIX or even other operating systems, then KNOME would need more ranges of double-stereotypes. For example, if UC also covered usage of the vi and emacs editors, then KNOME would need separate expertise ranges for each of these domains. KNOME would also need to encode how the user's level of expertise in one domain might be used to predict the user's level of expertise in another domain. So, an expert in using vi would undoubtedly have a high level of expertise in manipulating the UNIX file system. On the other hand, a high level of expertise in using emacs does not necessarily indicate a high level of expertise in using UNIX, since the emacs editor is commonly found in other operating systems.

2.2. Modeling Individual Users

Individual users usually do not completely fit any one stereotype. As a result, a user model that uses stereotypes must also encode how individual users differ from the stereotypes. KNOME stores specific information about what individual users know and do not know. Such information is stored as a collection of propositions about what a user knows or does not know. These propositions are represented using KODIAK. In reasoning about what a user knows, this collection of individual facts is checked before resorting to reasoning from the user's stereotype category.

To avoid storing too many individual facts about what users know, only those facts that cannot be inferred with high likelihood from a user's category are stored. For example, the fact that a particular intermediate user knows the rm command (a simple command) does not need to be stored explicitly, since it is directly inferable from the fact that intermediate users know *all* simple facts. Also, the fact that a particular beginner knows the rm command does not need to be stored. This is because beginners know *most* simple facts and hence it is likely that any particular beginner would know rm. On the other

hand, the fact that a particular novice knows the rm command does need to be stored explicitly, since that is not inferable with high likelihood. Because novice users know just *a few* simple facts, it is unlikely that any particular novice would know rm.

2.3. Types of Knowing

In UC, two senses of "know" are used. The first is the classical sense of know: the knower believes that some proposition P is true, and P is true. An example of this sense of knowing is: " the user knows that the cd command is used to change the current working directory." There is also another sense of know that is found in UC. This sense of know is closer to the colloquial usage wherein knowing an object means being familiar with that object. However, in UC this vague usage is made precise. For UC, knowing a command such as the UNIX-RM-COMMAND implies knowing that there exists a command with the name rm, knowing the main purposes of this command, knowing the format of the command, knowing the main effects of the command, and knowing the main preconditions of the command.

The main purposes of a command are represented in UC using the *planfor* ([Chin, 1983a] and [Wilensky et al., 1986]) relation, which relates a goal and the plan that is typically used to satisfy that goal. The planfors of a plan are different from the effects of the plan, since not all of a plan's effects can be considered goals that the plan is typically used to satisfy. All goals in the planfors of a command are also effects of the command, but some side effects of the command may not be part of its planfors. For instance, calling the game program rogue with a file argument has a side effect that is not found in the planfors of rogue. When rogue is given a file argument that is not a rogue-format save file, then rogue arbitrarily removes the file. This is a side effect of the rogue program, but it is not in any planfors, since users do not typically run rogue in order to remove files.

2.4. Dealing with Uncertainty

In any user model, the inferences that the model makes about the user contains some degree of uncertainty. In KNOME, an attempt was made to avoid the use of numeric representations of uncertainty, because their use tends to lead systems to overestimate the accuracy of their ratings of uncertainty. So, UC uses a fixed number of simple rating levels instead: LIKELY, UNLIKELY, VERY-LIKELY, VERY-UNLIKELY, SOMEWHAT-LIKELY, SOMEWHAT-UNLIKELY, TRUE, FALSE, and UNCERTAIN (c. f. fuzzy logic [Zadeh, 1965]). KNOME also uses predicates such as AFEW, MOST, ALL, and NONE (c. f. [Zadeh,1982]). Such predicates and ratings are used by KNOME to express the certainty of its inferences. For example, when KNOME is asked whether an particular user knows some particular fact, it will return either TRUE, LIKELY, UNLIKELY, FALSE, or UNCERTAIN. TRUE and FALSE indicate that KNOME has complete confidence. LIKELY and UNLIKELY indicate some uncertainty. An UNCERTAIN result indicates that KNOME does not have enough information to guess whether the user does or does not know that fact.

To deduce whether a particular user knows some particular fact, KNOME uses a multi-step inference process. First, KNOME checks the individual user model for that user. If the individual user model records that this user does or does not know this fact, then the answer is, respectively, TRUE or FALSE. If there is no specific information in the user's individual user model, KNOME next checks the stereotype category of the user. If users of the user's stereotype do or do not know that fact, then the answer is again TRUE or FALSE, respectively. If these checks fail, KNOME resorts to inference based on the difficulty level of the fact. If the user's stereotype knows ALL facts of that difficulty level, then the answer is TRUE; if the stereotype knows MOST facts of that difficulty then, the answer is LIKELY; and if the stereotype knows AFEW facts of that difficulty, then the answer is UNLIKELY. Finally, if this process fails due to lack of information, then the answer is UNCERTAIN. This algorithm is summarized in the flow chart of Figure 2.4.

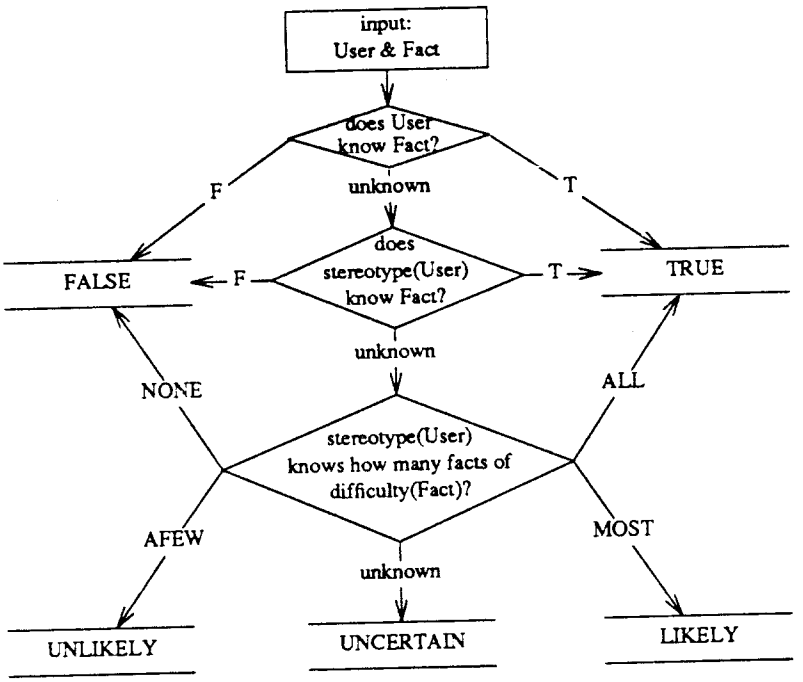

Figure 2.4. Algorithm for determining whether User knows Fact.

3. Deducing the User's Level of Expertise

One of the main problems in user modeling is how to build up a model of a particular user. In a user modeling system based on stereotypes, this means determining which stereotype(s) best fit the current user and how that user differs from the stereotypes. In UC, this means that KNOME must determine the user's level of expertise in using UNIX.

3.1. Other Model Acquisition Systems

Various approaches to building up models of users have been used. Some systems such as the Real Estate Agent ([Morik and Rollinger, 1985]) expect users to mention enough self-characteristics at the start of a session so that the system can build up the user model. This works for situations such as a real estate agent interchange with a client where the client is likely to mention what he/she needs/wants at the beginning of a dialog. However, this does not extend to other consultation situations where the user usually will not mention what he/she knows at the start of a session. In such consultation situations the user typically begins by describing his/her problem to the consultant.

Another approach is to ask the user for a self-description at the start of a session as in Grundy [Rich, 1979]). This is a reasonable approach for judging the user's personality traits, but does not work well for determining what the user knows. For judging the user's level of expertise, this approach is inaccurate, since the user's concept of different levels of expertise like "intermediate" may differ considerably from the system's concept or from other users' concept of an "intermediate." Also, this requirement is not very practical for occasional users, since this increases the overhead of using the system.

A third approach is to quiz users at the start of a session so that the system may estimate the user's expertise. For example, the MAP system ([Macmillan, 1983]) helps a user to specify a correct user model of how that user interacts with an operating system. Such an approach is extremely time consuming. For tasks like intelligent operating system interfaces for which MAP was designed, such a protracted process is somewhat more acceptable, since the user will realize its benefits over a long period of time. However for interacting with a consultation system like UC where typical sessions are short, such a process would be too time consuming. The casual user that only needed the answer to a single question would be better off without a user model rather than having to put up with such a lengthy process.

The most common approach in ITS (Intelligent Tutoring Systems) is to compare the user's performance with what a built-in expert would do. Differences in performance can then be attributed to either a lack of user knowledge or differences in user knowledge such as improper procedures or rules. However such an approach is only applicable if the system can "look over the shoulder" of the user. Consultation systems often cannot observe the user at work. Even when this is possible, consultation systems should not be expected to constantly observe the user, since human consultants do not do so.

To properly emulate a human consultant, the system should be able to assess the expertise level of the user while talking to the user. None of the previously mentioned approaches model this ability, which is a difficult problem in natural language understanding. One system that has addressed this problem is the VIE-DPM user modeling system ([Kobsa, 1985a]). However, VIE-DPM uses only a syntactic interpretation of the user's statements to derive the user's beliefs. VIE-DPM makes assumptions about the user's beliefs based on the form of the user's input (a yes/no question, a wh-question, or a command) ([Kobsa, 1985a] and [Kobsa, 1986]) or the presence of linguistic particles (e. g. "not") in the user's input ([Kobsa, 1985b]). Such a syntax-only system would not be able to properly interpret phenomena such as indirect speech acts ([Austin, 1962], [Searle, 1975]). For example, "Can you pass the salt?" is a yes/no question, but one

should not assume from this question that the speaker does not know whether the listener can pass the salt. Despite such limitations, VIE-DPM is a step in the right direction.

3.2. KNOME's Approach

KNOME uses the approach of inferring the user's level during the course of a session. At the beginning of a session, UC has no information about the user. In such cases, KNOME remains open to the possibility that the user may belong to any stereotype. KNOME maintains a list of candidate stereotypes to which the user may belong. KNOME also rates the likelihood that the user may be a member of any particular stereotype. At the start, the likelihood ratings for membership in any stereotypes is neutral, except for the beginner stereotype. The likelihood rating for beginner is initially slightly higher than neutral, since more of UC's users tend to be beginners rather than novices, intermediates, or experts. As more information is gathered about the user during the session, the stereotypes may rise or fall in likelihood.

KNOME deduces the user's level of expertise using a two phase process. First, during the course of the conversation, KNOME deduces individual facts about what the user does or does not know. As such facts are gathered, KNOME combines the evidence to adjust the likelihood ratings for the different levels of expertise. Eventually KNOME identifies the stereotype level of expertise to which the user belongs.

3.3. Collecting Evidence

The first step in the process of deducing the user's level of expertise is to collect evidence on that subject. Typically, such evidence takes the form of individual facts about what the user knows or does not know. Such facts and other forms of evidence can be deduced from what the user says (or does not say) and from an analysis of the user's goals. KNOME distinguishes six classes of inferences that were found to be useful in UC for determining the user's level of expertise. These inference classes are summarized in Table 2.2, which lists the inference class and the kinds of inferences that are commonly found in each inference class.

Inference Class	Kinds of Inferences
Claim	user states that user does (not) know ?x → user does (not) know ?x
Goal	user wants to know ?x → user does not know ?x
Usage	user uses ?x → user knows ?x
Background	user mentions his/her background → user knows as much as the stereotype indicated by the background
Query-Reply	system asks if user knows, user replies → user's reply
No-Clarify	system uses new terminology, user does not ask for clarification → user understands terminology

Table 2.2. Taxonomy of deduction types for inferring what users know.

The first class of inferences, Claim inferences, applies when the user claims to know or not know something. The second class, Goal inferences, applies when the system can infer that the user wants to know something. In such cases, the system can also infer that the user must not know whatever is wanted, since if the user already knew it, then the user would not have the goal of knowing it. The third class, Usage inferences, are made when the user properly uses some command or terminology. KNOME can then deduce that the user must know the command or terminology. The fourth class, Background inferences, are made when the user mentions some relevant information about his/her background. The fifth class of inference, Query-Reply, applies when the system queries the user about the user's knowledge and the user replies. The last type of deduction, No-Clarify, is made when the system uses some new terminology that the user may or may not know. If the user does not ask for a clarification of the terminology, then the system can assume that the user probably already knew the terminology. Each of these classes of inference is described in greater detail below.

Claim inferences are concerned with direct statements by the user about what the user does or does not know. Examples of such statements and the inferences that KNOME makes based on the statements include:

I already know about mkdir.
→ the user knows the mkdir command

I don't know how to move a file.
→ the user does not know how to move a file
→ the user does not know that mv is a plan for moving files
→ the user does not know the mv command

In the first example, the user claims to already know the mkdir command. This statement is not as straightforward as it might seem, since the user and the system may not agree on what it means to know a command. A reasonable interpretation of what a user might mean is that the user knows how to use mkdir, knows the main purpose of mkdir and knows the main effects and preconditions of mkdir. This interpretation is the same as what UC means when it encodes that a user knows a command (see Section 2.3).

Of course, it may be that the system should not believe the user's statement about knowing mkdir. It is possible that the user may be confused and does not actually know mkdir. But, there is always doubt about what someone might really know, no matter how strong the evidence. For example, even if the system were to watch the user use the mkdir command successfully, it is possible that the user may have actually wanted to create a file instead of a directory. However, in none of these cases is there any reason to doubt the user. Unless KNOME obtains evidence to the contrary, it assumes that the user really does (or does not) know something when the user makes such a claim.

The second example requires somewhat more complex processing than the first example. In the second example, the user claims not to know how to move a file. Since UC knows that mv is used to move files, KNOME is able to deduce that the user does not know that mv is a plan for moving files. Finally, because the user does not know one of the main purposes of the mv command, KNOME deduces that the user must not know mv.

KNOME only stores information about levels of difficulty for objects and not for descriptions, so only facts about a user knowing (or not knowing) specific objects (e. g. commands, concepts, etc.) can be used in deducing the user's level of expertise. This approach eliminates redundancy, since any object will have many possible descriptions. For example, the user could have referred to mv by "how to rename a file" rather than "how to move a file." So, the fact that the user does not know mv is useful for deducing the user's level of expertise, but the descriptive fact that the user does not know how to move a file cannot be directly used. UC must first determine that this description applies to the mv command before it can be used for deducing the user's level of expertise. In UC, descriptions of the purpose of commands are interpreted by UC's UNIX planner component. After UC's UNIX planner has determined the command referred to by the command description, KNOME can use this information in deducing the user's level of expertise.

Goal inferences apply when UC can deduce the user's goals. If UC can infer that the user wants to know something, ?x, then KNOME can deduce that the user does not

know ?x and also does not know the answer to ?x. For example, if the user asks "How can I delete a file?" then UC can deduce that the user's goal is to know how to delete a file. From this goal, KNOME can deduce that the user does not know how to delete a file. Then, after UC computes that a plan for deleting a file is to use the rm command, KNOME can infer that the user does not know this planfor relation between deleting files and the rm command. Finally, since the planfor relation is central to UNIX commands, KNOME can deduce that the user does not know rm in the sense that the user is not familiar with rm (see Section 2.3).

In UC, the initial deduction about the user's goal is made by UC's goal analysis component, PAGAN ([Wilensky et al., 1986] and [Mayfield, forthcoming]). PAGAN computes the user's plans and goals from the dialogue, and can handle phenomena such as indirect speech acts. In some cases, PAGAN utilizes KNOME's model of the user's knowledge in deducing the user's goals. For example, if the user asks, "Do you know how to compact a file?", then the choice between the direct interpretation (the user wants to know if UC knows how to compact a file) and the indirect interpretation (the user wants to know how to compact a file) depends partly on whether or not KNOME believes the user already knows the compress command. If KNOME does believe that the user is likely to know compress, then that will bias PAGAN toward the direct interpretation that the user is testing UC (not uncommon), and vice-versa.

This process may seem circular, since PAGAN's analysis of the user's goal depends on KNOME, while KNOME's analysis of the user's knowledge depends on PAGAN. However, KNOME's initial evaluation of whether the user knows is only a prediction from a partial user model. This prediction does not determine PAGAN's interpretation, because PAGAN must also take into account other factors in its analysis. One such factor is the frequency of usage of the direct versus indirect interpretations. PAGAN is biased toward adopting the more frequent usage. Another factor is the the context of the dialog. When UC is configured in knowledge acquisition mode as UCTeacher ([Martin, 1985]), then the direct interpretation (i. e. the user is testing UC's knowledge), is inherently more likely than the indirect interpretation. KNOME's beliefs about the state of the user's knowledge is only one factor in PAGAN's analysis, which then leads KNOME to its own inference about the state of the user's knowledge.

Besides questions on how to do something, Goal deductions apply to many other kinds of queries such as:

What is a directory?
→ the user wants to know what is a directory (goal)
→ the user does not know what is a directory
→ the user does not know the directory concept

Can you tell me how to find out who is on the system?
→ the user wants to know who is on the system (goal)
→ the user does not know who is on the system

What does ruptime do?
→ the user wants to know the effects of the ruptime command (goal)
→ the user does not know the effects of the ruptime command
→ the user does not know the ruptime command

All of the above queries represent Goal deductions, since PAGAN can deduce that the user wants to know some information. Based on this initial deduction, KNOME can infer that the user does not know the information that is sought.

Usage deductions are made when the user applies some plan/procedure correctly, or uses some specialized terminology correctly. Examples include:

I tried typing "rm foo," but I got the message, "rm: foo not removed."
→ the user knows the rm command

How can I find out the inode of a file?
→ the user knows the inode concept

I tried typing "del foo", but I got the message "del: Command not found."
→ the user knows the DOS del command
→ the user does not know the UNIX rm command

In the first example, KNOME can deduce that the user knows the rm plan. In the second example, KNOME deduces that the user knows the meaning of the term "inode." This is the main type of deduction used in ITS in which the system observes the user at work solving problems. This type of deduction is suggested by [Rich, 1983] for consulting situations.

In general, whenever the user uses specialized terminology correctly, KNOME infers that the user must know the concept denoted by the terminology. In UC, terms include UNIX command names and operating system terminology such as "file protection", "directory," and "inode." Checking that the user used the terms *correctly* is not very sophisticated in KNOME. The first check is simply that the term must be used in an English sentence that UC can parse and understand. However, the fact that the user used a term in an understandable sentence does not automatically mean that the user knows

the meaning of the term. For example, if the user asks, "What does rm do?", KNOME should not infer that the user knows the rm command. The same principle applies when the user states, "I don't know the rm command." In these types of sentences, KNOME actually first infers that the user does know rm and then reverses its inference later based on the fact that the user has the goal of knowing the effects of rm, which is a Goal type inference. In cases when KNOME has evidence both that the user knows something and that does not know it, KNOME always assumes that the user doesn't know, since it is always possible that the user just seems to know and in fact doesn't know.

The last example is somewhat more complex than the previous examples. In this case, KNOME first infers that the user knows the DOS del command. Then, since the DOS del command is used to delete files and the user mistakenly tried to use the del command, KNOME can deduce that the user does not know the UNIX command for deleting files, that is, the rm command. In general, whenever the user tries to use a foreign command in UNIX, then KNOME assumes that the user does not know the corresponding UNIX command. This correspondence is found by looking at the goal of the foreign command and finding a UNIX command that is a plan for the same goal.

Background deductions are made when the system learns some relevant fact about the user's background. Examples include:

I am a fourth year computer science graduate student.
→ the user is a member of the 4TH-YEAR-CS-GRAD stereotype
→ the user is LIKELY to be an EXPERT

In my CS2 class, . . .
→ the user is a member of the CS2-STUDENT stereotype
→ the user is LIKELY to be a BEGINNER

In the first example, KNOME infers that the user is a member of the 4TH-YEAR-CS-GRAD stereotype. This allows KNOME to further infer that the user is LIKELY to be an expert, because most 4th year CS graduate students are experts. In the second example, UC's Concretion mechanism infers from "my CS2 class" that the user is a CS2-STUDENT. Based on this, KNOME then infers that the user is a member of the CS2-STUDENT stereotype, and so is likely to be a beginner.

To make Background deductions, KNOME uses a collection of rules, one for each stereotype related to the user expertise levels. For example, the rule for CS2-STUDENT is: if the user is a CS2-STUDENT, the the user is LIKELY to be a BEGINNER.

Query-Reply deductions are made after the system asks whether or not the user knows some fact. The user's answer usually provides information about whether or not the user knows the fact. For example, if a system were to ask the user, "Do you know the rcp command?" and the user answers "Yes," then the system can infer that the user knows the rcp command. This type of deduction has not been implemented in KNOME, since the current version of UC does not ask the user whether the user knows particular facts.

No-Clarify deductions show how a system might obtain information based on what the user does *not* say. If the system uses some terminology that the system is not sure of the user knows, then the system can infer that the user does know this terminology provided that the user does not ask for a clarification and the meaning of the terminology is not evident from the context. Such deductions are more complex than the previous ones, since they require that the system keep track of the conversation. This type of deduction has not been implemented in KNOME, since UC tries to avoid the use of terminology unfamiliar to the user.

In KNOME, deductions are made using a rule-based system. The actual rules were shown in Table 2.2, with the exception of the Background type of inference. The Background type inference is actually implemented by a collection of rules. For every type of user background that gives some clue about the user's level of expertise in using UNIX, there is a rule. Since there are potentially many types of user backgrounds that are relevant, there will be a corresponding number of such rules. For example, if the user is a CS2 student, then that is a relevant piece of background information which provides some clue about user's level of expertise in using UNIX. On the other hand, the fact that the user is a scuba diver, likes blueberry pies, or is a bachelor are irrelevant. The particular Background rules that have been implemented in KNOME are shown in Table 2.3, which also lists the auxiliary inferences which support the major inferences that were shown in Table 2.2.

user is a fourth year CS graduate student	→	user is likely to be an expert
user is a CS2 student	→	user is likely to be a beginner
user does (not) know ?x, where ?x is a description, and ?y is the referent of ?x	→	user does (not) know ?x
user does (not) know a planfor of command ?x	→	user does (not) know ?x
user does (not) know the effects of command ?x	→	user does (not) know ?x

Table 2.3. Other rules used to infer what users know.

Such rules are implemented using *if-detected daemons* (see Chapter VI), which are daemons that detect specific configurations of knowledge in UC's KODIAK knowledge bases. For example, the first rule shown above would normally fire after PAGAN has inferred that the user wants to know something. Then the action part of the rule allows KNOME to infer that the user does not know some fact.

These classes of deductions were found to be important in the domain of a natural language consultation system. Other domains may provide other clues as to what a user knows. For example, in a domain where the user provides speech input, a system may be able to make deductions about the user's knowledge when the user mispronounces technical words.

3.4. Combining Evidence

After collecting evidence that the user does or does not know certain facts, the system must combine the evidence to determine the user's level of expertise. Facts are particular pieces of information about what the user does or does not know as described in the previous section. Note that a single user statement or query may produce several facts. Each unique fact is treated equally in determining the user's level of expertise. When KNOME deduces that the user does or does not know something more than once (perhaps from different user statements), only the first such fact is used in deducing the user's level of expertise. The rest only reconfirm the first fact, so they do not contribute to determining the user's level of expertise.

At the start of a session, KNOME has very little idea about the level of expertise of the user. KNOME's beliefs about the user's level is encoded as ratings of the likelihood that the user is a member of a candidate stereotype category. As evidence is gathered during the dialog, KNOME continuously adjusts its ratings concerning the likelihood that the user might be a member of any particular stereotype. Eventually, KNOME gathers enough evidence to peg the expertise level of the user. Typically, this takes about three interchanges. After making this decision, KNOME does not change its estimation of the user's level. This works well in UC where typical sessions are short and it is advantageous to quickly guess the user's level of expertise. A more flexible approach is to keep a running account of likelihoods as in SC-UM ([Nessen 1986]), a user modeling system that is based on the methodology demonstrated in KNOME. However, a system that does not commit to a particular interpretation of the user's level of expertise cannot avoid the need to store information that can be predicted from the user's stereotype. This becomes inefficient as more is learned about the user.

There are two types of conclusions that may be drawn from any particular fact about what the user knows or does not know. First, the evidence may be enough to eliminate some stereotypes from consideration. An example is when the user does not know a simple fact such as the rm or ls commands. In these cases, KNOME can rule out the possibility that the user may be an intermediate or an expert, since all intermediates and experts know all simple facts (see Table 2.1). So, if the user does not know a simple command like rm or ls, then the user could not possibly be more advanced than a beginner.

Even when the evidence is not enough to rule out a category, the system can still infer that the category is either more likely or less likely. KNOME does this by increasing or decreasing the likelihood rating of candidate categories.

Likelihood ratings are combined using the following linear scale: FALSE, VERY-UNLIKELY, UNLIKELY, SOMEWHAT-UNLIKELY, UNCERTAIN, SOMEWHAT-LIKELY, LIKELY, VERY-LIKELY, TRUE. The UNCERTAIN rating is neutral. It

implies that the system does not believe that membership in that category is either likely or unlikely. Likelihood ratings combine linearly according to the scale. For example, a rating of LIKELY when combined with another LIKELY rating produces a TRUE rating. Also a SOMEWHAT-LIKELY rating combined with a VERY-UNLIKELY rating produces an UNLIKELY rating.

To see why KNOME might change a likelihood rating, consider what happens when KNOME determines that a user does not know some mundane level fact. In this case, KNOME infers that the user cannot be an expert, since experts know all mundane facts. Also, KNOME considers it somewhat less likely that the user might be an intermediate, since intermediates know most mundane level facts. Similarly, the user is considered somewhat more likely to be a beginner, since beginners know a few mundane facts. Finally, since the stereotypical novice does not know any mundane facts, the user is considered more likely to be a novice.

Table 2.4 shows the basis for these deductions. Given that a user does or does not know a fact of some difficulty level, and given that a particular user stereotype knows none/afew/most/all of the facts of that difficulty level, then KNOME can deduce that the user belongs to that stereotype with the additional likelihood shown in Table 2.4.

Stereotype knows Difficulty Level	**Likelihood (user ∈ stereotype)**	
	user does know fact	user does not know fact
NONE	FALSE	LIKELY
AFEW	SOMEWHAT-UNLIKELY	SOMEWHAT-LIKELY
MOST	SOMEWHAT-LIKELY	SOMEWHAT-UNLIKELY
ALL	LIKELY	FALSE

Table 2.4. Deductions when user does (not) know some fact.

Combining Table 2.4 with Table 2.1, yields Tables 2.5 and 2.6. Given the particular stereotypes used in KNOME, these two tables show the deductions that are made when KNOME determines that a user does (Table 2.5) or does not (Table 2.6) some fact that has the difficulty level indicated. Deductions consist of eliminating categories (FALSE) or increasing/decreasing the likelihood ratings of categories. Since these two Tables are easily derivable from the previous two, KNOME does not actually store the information of these two Tables internally. Table 2.1 is represented declaratively as a KODIAK network that is accessible to the rest of UC. Only Table 2.4 is stored internally in KNOME. This Table is used when needed to derive the likelihood rating differences shown in Tables 2.5 and 2.6.

User Stereotype	Difficulty Level of Fact			
	simple	mundane	complex	esoteric
novice	SOMEWHAT-UNLIKELY	FALSE	FALSE	FALSE
beginner	SOMEWHAT-LIKELY	SOMEWHAT-UNLIKELY	FALSE	—
intermediate	LIKELY	SOMEWHAT-LIKELY	SOMEWHAT-UNLIKELY	—
expert	LIKELY	LIKELY	SOMEWHAT-LIKELY	—

Table 2.5. Deduction when user knows some fact.

User Stereotype	Difficulty Level of Fact			
	simple	mundane	complex	esoteric
novice	SOMEWHAT-LIKELY	LIKELY	LIKELY	SOMEWHAT-LIKELY
beginner	SOMEWHAT-UNLIKELY	SOMEWHAT-LIKELY	LIKELY	—
intermediate	FALSE	SOMEWHAT-UNLIKELY	SOMEWHAT-LIKELY	—
expert	FALSE	FALSE	SOMEWHAT-UNLIKELY	—

Table 2.6. Deduction when user does not know some fact.

When the likelihood rating of a stereotype reaches TRUE, it is selected as the user's category. The user's category can also be deduced by elimination. Candidate stereotypes are eliminated when their likelihood drops to FALSE. When all but one candidate have been eliminated, the remaining candidate is selected. During the period before the final decision, KNOME works under the provisional assumption that the user belongs to the category with the highest current likelihood. In the case of a tie (i. e. when there are two or more candidate categories with the same highest likelihood rating), the lower level of expertise is chosen to represent the user temporarily. Using the lower level helps prevent KNOME from overestimating the user before KNOME is sure about the user's level of expertise. Underestimating the user's level may result in more verbose descriptions, but does not lead KNOME to leave out information that the user may not know. Also, inferences that are made based on these provisional stereotype assumptions are

marked as having greater uncertainty.

3.5. Examples

To see how KNOME deduces a user's level of expertise, consider the following traces from the UC sessions shown in Figures 2.5 and 2.6. At the start of either session, UC does not have any information about either user. In such cases, UC starts with a slight bias toward believing that the user is a beginner, because more UC users tend to be beginners. Hence the likelihood rating for the user being a beginner is SOMEWHAT-LIKELY, whereas the likelihood ratings for the user being a novice, intermediate or expert are all UNCERTAIN (the neutral likelihood rating).

In Figure 2.5, KNOME deduces that the user is a novice, because the user does not know either the more or lpr commands, which are both simple commands, nor the rwho command, which is of mundane difficulty. When KNOME first deduces that the user does not know the more command, KNOME can draw a number of inferences based on the fact that the user does not know a simple command. Since experts and intermediates know all simple commands (see Table 2.1), KNOME rules out the possibility that the user might be an intermediate or an expert. Also, since beginners know most simple facts, KNOME lowers the likelihood rating for the user being a beginner from SOMEWHAT-LIKELY to UNCERTAIN. Likewise, since novices know only a few simple facts, KNOME raises the likelihood rating for the user being a novice to SOMEWHAT-LIKELY.

When KNOME next determines that the user does not know the simple command, lpr, KNOME makes additional inferences. KNOME raises the likelihood rating of the user as a novice to LIKELY and lowers the likelihood rating of the user as a beginner to SOMEWHAT-UNLIKELY. Finally when KNOME finds that the user does not know the mundane command rwho, the likelihood rating for the user being a novice rises from LIKELY to TRUE. TRUE is the threshold (nothing can have a higher likelihood than TRUE), so KNOME concludes that this user is a novice.

```
Welcome to UC (Unix Consultant) version 3.23
To a '#' prompt, please type in your questions
about the UNIX file system in English.
To leave, just type '^D' or '(exit)'.

Hi.
How can I help you?

# What will display a file on my terminal?

KNOME: Asserting *USER* does not know ACTION12?

KNOME: Asserting *USER* does not know PLANFOR40

KNOME: Asserting *USER* does not know UNIX-MORE-COMMAND0
```

```
KNOME: Since UNIX-MORE-COMMAND0 is a UNIX-MORE-COMMAND,
       asserting *USER* does not know unix-more-command
KNOME: UNIX-MORE-COMMAND has difficulty SIMPLE, so deducing:
KNOME: ruling out *USER* = INTERMEDIATE
KNOME: ruling out *USER* = EXPERT
KNOME: *USER* is SOMEWHAT-LIKELY to be NOVICE
       => likelihood(*USER* = NOVICE) = SOMEWHAT-LIKELY
KNOME: *USER* is SOMEWHAT-UNLIKELY to be BEGINNER
       => likelihood(*USER* = BEGINNER) = UNCERTAIN

Use more.
For example, to display the file named foo, type 'more foo'.

# How can I print a file on the lineprinter?

KNOME: Asserting *USER* does not know UNIX-DLPR-COMMAND0
KNOME: Since UNIX-DLPR-COMMAND0 is a UNIX-DLPR-COMMAND,
       asserting *USER* does not know UNIX-DLPR-COMMAND
KNOME: UNIX-DLPR-COMMAND has difficulty SIMPLE, so deducing:
KNOME: *USER* is SOMEWHAT-LIKELY to be NOVICE
       => likelihood(*USER* = NOVICE) = LIKELY
KNOME: *USER* is SOMEWHAT-UNLIKELY to be BEGINNER
       => likelihood(*USER* = BEGINNER) = SOMEWHAT-UNLIKELY

Use lpr.
For example, to print the file named foo, type 'lpr foo'.

# What does rwho do?

KNOME: Asserting *USER* does not know STATE11?

KNOME: Asserting *USER* does not know HAS-EFFECT23

KNOME: Asserting *USER* does not know UNIX-RWHO-COMMAND0
KNOME: Since UNIX-RWHO-COMMAND0 is a UNIX-RWHO-COMMAND,
       asserting *USER* does not know UNIX-RWHO-COMMAND
KNOME: UNIX-RWHO-COMMAND has difficulty MUNDANE, so deducing:
KNOME: asserting *USER* = NOVICE

rwho is used to list all users on the network, list their tty,
and list their login time.
```

Figure 2.5. UC session with an novice.

In Figure 2.6, KNOME deduces that the user is an intermediate. First, KNOME deduces that the user knows what an inode is, since the user mentions "the inode of a file." Since knowing about inodes is a complex fact, KNOME is able to eliminate the novice and beginner categories as possibilities. KNOME also lowers the likelihood rating of the user as an intermediate and raises the likelihood rating of the user as an expert.

However, these changes in likelihood ratings are canceled out by KNOME's deduction from the same user query that the user does not know the ls -i command, which is of complex difficulty. Next, KNOME deduces that the user does not know the rwho command, which is of mundane difficulty. This allows KNOME to eliminate the possibility that the user might be an expert, leaving KNOME to conclude that the user is an intermediate, since that is the only possibility left.

```
Welcome to UC (Unix Consultant) version 3.23
To a '#' prompt, please type in your questions
about the UNIX file system in English.
To leave, just type '^D' or '(exit)'.

Hi.
How can I help you?

# Can you tell me how to find out the inode of a file?

KNOME: Asserting *USER* knows INODE1
KNOME: Since INODE1 is a INODE,
       asserting *USER* knows INODE
KNOME: INODE has difficulty COMPLEX, so deducing:
KNOME: ruling out *USER* = NOVICE
KNOME: ruling out *USER* = BEGINNER
KNOME: *USER* is SOMEWHAT-UNLIKELY to be INTERMEDIATE
       => likelihood(*USER* = INTERMEDIATE) = SOMEWHAT-UNLIKELY
KNOME: *USER* is SOMEWHAT-LIKELY to be EXPERT
       => likelihood(*USER* = EXPERT) = SOMEWHAT-LIKELY


KNOME: Asserting *USER* does not know ACTION15?

KNOME: Asserting *USER* does not know PLANFOR390

KNOME: Asserting *USER* does not know UNIX-LS-i-COMMAND0
KNOME: Since UNIX-LS-i-COMMAND0 is a UNIX-LS-i-COMMAND,
       asserting *USER* does not know UNIX-LS-i-COMMAND
KNOME: UNIX-LS-i-COMMAND has difficulty COMPLEX, so deducing:
KNOME: *USER* is SOMEWHAT-LIKELY to be INTERMEDIATE
       => likelihood(*USER* = INTERMEDIATE) = UNCERTAIN
KNOME: *USER* is SOMEWHAT-UNLIKELY to be EXPERT
       => likelihood(*USER* = EXPERT) = UNCERTAIN

Type `ls -i'.

# What does rwho do?

KNOME: Asserting *USER* does not know UNIX-RWHO-COMMAND0
KNOME: Since UNIX-RWHO-COMMAND0 is a UNIX-RWHO-COMMAND,
       asserting *USER* does not know UNIX-RWHO-COMMAND
```

```
KNOME: UNIX-RWHO-COMMAND has difficulty MUNDANE, so deducing:
KNOME: ruling out *USER* = EXPERT
KNOME: only one candidate left, so asserting *USER* = INTERMEDIATE

rwho is like who, except rwho is for all users on the network.
```

Figure 2.6. UC session with an intermediate.

Even before KNOME has completely determined the expertise level of the user, KNOME's partial user model is still useful to the other components of UC. In the dialog with the novice, UC chooses to provide the user with examples of command-formats based on KNOME's initial deduction that the user is SOMEWHAT-LIKELY to be a novice. On the other hand, UC does not provide examples of simple command-formats to the intermediate user who presumably would already know such simple formats. Also, UC is able to use a simile to explain to the intermediate user the rwho command in terms of the who command. The novice user was given a complete explanation of rwho, since KNOME believed that the user would not know the who command and so would not understand the simile.

4. Modeling UC's Knowledge

Besides modeling what users know, KNOME also models what UC itself knows. Such a model allows KNOME to differentiate between cases where UC lacks knowledge and where the user has a misconception (see Chapter III, Section 4).

4.1. Open vs. Closed World Models

An important problem in AI knowledge bases is to define the limitations of the knowledge base. One solution is to use a closed world model, which states that the knowledge base knows everything (see [Reiter, 1978]). In a closed world model, anything that is neither in the knowledge base nor deducible from the knowledge base is deemed false. Such a model works only for completely self-contained micro-worlds such as airline reservation systems that know all airline flights. For real world applications, a closed world model is untenable, since real world knowledge bases cannot know everything even in just their own area of expertise. So a system with a closed model would be prone to giving out false information such as claiming that objects that the system did not happen to know about do not exist, claiming that actions that the system could not figure out cannot be done, etc.

The other extreme is an open world model, which states that anything that is neither in the knowledge base nor directly deducible from the knowledge base is not known. This model has a significant problem with ruling out things that do not exist. For example, a knowledge base might list all the participants in any particular relation. Using a

closed world model, the system would know that these are the only participants of that relation. However, in an open world model, the system cannot rule out that there might not be other participants unless the knowledge base explicitly encodes for all other possible participants that they do not participate in that relation. Since there are usually many more such negative facts than positive facts, encoding such facts in a knowledge base is extremely inefficient.

Neither model is quite adequate. A closed world model allows a system to rule out spurious hypotheses easily, but is prone to ruling out true facts that were not in the knowledge base. An open world model allows the system to be non-committal about hypotheses that are not covered by its knowledge base, but does not allow it to rule out spurious hypotheses when the system *does* have complete information. The ideal system needs to combine the best of both models. It needs to be assertive in ruling out spurious hypotheses when the system does have complete information, yet remain non-committal when it does not have complete information. To do this, a system needs to know the limitations of its own knowledge.

4.2. Meta-Knowledge

UC uses an open world model augmented by knowledge about the coverage limitations of the knowledge base. By using an open world model, UC is able to remain non-committal about information that is not in its knowledge base. This allows UC to profess ignorance about things outside its area of expertise. On the other hand, when UC does have complete information about an area, this is explicitly encoded in KNOME's model of UC's knowledge. In a sense, this is a model of what UC itself knows. Such knowledge is called *meta-knowledge* .

The term meta-knowledge was previously used for AI knowledge bases by [Barr, 1977] and [Davis and Buchanan, 1977]. [Barr, 1977] argues that AI knowledge bases need meta-knowledge, i. e. the system's knowledge about "the extent and limit of available knowledge, what facts are relevant in a given situation, how dependable they are, and how they are to be used." [Davis and Buchanan, 1977] describes meta-knowledge in the TEIRESIAS system and its use in knowledge acquisition and in selecting which rules to try applying first through the use of *meta-rules*. [Smith and Genesereth, 1983] use meta-level reasoning to find all of the solutions to a problem. Although they use a closed world assumption, they suggest that other systems for which the closed world assumption is not valid can encode limits on the number of solutions and use these limits to cut off inference. Such limits can be considered a type of meta-knowledge.

In KNOME, meta-knowledge is not primarily used for the same purposes as in TEIRESIAS; rather it is primarily used in dealing with user misconceptions. So KNOME's meta-knowledge describes the coverage limitations of UC's knowledge base. Chapter III, Section 4 describes how user misconceptions are detected and how meta-knowledge is used in correcting the user's misconceptions.

4.3. Representation of Meta-Knowledge

Meta-knowledge in KNOME is represented explicitly as a collection of KODIAK statements about what UC knows. Examples of meta-knowledge include the fact that UC knows that it knows all file manipulation commands and the fact that UC knows all possible side effects for all known simple commands, and most known mundane commands and a few known complex commands. The fact that UC does not know all of the side effects for known commands is due to not enough programming and the knowledge limitations of the UC's programmers rather than any inherent limitation of UC.

Figure 2.7 shows the representation of KNOME's meta-knowledge about simple commands. KNOW1 states that UC knows all of the effects of simple commands, and KNOW2 represents the fact that UC knows all of the options of simple commands.

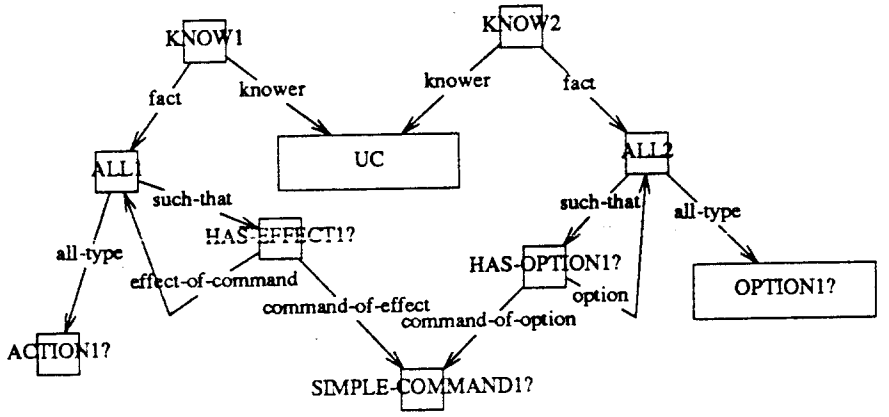

Figure 2.7. KODIAK representation of some of UC's meta-knowledge.

Besides the explicit meta-knowledge modeled by KNOME, UC also has meta-knowledge that is represented implicitly within the KODIAK representation language. For example, any aspectual of a relation can be constrained to be filled by only certain classes of objects. This effectively constrains the way in which different objects may be related to other objects in KODIAK. Of course, these constraints are sometimes violated when users speak metaphorically. [Martin, 1987] describes how such metaphors and analogies can be handled in the UNIX domain.

5. Conclusion

5.1. Summary

The user modeling system of KNOME is extremely simple, yet it provides enough functionality for the needs of UC. KNOME can infer what the user knows from a consulation dialog with the user. Based on only a few such facts about what a user does or

does not know, KNOME can also *predict* many other default facts about what the user might or might not know. KNOME's stereotypes serve to efficiently group those default inferences that commonly co-occur. Using only a few such stereotypes, KNOME is able to provide enough differentiation among potential users so that UC can use these differences in answer expression, goal analysis, and detection of situations when the user lacks knowledge or has misconceptions. The double-stereotype system is not only space efficient, but also allows easy addition of knowledge to KNOME. When a new fact is added to UC, the knowledge engineer needs only to classify the new fact according to its difficulty level.

Although no attempt has been made yet to apply the methodology to other areas, it would seem that many of the techniques should transfer well. Stereotype categories are used in UC in similar fashion to the way humans use categories, that is as reference points for reasoning about individual members (see [Rosch, 1977], [Rosch, 1978], and [Rosch, 1983]). Thus stereotypes should be useful for approximate reasoning wherever human categorization has proven to be useful. The double-stereotype system extends the use of computer stereotype categories to cover more of human categorization, that is to include categories for knowledge as well as categories/stereotypes for people/users. The double-stereotype system can be applied straightforwardly to other domains where a computer system needs to model the relation between different classes of users and different classes of information. In cases where the user population is homogeneous or where the information is homogeneous, then a double-stereotypes system would not be applicable.

The methodology used to deduce the user's stereotype in KNOME transfers in two ways to other domains. First, the method used to infer individual facts about what the user knows is applicable to any natural language system that needs to infer facts about the user's knowledge based on what the user says. On the other hand, the methodology used to combine such evidence is highly dependent on a double-stereotype system. However it can be used in any double-stereotype system where the system can deduce particular relations between the user's stereotype and the modeled information.

5.2. Problems

Currently, KNOME does not address the problem of whether the user knows something after UC has informed the user. The difficulty is that the user may not understand the system's presentation, in which case the user modeling system should not infer that the user knows the new information. A drawback of systems limited to natural language interactions is that these systems cannot watch the user's reactions to deduce whether or not the user is having trouble understanding the system's presentation. Human consultants are able to switch explanation strategies midway through a presentation when they notice their clients' facial expressions showing confusion. A computer consultant would have to wait for the user to respond with language in order to get feedback. In such cases, the best that a user modeling system can do may be to assume that the user has understood (perhaps with a lower certainty factor), until the modeler gets evidence to the contrary. Then, if the user does not ask for a clarification in the next exchange, the modeler can increase the certainty of its belief that the user has indeed understood.

Another deficiency in KNOME is that it does not contain very many specific rules for inferring what specific facts a user might know based on other specific facts that the user knows. For example, if a user knows the rwho command, then it is very likely that the user will also know the who command. KNOME has avoided these types of explicit rules in favor of the more efficient, though less specific mechanism of double stereotypes. The double stereotypes allow KNOME to combine a small number of explicit facts to first infer the user's level of expertise and from this, the user's knowledge of large sets of other facts. Although this powerful general mechanism works well, there are still cases when explicit rules will give more specific information about what a user knows with a higher degree of certainty. The GUMS general user modeling shell ([Finin, 1987]) makes good use of these explicit rules.

A possible deficiency with KNOME's method of combining evidence is that KNOME marks the user as belonging to a particular stereotype after only a short time (typically ~3 interchanges). After this, KNOME does not alter its conception of the user's level of expertise. This is reasonable in the UC domain, because UC sessions are typically short. However, this would not be acceptable for longer term user models. If a system needs to keep track of a user's level of expertise over periods long enough that the user's expertise may increase appreciably, then the system needs to be able to change its judgment of the user's level of expertise. This process is easier than one might suppose, since a user's level of expertise tends to grow monotonically (it seems reasonable to ignore the phenomenon of forgetting).

Currently, KNOME only has a single range of user expertise and a single range of knowledge difficulty. This is sufficient for UC, since UC only covers commands related to the UNIX file system. In order to extend UC to cover other areas of UNIX such as using the vi or emacs editors, KNOME would need additional ranges of user expertise levels and information difficulty levels. Problems which need to be addressed with multiple ranges include how the levels in one range might relate to the levels of other ranges. For example, a system may be able to predict that an expert in using vi is likely to be an expert in using the UNIX file system. On the other hand, there may be no such relations between familiarity with emacs and familiarity with the UNIX file system, since the emacs editor is common to many other operating systems.

Another area which KNOME does not address completely is how to model users of other operating systems. Such users may be able to transfer some amounts of expertise to UNIX. These users can benefit from analogies between UNIX and the operating system that they know. On the other hand, such users sometimes incorrectly apply analogies and often mix up commands. Adding additional stereotypes for such users would enhance KNOME's ability to provide proper analogies and to detect improper analogies or command usage. [Macmillan, 1983] has shown that one user modeling system can adequately model different operating systems (UNIX, TOPS-20, and VM). However, no-one has investigated how knowledge of the user's expertise in one domain transfers to another domain (correctly or incorrectly) or how to exploit such knowledge (e. g. to provide analogies).

Other areas not addressed by KNOME include applying the techniques demonstrated in KNOME for other purposes such as selection of terminology in generation (c. f. [Lehman and Carbonell, 1987]) The approach used in KNOME would also be useful in

the selection of different explanation strategies as in the TAILOR system ([Paris, 1987]).

Chapter III

Goal Detection

The central problem in building an intelligent agent is how to detect appropriate goals for the agent, goals that can then be used to guide the agent's actions. This process is called *goal detection* ([Wilensky, 1983]).

Although considerable work has been done in the area of planning, very few planning systems have addressed the problem of how to detect appropriate goals for planning. In almost all planning systems, the high level goals are provided by the human operators of the planners. An exception is described by [Allen, 1979], whose system simulated a train station ticket agent. It detected goals based on an analysis of obstacles to a user's plans. By addressing these obstacles, the system could volunteer information that the user would need to achieve the user's plan. This approach addresses only a fraction of the general problem of goal detection. An analysis of obstacles to the user's plan does not address how these obstacle-related goals might interact with the system's other goals. Also, analyzing obstacles would not lead a system to detect user misconceptions and detect the goal of correcting the misconceptions. Even in terms of volunteering useful information to the user, an analysis of obstacles to a user's plans does not address the problem when the system should volunteer an alternative plan.

The PANDORA planner ([Faletti, 1982]) also detected its own goals. It detected goals when actual or projected states conflicted with goals or plans and when certain frames describing situations were activated. For example, the goal of "find out about the world" was attached to the "morning" frame, which meant that PANDORA would try to read a newspaper in the morning. However, except for very simple frames, PANDORA did not address the problem of when is it proper to invoke frames and their associated goals. Also, because PANDORA existed in a self-contained simulated world, it did not address the problem of detecting goals when the system must interact with real users.

Once an agent has determined its goals, then the relatively better understood process of planning can be applied to satisfy those goals. A simple plan selection and execution mechanism for satisfying such goals is described in Chapter IV. This chapter describes how goals are represented in UC and how such goals are detected by UCEgo.

1. Definitions and Classifications

This section describes the types of goals found in UC and how they are represented. Different goals are detected by UCEgo in different *situations*. The types of situations that give rise to new goals for UC are classified here, while later sections describe the specific goals that are detected in each type of situation.

1.1. Types of Goals

In UC, goals are represented in KODIAK (see Appendix A) using the HAS-GOAL relation, which has two aspectuals: goal and planner. That is, a goal is modeled as a relation between an individual (planner) and some state (goal) which that individual wishes to achieve. There is no absolute category of goals, since a state cannot be said to be a goal unless some individual can be said to have that goal. This is not to say that there are not some states (e. g. having lots of money) that are habitually thought of as being goals. However, habitual goals encompass only a fraction of what is meant by the term goal. Almost any state can be a goal provided only that some individual wishes to achieve that state. Thus treating goals as aspectuals of HAS-GOAL relations captures the meaning of the term goal. A graphical representation of the definition of the HAS-GOAL relation is shown in Figure 3.1.

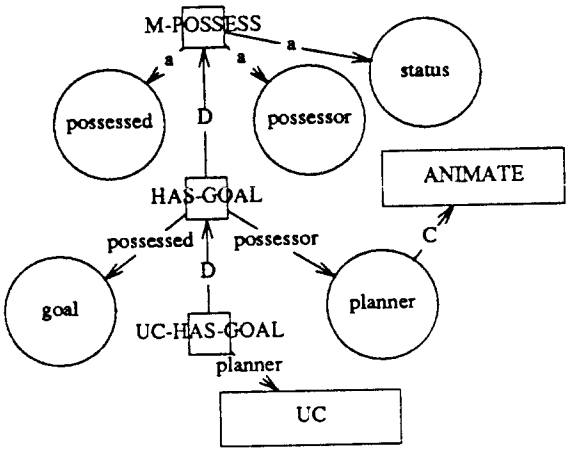

Figure 3.1. Definition of goals in UC.

Besides the goal and planner aspectuals, HAS-GOAL has a status aspectual. This aspectual of HAS-GOAL is actually inherited from M-POSSESS (for mental-possess), which dominates HAS-GOAL as well as KNOW, BELIEVE, and HAS-INTENTION. That is, HAS-GOAL is a kind of M-POSSESS where the planner is the possessor and the goal is the possessed object. The status aspectual describes the status of the relation and can have the values ACTIVE, INACTIVE, or DONE. An HAS-GOAL with an ACTIVE status means that the planner is currently actively pursuing that goal. INACTIVE implies that it is not the case that the planner currently has that goal. Note that this is not the same as saying that the planner has the inverse goal. For instance, if it is not the case that UC wants a file to exist, then it does not necessarily follow that UC wants that file to not exist. It may just be that UC does not care whether that file exists or not. The status DONE implies that the planner has already achieved the goal. The DONE marker is usually applied to ACTIVE goals that become satisfied during the course of a UC session. To be complete, UC should have a full temporal representation based on points, intervals, or some other formalism. However, because full temporal reasoning was not found to be essential for the proper operation of UC, the

DONE status was devised as a simple expedient for marking goals that have been achieved during a session.

Among HAS-GOALs, there is a sub-class called UC-HAS-GOAL where the planner is UC. This separation highlights the importance of UC's own goals to itself. It also facilitates differentiation between the goals of UC and the goals of other agents that can be found in UC's knowledge base.

Other types of goals include *recurrent goals* and *background goals*. Recurrent goals are those goals that recur even after they have been satisfied. An example is UC's goal of helping the user. Even after UC has helped the user once, UC continues to try to help the user. (Recurrent goals in UC are different than cyclic physiological goals — e. g. satiation of hunger, satiation of thirst, and need for rest/sleep — because the cyclic goals are not constantly active; rather, they are inactive for a period immediately after their satisfaction.) Recurrent goals are represented in UC as simple sequences in which the next item in the sequence is itself. In handling goal sequences, UC adopts successive members of the sequence as goals, continuing to the next member after satisfying the previous member. By encoding recurrent goals as self-referential goal sequences, UCEgo avoids the need for a special status for recurrent goals.

Background goals are those goals that do not require immediate attention. Examples of background goals include goals of politeness and various *preservation goals* ([Schank and Abelson, 1977]) that arise from the *Preservation theme* ([Wilensky, 1983]). Examples include self-preservation and preservation of the UNIX system. Since background goals are not dealt with immediately, this creates the problem of deciding when a background goal should be pushed to the foreground. This is equivalent to the problem of detecting new goals, since pushing a background goal to the foreground can be thought of as having a new goal to consider. In UCEgo, the problem of activating background goals is handled by ignoring background goals until UCEgo encounters a situation that suggests a plan for handling the background goal. Once UCEgo detects a plan for the background goal, then the background goal is activated and is treated like any other foreground goal.

[Wilensky, 1983] argues against the need for background goals, such as preservation goals. Instead, themes, such as the Preservation theme, would directly give rise to specific goals in appropriate situations. This scheme would eliminate the extra intermediate level of background goals between themes and the specific goals. However, without this intermediate level of general goals, a system would not be able to abandon some of the general goals of a theme without either abandoning the whole theme, or adding ad-hoc rules that disable the detection of a some subset of situation classes in which that theme should detect specific goals. With intermediate level goals, a system can easily disable part of a theme by disabling the appropriate intermediate level goals. The intermediate level goals also make it easier for a system to reason about its own general goals (e. g. does the system want to preserve itself), since these general goals are explicitly represented in the system.

1.2. Types of Situations

In general, an agent may detect new goals whenever there is a change in an agent's environment or internal state. Any such combination of factors in the agent's environment and/or in the agent's internal state that leads to a new goal for the agent is called a *situation* after the terminology of [Wilensky, 1983].

For the purposes of detecting goals, it is less important to provide a taxonomy of goals, such as in [Schank and Abelson, 1977] and [Carbonell, 1982]. Rather, it is more important to catalog the types of situations that lead a planner to detect new goals. This section will classify the kinds of situations that lead UCEgo to detect new goals.

Since UC is a computer consultation system, UCEgo's environment is limited to a dialogue with the user on the subject of the UNIX operating system. So for UCEgo, situations are composed of combinations of UC's internal state, the user's statements, and information that might be derived from the user's statements such as the user's plans and goals and the user's knowledge and beliefs. UC's internal state includes UC's domain knowledge, UC's own goals, and UC's *themes* ([Schank and Abelson, 1977]).

The situations that give rise to goals in the UC domain can be divided into five main *situation classes:*

1) themes → goals
2) plans → sub-goals
3) goal interactions → meta-goals
4) gaps in the user's knowledge → goals
5) user misconceptions → goals

Themes can be considered as the internal motivations of an agent; so they are a prime source of new goals. Another source of goals is the planning process. As an agent plans for goals, the resulting plans may produce sub-goals that the agent will need to adopt and in turn plan to satisfy. When an agent has several goals, these goals may interact, giving rise to a *meta-goal* ([Wilensky, 1983]), which is a goal for dealing with the interaction among other goals.

The previous three sources of goals are universal in that they are common to all intelligent agents. The other two sources of goals are somewhat more particular to a consulting environment. In a consulting environment, situations commonly arise wherein the state of the user's knowledge base (as deduced by KNOME from conversing with the user) differs from the consultant's knowledge base. One kind of difference is detected when the consultant determines that the user lacks some necessary knowledge. Another kind of difference is found when the consultant determines that the user's knowledge base conflicts with its own knowledge base, that is, that the user has a misconception. Both of these classes of situations are concerned with the user's knowledge, because that is the main task of a consultant, namely to impart information to the user. In other types of programs, a different focus may lead to other situation classes. The situation classes listed above are described in greater detail in the following sections.

In UCEgo, situation classes are encoded using *if-detected daemons*. Each daemon is a tiny inference engine that detects the presence of particular configurations of KODIAK network in UC's knowledge base. When an if-detected daemon detects a match, the daemon is activated and adds its inference in the form of more KODIAK network to UC's memory. In the case of if-detected daemons that are used for goal detection, the matching KODIAK network represents a situation class, and the inference adds a new goal to UC's memory.

An example of an if-detected daemon used for goal detection is shown in Figure 3.2 This is a very simple daemon that asserts that in situations where the user wants to exit (HAS-GOAL1), UCEgo should detect the goal of exiting (UC-HAS-GOAL1). More details on the implementation of if-detected daemons can be found in Chapter VI.

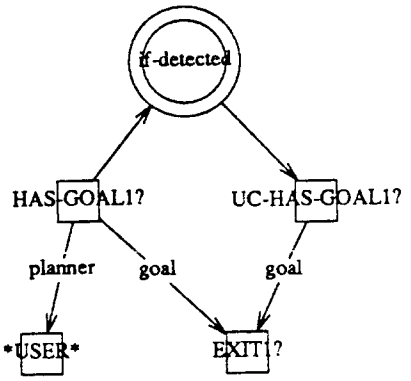

Figure 3.2. If-detected daemon for detecting the goal of exiting.

2. Goals from Themes and Plans

UCEgo has a number of *themes* ([Schank and Abelson, 1977]), which give rise to goals. These include life themes as well as role themes. An example of a life theme is UCEgo's Stay-Alive theme. This theme gives rise to the recurrent background goals of preserving the UC program and preserving the UNIX system. The Stay-Alive theme is also an instance of the *Preservation theme* ([Wilensky, 1983]), since it gives rise to preservation goals. An example of a role theme is UCEgo's Consultant role theme. This gives rise to the recurrent goals of helping the user and being polite to the user. The goal of being polite is a background goal, since UCEgo does not attempt to plan for it immediately. On the other hand, the goal of helping the user is a foreground goal, because UC immediately tries to find ways to help the user.

Since all goals that arise from themes are detected when UC first starts up, it might seem that attributing these goals to themes is extraneous. However themes are really quite useful. First of all, themes provide relative importance ordering for goals, which is useful in case of goal conflicts (see Section 3, Meta-Goals). This relative importance for goals can be used as a basis for a more complete theory for the calculus of the value of different goals for an agent. Secondly, they provide a means of organizing goals into

related groups. For example, if UC were to provide other functions besides a UNIX Consultant, then the goals that arise from its Consultant role theme could be added when UC starts working as a consultant and removed when UC stops working as a consultant.

The goals that arise from themes usually give rise to yet more goals, called sub-goals, during the planning phase of UCEgo. In fact, all of UC's goals can ultimately be traced back to UC's themes, either directly or as sub-goals of other goals, which in turn can be traced back to UC's themes.

2.1. An Example Trace

To see how UCEgo detects goals during the planning process, consider a very simple interaction with UC such as shown in Figure 3.3. The main goal that UCEgo detects in this interaction is the goal (UC-HAS-GOAL50) of having the user know how to delete a file (KNOW-ga0). This goal leads UCEgo to call the UNIX domain planner component of UC to determine the answer. Then UCEgo selects the plan (PLANFOR56) of telling the user the answer. The goal of having the user know is actually a sub-goal of UC's goal of helping the user (UC-HAS-GOAL47), which arose from UC's Consultant role theme.

```
Welcome to UC (Unix Consultant) version 3.23
To a '#' prompt, please type in your questions
about the UNIX file system in English.
To leave, just type '^D' or '(exit)'.

Hi.
How can I help you?

# How can I delete a file?

UCEgo: suggesting the plan:
(PLANFOR55
 (goals55 = (HELP4 (helpee4 = *USER*) (helper4 = UC)))
 (plan55 = (SATISFY4
            (need4 = (KNOW-ga0? (knower-ga0 = *USER*)
                                (fact-ga0 = (ACTION12?
                                             (actor6 = *USER*)))))

            (actor4-0 = UC))))
based on the situation:
(UC-HAS-GOAL47 (status47 = ACTIVE) (goal47 = (HELP4 &)))
(HAS-GOAL-ga0 (planner-ga0 = *USER*) (goal-ga0 = (KNOW-ga0? &)))
```

Here UCEgo suggests a plan for helping the user: satisfy the state of having the user know how to delete a file. ACTION12 represents "how to delete a file."

```
UCEgo: detected the goal:
(UC-HAS-GOAL50 (goal50 = (KNOW-ga0? &)))
from the situation:
(UC-HAS-INTENTION10 (intention10 = (SATISFY4 &))
                    (status10 = ACTIVE))
```

After UCEgo adopts the intention of carrying out the previous plan, UCEgo adopts the sub-goal of having the user know how to delete a file. This is a sub-goal of UC's goal of helping the user.

```
UCEgo: suggesting the plan:
(PLANFOR56
 (goals56 = (KNOW-ga0? &))
 (plan56 = (TELL6 (listener6-0 = *USER*)
                  (speaker6-0 = UC)
                  (proposition6 =
                   (PLANFOR180
                    (goals180 = (DELETE-EFFECT0?
                                 (DELETE-EFFECT-final-state0 =
                                  (EXISTS0 (exist-object0 = FILE3?)
                                           (existence0 = FALSE)))
                                 (DELETE-EFFECT-initial-state0 =
                                  (EXISTS3 (exist-object3 = FILE3?)
                                           (existence3 = TRUE)))
                                 (del-object0 = FILE3?)))
                     (plan180 = (UNIX-RM-COMMAND0
                                 (rm-file0 = FILE3?)
                                 (UNIX-RM-COMMAND-effect0 =
                                  (DELETE-EFFECT0? &))))))
                  (effect6 = (STATE-CHANGE1 (final-state1 =
                                             (KNOW-ga0? &)))))))
based on the situation:
(ANSWER-FOR2 (answer2 = (PLANFOR180 &)) (query2 = (ACTION12? &)))
(UC-HAS-GOAL50 &)
```

After UC determines the answer, UCEgo suggests the plan of telling the user the answer in order the satisfy the goal of having the user know how to delete a file. Since that is a sub-goal of helping the user, it will also help to satisfy that goal.

```
Use rm.
For example, to delete the file named foo, type 'rm foo'.
```

Figure 3.3. Simple UC dialog showing sub-goals.

2.2. Goals from Themes

When UC first starts up, it has a number of themes. These immediately give rise to goals for UC. The themes that have been found to be important for UC include the Stay-Alive and Ethics life themes, and the Consultant role theme. Other systems may find that different themes will be useful. Certainly, systems that are not consultation systems will have a different role theme than UC's Consultant role theme.

The Consultant role theme represents UC's job of being a UNIX consultant. It motivates UC to behave as a consultant to help the user. Therefore, the Consultant role theme leads UCEgo to adopt the recurrent goal of helping the user. This goal is a recurrent goal, since once UC has helped the user, UC continues to have the goal of helping the user. Unlike other goals that arise from themes, the goal of helping the user is not a background goal. This means that UCEgo is constantly planning how to satisfy this goal.

Another aspect of being a consultant involves being polite to the client. So the Consultant role theme leads UCEgo to adopt the recurrent background goal of being polite to the user. This goal is a recurrent goal, since UC never stops being polite to the user. It is also a background goal, since UCEgo does not try to plan how to be polite to the user. Instead, when a situation arises in which UC should be polite, this goal will become activated. Such social situations include greetings, farewells, and apologies. See Chapter IV, Section 2.4 for more details on such social situations.

The Ethics life theme represents UC's desire to act ethically. It gives rise to UC's goal of ACT-ETHICALLY. Since UC cannot perform actions except for communicative acts, UC only has to worry about performing unethical communicative actions. For example UC worries about providing information to the user that will help the user perform an unethical act. In such situations, UCEgo detects a conflict between UC's goal of helping the user (from the Consultant role theme) and UC's goal of ACT-ETHICALLY. Such goal interactions are described further in Section 3.

The Stay-Alive life theme is an instance of the Preservation Theme (see [Wilensky, 1983]), whence arise preservation goals. This particular Preservation theme represents UC's desire to preserve itself. As a result, it leads UC to adopt the goals of preserving the UC program and preserving the UNIX system on which UC runs. Other preservation goals that need to be taken into account in planning how the user should do things in UNIX (e. g. preserving the user's files and preserving the privacy of the user's files) are handled by UC's UNIX domain planner ([Luria, 1985] and [Luria, forthcoming]).

Themes in UC and the goals that they give rise to are summarized in Table 3.1.

Theme	goal	goal-type
UC-Consultant Role Theme	UC help user	recurrent foreground goal
	UC be polite to user	recurrent background goal
UC-Stay-Alive Life-Theme	preserve UC program	recurrent background goal
	preverve UNIX system	recurrent background goal
UC-Ethics Life-Theme	UC act-ethically	recurrent background goal

Table 3.1. Themes that give rise to goals in UC.

2.3. Situations Leading to Sub-Goals

Except for those goals that UC adopts from its themes, all of UC's other goals are sub-goals or, infrequently, meta-goals. This section will show how UCEgo adopts sub-goals and describe some of the sub-goals found in UC. Other sub-goals are introduced as appropriate in later sections.

Sub-goals are created as part of the planning process. Many of UCEgo's plans contain steps that call for the achievement of a state. When UCEgo adopts such a plan, it adopts the sub-goal of the achieving that state. Figure 3.4 shows the if-detected daemon that adopts a sub-goal whenever UC has the intention of satisfying some state.

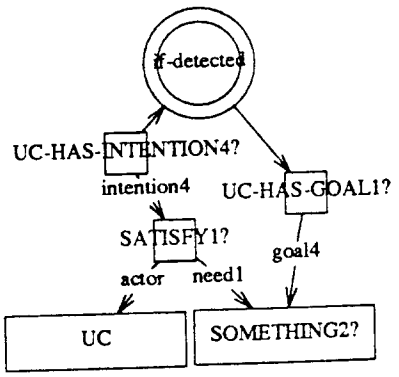

Figure 3.4. If-detected daemon that adopts a sub-goal from an intention.

An example of a particular sub-goal is shown in Figure 3.5. The if-detected daemon shown allows UCEgo to adopt the user's goal of knowing something. This daemon is triggered by a class of situations that consists of two parts:

1) UC has the goal of helping someone.
2) That person wants to know something.

When UCEgo encounters a matching situation, it asserts that a plan for helping the user is to satisfy the sub-goal of having the user know what the user wants to know. This situation is a very common one for UC, because UC's users usually want to know something about UNIX.

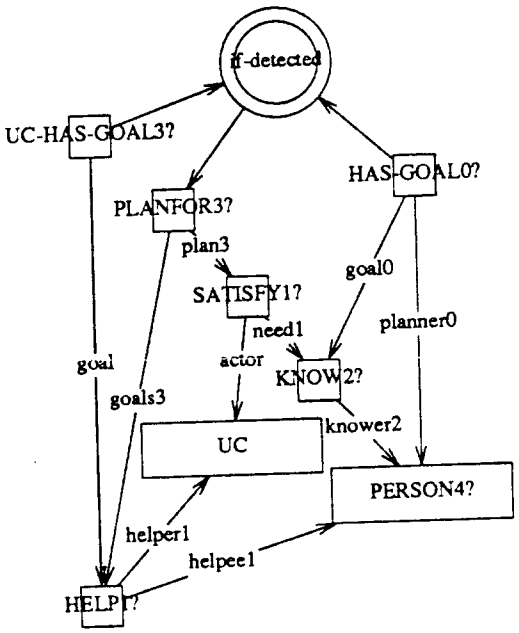

Figure 3.5. If-detected daemon for detecting the sub-goal of having the user know.

Another example of a sub-goal situation is shown in Figure 3.6. This if-detected daemon detects situations in which UCEgo adopts a sub-goal of UC's goal of acting ethically. The type of situation that triggers the daemon involves four relations:

1) UC has the goal of acting ethically.
2) Someone, ?p1, wants to alter a file (alter includes delete).
3) The owner of the file is someone, ?p2.
4) ?p2 (the owner) differs from ?p1 (the alterer).

After detecting such a situation, UCEgo asserts that a plan for acting ethically is to prevent the first person from altering the second person's file. This sub-goal is not very helpful for UC, since UC does not presently have any way to interact with UNIX and so cannot really do anything to prevent the user from deleting someone else's file (e. g. inform the system administrators and/or the owner of the file about the user's intentions).

If UC had emotions, then it would probably be frustrated.

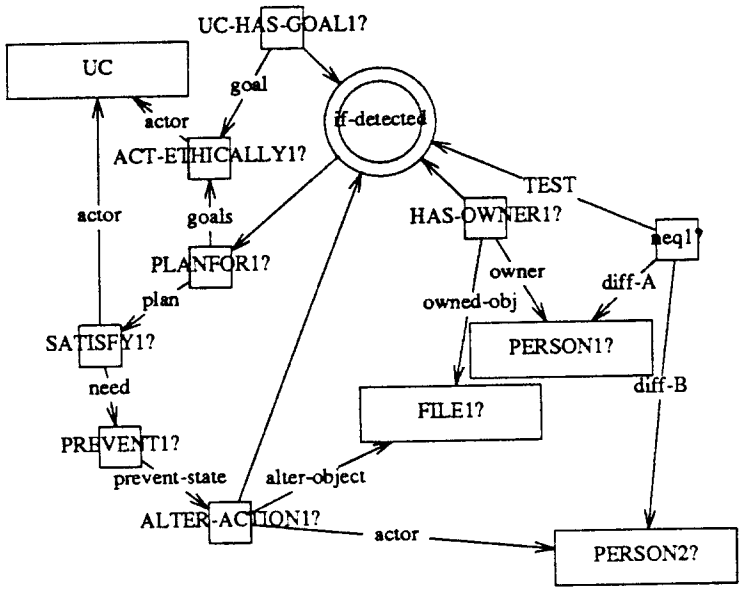

Figure 3.6. Detect sub-goal: prevent someone from altering someone else's file.

The situation class encoded in this if-detected daemon is rather specific; it only detects unethical situations in which someone wants to delete someone else's files. A more general approach would be to have an if-detected daemon detect situations in which:

1) Someone has an unethical goal.
2) UC has the goal of acting ethically.

Then, in such situations, UC can adopt the sub-goal of preventing that person from achieving their unethical goal. With this if-detected daemon, UCEgo would still need to determine which goals are unethical. Such an approach would help to highlight the difference between an agent's reaction to an immoral situation (attempt to prevent it) and the agent's judgment of morality. Since it was not the purpose of UCEgo to develop a systematic representation of morality, the rather specific approach shown above was used for efficiency. The same is true of the next if-detected daemon, shown in Figure 3.7, which encodes the related situation class of someone wanting to know how to alter someone else's file.

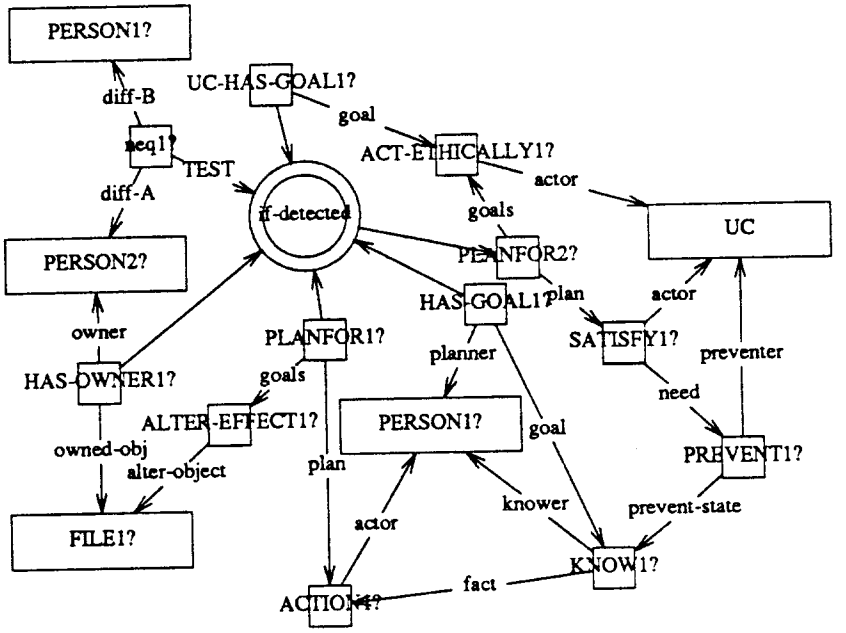

Figure 3.7. Detect sub-goal: prevent someone from knowing something unethical.

This if-detected daemon encodes situations in which:

1) UC has the goal of acting ethically.
2) Someone, ?p1, wants to know:
3) a plan for altering a file (altering includes the sub-class of deleting).
4) The owner of the file is someone, ?p2.
5) ?p2 (the owner) differs from ?p1 (the alterer).

When UCEgo detects such a situation, it asserts that a plan for acting ethically is to adopt the sub-goal of preventing the first person from knowing how to alter the second person's file. Normally this type of situation occurs when the user asks UC how to alter someone else's file. UC should not help the user to perform unethical actions, so in such cases, UC should not provide the user with such information. However this conflicts with UC's normal mode of operation in which UC adopts the user's goal of knowing in order to help the user. This results in an internal goal conflict for UC. Section 3 describes such goal interactions and how they are resolved.

3. Meta-Goals

To correct user misconceptions or provide suggestions to the user does not necessarily require that the system have explicit goals of its own. Those types of responses

can be provided by simpler systems that do not have goals and plans. However, a system based on planning for goals is more flexible, because such a system can much more easily handle interactions among goals. Goals can interact either negatively by conflicting, or positively by overlapping. When UCEgo detects a situation where goals conflict or overlap, it creates a new goal for dealing with the goal interaction. Such goals for dealing with other goals are called *meta-goals* ([Wilensky, 1983]).

Meta-goals in UC, like UC's sub-goals, are completely equivalent to all other goals of UC. That is, meta-goals are not discriminated from other goals in UC either by representational differences or by differences in their processing. Of course, in discussing goals, it is useful to distinguish meta-goals and sub-goals because these types of goals tend to originate from different kinds of situations and tend to have different subject areas.

Meta-goals are also useful for controlling the planning and plan execution process. In UC, meta-goals control when UCEgo tries to find out information for the user in situations in which UC does not know the information. Also, when UCEgo cannot find a pre-stored plan to satisfy one of UC's goals, UCEgo adopts the meta-goal of knowing a plan for finding out a plan to satisfy this goal. This is an example of *meta-planning* ([Wilensky, 1983]), since UC is planning to create a plan that is used to find another plan.

3.1. Controlling Planning

When UCEgo fails to find a plan to satisfy one of UC's foreground goals, it adopts the meta-goal of knowing a plan for satisfying this goal. Since UCEgo selects plans (see Chapter IV, Section 2) using if-detected daemons, it does not know whether it has found a plan for a particular goal until after any relevant if-detected daemons have been activated. So, after all possible daemons have been given a chance to activate, UCEgo looks through its memory to see if there are any foreground goals for which it does not have a plan for satisfying. For each such unsatisfied foreground goal, UCEgo adopts the meta-goal of knowing a plan for satisfying the unsatisfied foreground goal.

A very simple case of such a meta-goal is found when UC first starts up with a new user. At startup, UCEgo adopts the Consultant role theme (see Section 2), which leads to the foreground goal of helping the user. However, before the user has stated a problem to UC, UC does not know exactly how it can help the user. So UCEgo adopts the meta-goal of knowing a plan for helping the user. Then, since UC knows that the user knows how UC can help the user, UCEgo suggests the plan of asking the user how UC can help in order to satisfy the meta-goal of knowing a plan for helping the user. Executing the plan results in UC asking the user, "How can I help you?" A trace of this processing is shown in Figure 3.8.

```
Welcome to UC (Unix Consultant) version 3.23
To a '#' prompt, please type in your questions
about the UNIX file system in English.
To leave, just type '^D' or '(exit)'.

  •
  •
  •

Hi

  •
  •
  •

UCEgo: do not know a single planfor the foreground goal:
(UC-HAS-GOAL56 &)
so adding the meta-goal:
(UC-HAS-GOAL57 (goal57 = (KNOW49? (knower49 = UC)
                                  (fact49 = ACTION11?))))
(PLANFOR61? (goals61 = (HELP5 &)) (plan61 = ACTION11?))
```

UCEgo does not currently have a specific plan for the goal of helping the user, which is a foreground goal, so it adopts the meta-goal of finding out a plan for helping the user.

```
UCEgo: suggesting the plan:
(PLANFOR62
  (plan62 = (ASK8 (asked-for8 = (QUESTION8 (what-is8 = ACTION11?)))
                  (listener8 = *USER*)
                  (speaker8 = UC)))
  (goals62 = (KNOW49? &)))
based on the situation:
*USER*
(UC-HAS-GOAL57 &)
```

Since UC knows that the user knows how UC can help, UCEgo suggests the plan of asking the user in order to find out how to help the user. This plan is described in Chapter IV, Section 2.3.

```
The planner is passed:
((HELP5 &))
```

```
The planner produces:
nil
```

UCEgo always consults the UNIX domain planner, whenever it wants to know a plan. The domain planner fails to devise a plan in this case.

```
The generator is passed:
(ASK8 &)

How can I help you?

#
```

Figure 3.8. Trace showing meta-goal.

UCEgo also uses meta-goals to control the planning process, when UCEgo does not know some information sought by the user. In these situations, UCEgo adopts the meta-goal of finding out the information. Such situations are detected by the if-detected daemon shown in Figure 3.9. More specifically, UCEgo adopts the meta-goal (UC-HAS-GOAL2) of UC knowing something whenever UC has the goal (UC-HAS-GOAL1) of having the user know something, and UC does not know (KNOW1) that.

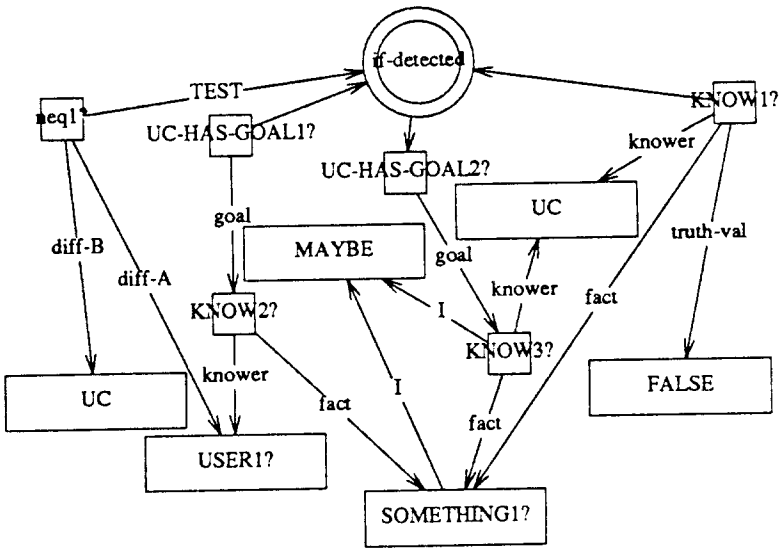

Figure 3.9. Daemon for detecting the goal of finding out something that UC does not know.

Once UCEgo has adopted the meta-goal, the regular planning process is used to devise a plan for UC to find out the information. This meta-goal is meant to model the behavior of human consultants who do not know the answer to a client's question, but can "look it up" to find out the answer. The meta-goal can also be thought of as a simple form of "curiosity," since UC wants to know something when it realizes that it does not know the information. Unfortunately, the current implementation of UC does not have the capability to "look up" answers. However, the plan that UC discovers for looking up answers may be useful to the user, since the user does have the capability to "look up" answers. In such cases, when UCEgo comes up with a plan for finding out the answer to the user's query, and KNOME does not believe that the user already knows this plan, then UCEgo suggests this plan to the user. Section 5.3 discusses how UCEgo makes suggestions in greater detail, while Section 5.5 shows a trace of UCEgo adopting this type of meta-goal.

3.2. Mutual Inclusion

Meta-goals are used to deal with both positive and negative interactions between goals. One way in which goals can interact positively is through *mutual inclusion* ([Wilensky, 1983]). This describes situations in which a planner has the same or similar goals for different reasons. In such situations, the planner can merge the goals into a single goal. This saves resources, because the planner no longer has to plan several times nor execute many similar plans.

When UCEgo finds that it has two goals that are similar, it adopts the meta-goal of merging the redundant goals. The redundant goals may actually be identical, or they may be just similar enough to be merged successfully into a single goal. For example, consider what happens when the user asked UC, "Is cp used to copy files?"

In processing this query, UC's goal analysis component, PAGAN, deduces that the user's goal is to know whether cp is a plan for copying files. PAGAN also deduces that this goal has two possible parent goals (one or both of which might hold):

1) The user wants to know the effects of the cp command.
2) The user wants to know how to copy files.

To see why this added level of goal analysis is necessary, consider the slightly different query, "Is cp used to create files?" The answer to this query is, "No." However this is not a very good answer. In fact, any human consultant who only replied, "No" in this case would be labeled uncooperative. The reason why "No" is not a good answer for this query, whereas "Yes" is a reasonable answer for the first query is because the "No" answer only superficially addresses the user's goals. It only addresses the user's immediate goal, to know whether or not cp is used to create files, but does not address either of the two possible goals that motivate that goal: the user wants to know the effects of the cp command, or the user wants to know how to create files, or both. To provide a more cooperative answer in such situations, UCEgo volunteers additional information by addressing the user's higher level goals as well as the user's immediate goal.

Volunteering information in such cases is described in Section 5.4.

The second query shows that UCEgo sometimes needs to address all of the user's goals rather than just the user's immediate goal. However, if UCEgo were to address all of the user's goals in the first query, then it would end up providing three very similar, indeed redundant, answers to the user. UCEgo might approach this problem of redundant answers in one of two ways. It could always handle only the user's immediate goal and then volunteer more information only if it discovers that satisfying the user's immediate goal does not contribute to satisfying the higher level goals that motivate the immediate goal. The problem with this approach is that it is fairly difficult to tell whether or not satisfying the immediate goal helps to satisfy the underlying goals. In fact, a planner will usually have to plan to satisfy the underlying goals before it can make such a judgment.

Since a planner often must plan for the underlying goals anyway, UCEgo uses the strategy of always planning to satisfy all of the user's goals and then noticing when these goals overlap to prevent redundant answers. Figure 3.10 shows the if-detected daemon that notices situations with potential goal overlap. Whenever UC has two different goals (UC-HAS-GOAL1 and UC-HAS-GOAL2) of wanting the user (PERSON1) to know something, and the answer (ANSWER-FOR1 and ANSWER-FOR2) for those queries are similar (implemented by the procedural test, UC-test-similar), then UCEgo adopts the meta-goal (UC-HAS-GOAL3) of merging the redundant goals (MERGE-REDUNDANT-GOALS).

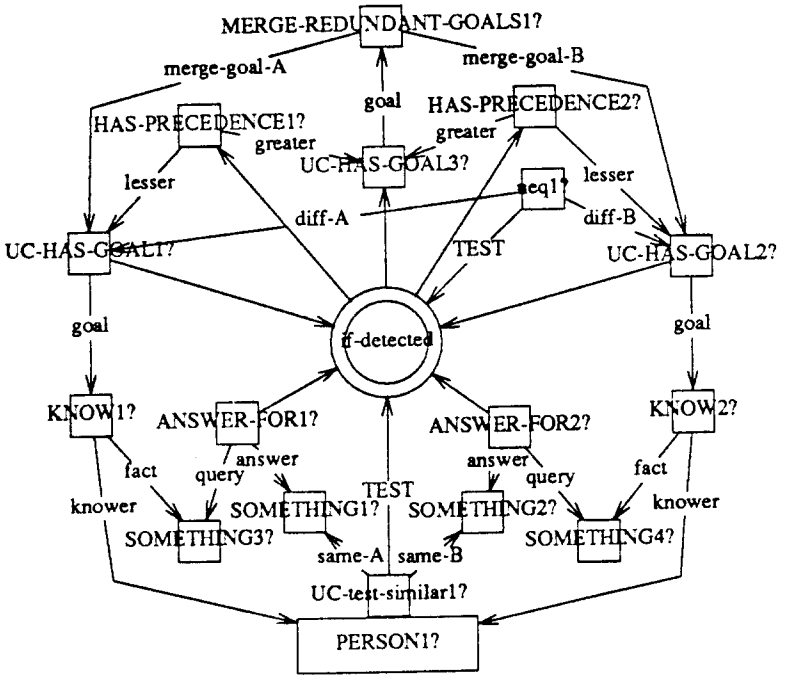

Figure 3.10. If-detected daemon for detecting overlapping goals.

The procedural test, UC-test-similar, checks to see if the two goals are similar enough to be merged. It first matches the two answers to see if they are the same. If so, then the two goals can be merged. It also has knowledge that certain types of relations

are similar enough that they convey essentially the same information. For example, HAS-EFFECT and PLANFOR are similar enough to be merged, provided of course that they relate similar concepts.

The trace of a UC session shown in Figure 3.11, shows how UCEgo merges goals during the processing of the user query, "Is cp used to copy files?" By adopting all three potential user goals, UCEgo detects the three goals:

1) UC wants the user to know whether cp is used to copy files (UC-HAS-GOAL67)
2) UC wants the user to know the effects of the cp command (UC-HAS-GOAL66)
3) UC wants the user to know a plan for copying files (UC-HAS-GOAL68)

UCEgo cannot tell that these goals are similar until after it has deduced the referent of the descriptions in UC's goals. For instance, the referent (encoded as an ANSWER-FOR relation) for the goal of knowing the effects of the cp command is the HAS-EFFECT21 relation, whereas the referent of the goal of knowing a plan for copying files is the PLANFOR340 relation. These are similar, albeit different relations, so they are candidates for merging. Since they both relate a UNIX-CP-COMMAND to a COPY-FILE-EFFECT, UC-test-similar decides that they can indeed be merged. Another type of similarity is found when one answer is contained in the other. In this example, the answer for the goal of knowing whether cp is a plan for copying files is the HAS-TRUTH-VALUE0 relation relating the truth-value, TRUE, and the proposition, PLAN-FOR71, that the cp command is a plan for copying files. When comparing this answer, only the propositional content is compared. Eventually, all three goals are merged by discarding all but the immediate goal (the merging of goals is described in more detail in Chapter IV, Section 2.5).

The final answer is shortened to just, "Yes," rather than "Yes, cp is used to copy files." This is done since the proposition, "cp is a plan for copying files," is already part of the context (it is part of the user's query). This is an example of *pruning* the concepts to be expressed to the user and is described in Chapter V, Section 2.

```
Welcome to UC (Unix Consultant) version 3.23
To a '#' prompt, please type in your questions
about the UNIX file system in English.
To leave, just type '^D' or '(exit)'.

Hi.
How can I help you?

# Is cp used to copy files?

The parser produces:
```

```
(ASK10 (asked-for10 =
        (QUESTION10
         (what-is10 =
          (PLANFOR71? (goals71 = (COPY-FILE-EFFECT0?
                                  (copy-to-file0 = FILE5?)
                                  (copy-from-file0 = FILE4?)))
                      (plan71 = UNIX-CP-COMMAND0)))))
       (listener10 = UC)
       (speaker10 = *USER*))
(COPY-FILE-ACTION0?
 (COPY-FILE-ACTION-effect0 = (COPY-FILE-EFFECT0? &)))

The goal analyzer produces:
((HAS-GOAL-ga2
  (planner-ga0 = *USER*)
  (goal-ga2 = (KNOW-ga2? (knower-ga0 = *USER*)
                         (fact-ga2 = SOMETHING-ga0?))))
 (HAS-GOAL-ga1
  (goal-ga1 = (KNOW-ga1? (knower-ga1 = *USER*)
                         (fact-ga1 = (PLANFOR71? &))))
  (planner-ga1 = *USER*))
 (HAS-GOAL-ga0
  (planner-ga2 = *USER*)
  (goal-ga0 = (KNOW-ga0? (knower-ga2 = *USER*)
                         (fact-ga0 = ACTION-ga0?)))))
```

PAGAN deduces that the user has the goals:

know whether cp is a plan for copying files — HAS-GOAL-ga1

know a plan (ACTION-ga0) for copying files — HAS-GOAL-ga0

know the effects (SOMETHING-ga0) of the cp command — HAS-GOAL-ga2

```
UCEgo: suggesting the plan:
(PLANFOR72 (goals72 = (HELP5 (helpee5 = *USER*) (helper5 = UC)))
           (plan72 = (SATISFY6 (need6 = (KNOW-ga2? &))
                               (actor6 = UC))))
based on the situation:
(UC-HAS-GOAL63 (status63 = ACTIVE) (goal63 = (HELP5 &)))
(HAS-GOAL-ga2 &)

UCEgo: suggesting the plan:
(PLANFOR73 (goals73 = (HELP5 &))
           (plan73 = (SATISFY7 (need7 = (KNOW-ga1? &))
                               (actor7 = UC))))
based on the situation:
(UC-HAS-GOAL63 &)
```

```
(HAS-GOAL-ga1 &)

UCEgo: suggesting the plan:
(PLANFOR74 (goals74 = (HELP5 &))
           (plan74 = (SATISFY8 (need8 = (KNOW-ga0? &))
                               (actor8 = UC))))
based on the situation:
(UC-HAS-GOAL63 &)
(HAS-GOAL-ga0 &)

UCEgo: detected the goal:
(UC-HAS-GOAL66 (goal66 = (KNOW-ga2? &)))
from the situation:
(UC-HAS-INTENTION11 (intention11 = (SATISFY6 &))
                    (status11 = ACTIVE))

UCEgo: detected the goal:
(UC-HAS-GOAL67 (goal67 = (KNOW-ga1? &)))
from the situation:
(UC-HAS-INTENTION10 (intention10 = (SATISFY7 &))
                    (status10 = ACTIVE))

UCEgo: detected the goal:
(UC-HAS-GOAL68 (goal68 = (KNOW-ga0? &)))
from the situation:
(UC-HAS-INTENTION9 (intention9 = (SATISFY8 &))
                   (status9 = ACTIVE))
```

UCEgo adopts all three of the user's goals as its own.

```
UCEgo: trying to find effects for UNIX-CP-COMMAND0
       the effects are:
((HAS-EFFECT21 (command-of-effect21 = (UNIX-CP-COMMAND0 &))
               (effect-of-command21 =
                (COPY-FILE-EFFECT-0 (copy-to-file-0 = FILE-2)
                                    (copy-from-file-0 = FILE-3)))))
```

This is to help satisfy the sub-goal of having the user know the effects of the cp command.

```
UCEgo: checking the truth value of the proposition:
(PLANFOR71? &)
the proposition is TRUE
```

This is to help satisfy the sub-goal of having the user know whether cp is used to copy files.

```
UCEgo: suggesting the plan:
(PLANFOR75
 (goals75 = (KNOW-ga1? &))
 (plan75 =
  (TELL5 (effect5 = (STATE-CHANGE1
                     (final-state1 = (KNOW-ga1? &))))
         (listener5-0 = *USER*)
         (speaker5-0 = UC)
         (proposition5 =
          (HAS-TRUTH-VALUE0 (truth-value0 = TRUE)
                            (t-v-prop0 = (PLANFOR71? &)))))))
based on the situation:
(ANSWER-FOR5 (answer5 = (HAS-TRUTH-VALUE0 &))
             (query5 = (PLANFOR71? &)))
(UC-HAS-GOAL67 &)

UCEgo: suggesting the plan:
(PLANFOR76
 (goals76 = (KNOW-ga2? &))
 (plan76 =
  (TELL6 (effect6 = (STATE-CHANGE2
                     (final-state2 = (KNOW-ga2? &))))
         (listener6-0 = *USER*)
         (speaker6-0 = UC)
         (proposition6 =
          (HAS-EFFECT21 (command-of-effect21 =
                         (UNIX-CP-COMMAND0 &))
                        (effect-of-command21 =
                         (COPY-FILE-EFFECT-0
                          (copy-to-file-0 = FILE-2)
                          (copy-from-file-0 = FILE-3)))))))))
based on the situation:
(ANSWER-FOR4 (answer4 = (HAS-EFFECT21 &))
             (query4 = SOMETHING-ga0?))
(UC-HAS-GOAL66 &)

UCEgo: detected the goal:
(UC-HAS-GOAL69 (goal69 = (MERGE-REDUNDANT-GOALS2
                          (merge-goal-A2 = (UC-HAS-GOAL66 &))
                          (merge-goal-B2 = (UC-HAS-GOAL67 &)))))
from the situation:
(UC-HAS-GOAL67 &)
(UC-HAS-GOAL66 &)
(ANSWER-FOR5 &)
(ANSWER-FOR4 &)
```

UCEgo detects a situation with redundant goals, so it adopts the meta-goal (UC-HAS-GOAL69) of merging the redundant goals.

```
UCEgo: suggesting the plan:
(PLANFOR79
 (goals79 = (MERGE-REDUNDANT-GOALS2 &))
 (plan79 = (UC-merge-goals1 (merge-A1 = (UC-HAS-GOAL66 &))
                            (merge-B1 = (UC-HAS-GOAL67 &)))))
based on the situation:
(UC-HAS-GOAL69 &)

UCEgo: merging the overlapping goals:
(UC-HAS-GOAL66 &)
(UC-HAS-GOAL67 &)
by discarding the goal, UC-HAS-GOAL66
and its sub-goal, UC-HAS-INTENTION12
```

UCEgo merges the goals by discarding one of the goals.

```
The planner is passed:
((COPY-FILE-EFFECT0? &))
```

This is to help satisfy the sub-goal of having the user know a plan for copying files.

```
The planner produces:
(PLANFOR340
 (goals340 = (COPY-FILE-EFFECT0? &))
 (plan340 = (UNIX-CP-COMMAND1 (cp-dest-file1 = FILE5?)
                              (cp-source-file1 = FILE4?)
                              (UNIX-CP-COMMAND-effect1 =
                               (COPY-FILE-EFFECT0? &)))))
(HAS-FILE-NAME17 (named-file17 = FILE5?)
                 (file-name17 = (lisp= nil)))
(HAS-FILE-NAME18 (named-file18 = FILE4?)
                 (file-name18 = (lisp= nil)))
(CP-HAS-FORMAT0
 (CP-HAS-FORMAT-command0 = (UNIX-CP-COMMAND1 &))
 (CP-HAS-FORMAT-format0 =
  (CP-FORMAT0 (CP-FORMAT-two-file-arg0 =
               (CP-TWO-FILE-SEQ0 (cp-dest-file-arg0 =
                                  (file-name17 =
                                   aspectual-of
                                   (HAS-FILE-NAME17 &)))
                                 (cp-source-file-arg0 =
```

```
                                        (file-name18 =
                                         aspectual-of
                                         (HAS-FILE-NAME18 &)))))
                (CP-FORMAT-step0 = cp))))
(HAS-COMMAND-NAME90
 (HAS-COMMAND-NAME-named-obj90 = (UNIX-CP-COMMAND1 &))
 (HAS-COMMAND-NAME-name90 = cp))

UCEgo: suggesting the plan:
(PLANFOR80
 (goals80 = (KNOW-ga0? &))
 (plan80 = (TELL7 (effect7 = (STATE-CHANGE3
                              (final-state3 = (KNOW-ga0? &))))
                  (listener7-0 = *USER*)
                  (speaker7-0 = UC)
                  (proposition7 = (PLANFOR340 &)))))
based on the situation:
(ANSWER-FOR6 (answer6 = (PLANFOR340 &)) (query6 = ACTION-ga0?))
(UC-HAS-GOAL68 &)

UCEgo: detected the goal:
(UC-HAS-GOAL70 (goal70 = (MERGE-REDUNDANT-GOALS3
                          (merge-goal-A3 = (UC-HAS-GOAL67 &))
                          (merge-goal-B3 = (UC-HAS-GOAL68 &)))))
from the situation:
(UC-HAS-GOAL68 &)
(UC-HAS-GOAL67 &)
(ANSWER-FOR6 &)
(ANSWER-FOR5 &)
```

UCEgo detects another pair of overlapping goals.

```
UCEgo: suggesting the plan:
(PLANFOR81
 (goals81 = (MERGE-REDUNDANT-GOALS3 &))
 (plan81 = (UC-merge-goals2 (merge-A2 = (UC-HAS-GOAL67 &))
                            (merge-B2 = (UC-HAS-GOAL68 &)))))
based on the situation:
(UC-HAS-GOAL70 &)

UCEgo: merging the overlapping goals:
(UC-HAS-GOAL67 &)
(UC-HAS-GOAL68 &)
by discarding the goal, UC-HAS-GOAL68
and its sub-goal, UC-HAS-INTENTION15

Express:  not expressing PLANFOR71?,
          since it is already in the context.
```

```
The generator is passed:
(TELL5 &)

Yes.
```

Figure 3.11. UC dialog showing the meta-goal of merging redundant goals.

3.3. Goal Conflict

Goals can interact negatively by conflicting. When UCEgo detects a situation in which UC has two goals that conflict with one another, UCEgo adopts the meta-goal of resolving the goal conflict. A frequent type of goal conflict situation is found when a planner both wants to achieve some state and wants to prevent that state from occurring. UCEgo detects such situations with the if-detected daemon shown in Figure 3.12.

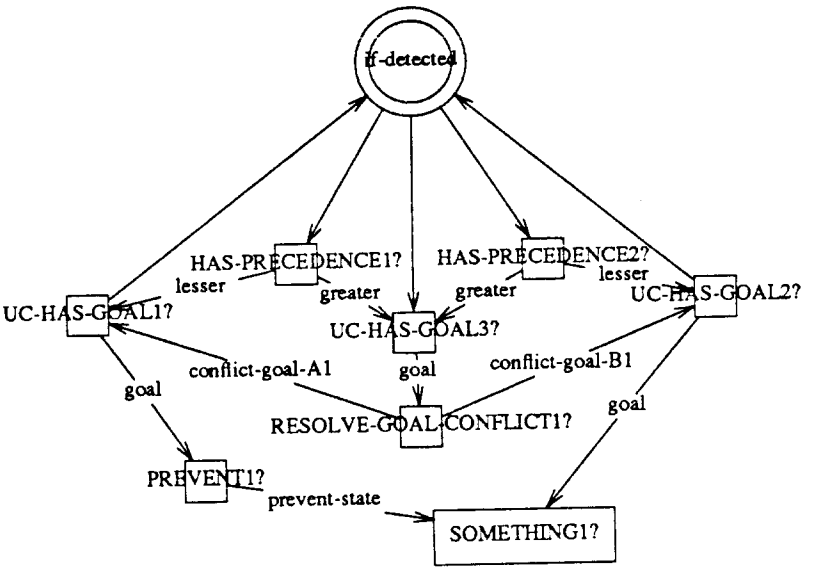

Figure 3.12. If-detected daemon for detecting goal conflict.

An example of a goal conflict situation is when the user asks UC, "How can I delete UC?" In this case, the usual flow of processing leads UCEgo to adopt the user's goal of having the user know a plan for deleting the UC program. This sub-goal can be traced back to UC's goal of helping the user, which in turn arose from UC's Consultant role theme (see Section 2).

In parallel to this, UCEgo also has a Stay-Alive life theme, which gives3 rise to UC's goal of preserving the UC program. This is a background goal (see Section 1, Types of Goals), which means that UCEgo does not immediately plan to satisfy the goal. Rather, the goal becomes active only after UCEgo detects a relevant situation, such as in

the present example. Whenever someone wants to know how to alter something that UC wishes to preserve, UCEgo adopts the sub-goal of preventing the user from knowing how to do that. In this example, the user wants to delete (a specific kind of altering) the UC program, which UC wants to preserve. So, UCEgo adopts the sub-goal of preventing the user from knowing how to delete the UC program. The if-detected daemon that detects such situations is shown in Figure 3.13.

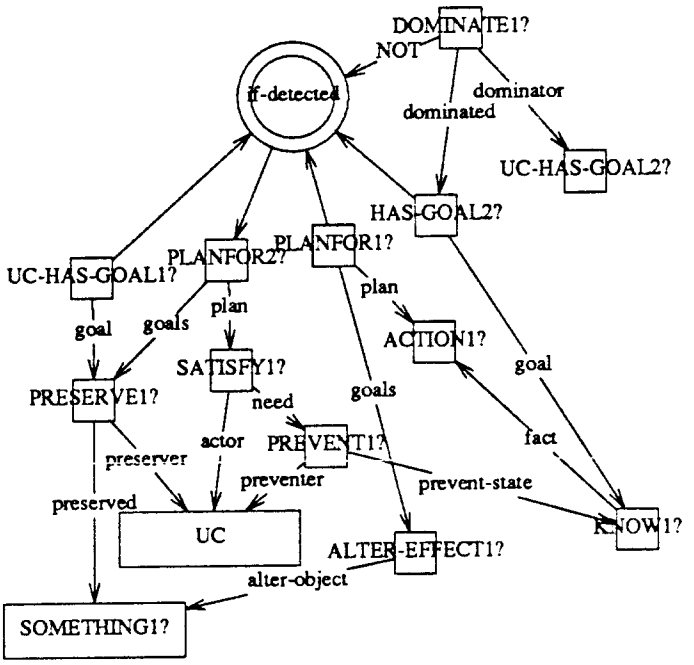

Figure 3.13. Daemon that detects the sub-goal of preventing someone from knowing.

At this point, these two lines of processing interact with a goal conflict. On the one hand, UC wants to help the user and hence wants the user to know how to delete the UC program. On the other hand, UC wants to preserve the UC program and hence wants to prevent the user from knowing how to delete the UC program. The goals of wanting the user to know and wanting to prevent the user from knowing serve to activate the conflict-detection daemon shown earlier in Figure 3.12. Activating the daemon gives UC the goal of resolving the conflict. In this case, the meta-plan of abandoning the lower priority goal (helping the user) is used to resolve the conflict. Thus, UC does not help the user by telling the user how to delete UC. Conflict resolution is described in Chapter IV, Section 2.5.

4. Handling User Misconceptions

User misconceptions are commonly encountered in systems like consultant programs where the system knows more than the user. A user is said to have a misconception when the user's beliefs conflict with what the consultation system knows. In order

to respond properly in such cases, the consultant system must first determine that the user's beliefs conflict with what the system believes, and then correct the user's misconception.

4.1. Other Approaches to User Misconceptions

Other systems that have dealt with user misconceptions use one of two common approaches. The first approach compiles an a priori list of possible user misconceptions for use in detecting misconceptions. This is common among ITS (Intelligent Tutoring Systems) (e. g. [Brown and Burton, 1978], [Stevens et al., 1979], [Sleeman and Smith, 1981], [Johnson and Soloway, 1984], [Anderson et al., 1985], and [Reiser et al., 1985]). These systems are not very flexible, since they cannot deal with any user misconceptions that are not listed among their precompiled a priori lists.

Another approach attempts to detect classes of misconceptions. Among the best of these is Kaplan's CO-OP natural language database-query system ([Kaplan, 1983]), which handles a class of faulty user presumptions. An example from CO-OP is shown in Figure 3.14.

User: Who advises projects in area 36?

CO-OP: I don't know of any area #36.

Figure 3.14. CO-OP example with hedged response to misconception.

The user in the CO-OP example believes that there is an area 36. In processing the user's query, CO-OP finds that there is no area 36 listed in its database, so it detects a possible user misconception: the user may mistakenly presume that there is an area 36 when in actuality there might not be an area 36. Note that CO-OP's reply does not actually *correct* the user's misconception, rather it *hedges*, claiming that it does not know of any area #36. By hedging, CO-OP cannot correct the user's misconception, but can only *suggest* to the user that the user might be mistaken. CO-OP is unable to take corrective action, because it does not assume either an open or closed world model for its database. As a result, it cannot tell from the fact that there is no area 36 in its database whether there really is no area 36, or if area 36 was just left out of its database. Unlike UC, in which the KNOME subcomponent models UC's own knowledge with meta-knowledge (see Chapter II, Section 4), CO-OP does not have a model of its own database.

A corrective response is shown in Kaplan's example of a cooperative human, shown in Figure 3.15.

User: Which students got a grade of F in CS105 in Spring 1980?
Human: CS105 was not given in Spring 1980.

Figure 3.15. Kaplan's example of a cooperative human respondent.

Because CO-OP does not adopt either an open or closed world model, it would not be able to deny that CS105 was given in Spring 1980 as in Kaplan's example of a cooperative human. CO-OP would only be able to suggest a misconception by saying something like, "I don't know of any CS105 in Spring 1980." To be able to actually correct the user, CO-OP would need to know that the absence of a CS105 among the Spring 1980 courses implies that there was no CS105 offered then. That is, CO-OP would need meta-knowledge. Since CO-OP was designed to be easily transportable among different databases, it did not have such non-transportable meta-knowledge.

The DAIQUIRY module of HAM-ANS ([Marburger, 1986]) extended the misconception approach of CO-OP to include conceptual failures in which the user asks about objects not modeled by the database. Because DAIQUIRY, like CO-OP, did not have meta-knowledge, it too could only hedge its replies to the user. For example, it would reply, "The system has no knowledge about 'SHIP'," when SHIP was not in its conceptual hierarchy.

[Mays, 1980] and [Webber and Mays, 1983] report efforts to deal with user misconceptions that can be found by constraint violations on relations. For example, if the TEACH relation is constrained to relate only FACULTY to COURSES, then the system can detect a misconception if the user asks about UNDERGRADUATES that TEACH. This type of relational constraint can be considered a kind of implicit meta-knowledge. It differs from the explicit meta-knowledge of KNOME in that relational constraints can only be used to detect misconceptions about what can be the case, and not misconceptions about what is the case. For example, relational constraints can be used to deduce that only commands can have command-options, but not whether any particular command has any particular command-option.

After a user misconception has been detected, it must be corrected. [McCoy, 1985] and [McCoy, 1987] discuss strategies for correcting object-oriented misconceptions. [Quilici et al., 1987] discusses correcting plan-oriented misconceptions in AQUA. AQUA assumes a closed world model for its knowledge base of plans, called an "advisor model." As a result of this assumption, AQUA can only model perfect advisors that have complete knowledge of the domain. If AQUA's advisor model were incomplete, then AQUA would tend to mislead the user. For example, if an obscure side effect of a plan were inadvertently left out of AQUA's knowledge base, then AQUA would mislead the user to believe that the plan did not have this a side-effect. Some form of meta-knowledge would be required to allow AQUA to model an advisor without perfect knowledge of the domain.

4.2. Detecting Misconceptions in UC

User misconceptions are detected by UC during the processing of the user's query. Currently, UC only handles relational misconceptions, that is, misconceptions in which the user believes a relation holds between two objects when in fact, such a relation cannot hold or does not happen to hold between those particular objects. UC does not handle object-oriented misconceptions, such as those for which [McCoy, 1985] discusses correction strategies.

In processing the user's query, UC checks to see whether all relations mentioned by the user in the user's query have a counterpart in UC's knowledge base. For example, if the user asks, "What does ls -e do?", then UC's parser/understander understands part of this as a HAS-OPTION relation relating an instances of the ls command and an instance of the -e option. While looking for the effects of the command, UC finds that there is no corresponding HAS-OPTION relation between the ls command and the -e option in UC's knowledge base. At this point, UC suspects that the user may have a misconception.

In order to see whether the user actually has a misconception, or whether UC just lacks information about this particular option of this command, UC checks with the KNOME subcomponent, which models the limitations of UC's knowledge base using meta-knowledge (see Chapter II, Section 4). If KNOME has meta-knowledge that UC knows all of the options of this command, then UC knows that the user has a misconception. On the other hand, if the options of this command are not covered by KNOME's meta-knowledge, then UC cannot assume that the user has a misconception. Instead, UC must assume that it lacks information about this command/option combination.

4.3. Correcting Misconceptions

After UC has determined either that the user has a misconception or that UC lacks knowledge, UCEgo decides how UC should reply to the user. When the user has a misconception, UCEgo tries to correct this misconception by denying that what the user mistakenly believes is the case. On the other hand, if UC determines that it lacks information, then UCEgo apologizes to the user for not knowing the answer to the user's query. These two types of responses are shown in the UC session of Figure 3.16.

```
Welcome to UC (Unix Consultant) version 3.23
To a '#' prompt, please type in your questions
about the UNIX file system in English.
To leave, just type '^D' or '(exit)'.

Hi.
How can I help you?

# What does who -b do?

who does not have a -b option.

# What does ruptime -t do?

I'm sorry, I do not know that.
```

Figure 3.16. UC session showing different UC responses to possible misconceptions.

In the first query, UC corrects the user's misconception that who has a -b option. It does this by first noticing that the user's usage of "who -b" translates into a HAS-OPTION relation between a UNIX-WHO-COMMAND and a -b-OPTION. There is no equivalent HAS-OPTION relation in UC's knowledge base, so UC suspects a possible user misconception. To see if this is the case or if UC just lacks knowledge about this particular option, UC consults the meta-knowledge stored in KNOME. The appropriate meta-knowledge in this case is the fact that UC knows all the options of all simple commands. Since who is a simple command, and since the -b-OPTION is not listed among the options of UNIX-WHO-COMMAND in UC's knowledge base, KNOME can infer that there is no such option for who. Next, UCEgo decides that UC should correct the user's misconception by denying the existence of a -b option for who.

On the other hand, in the second query, UC professes ignorance about the -t option of the ruptime command. As in the previous query about who, UC detects a possible misconception when it does not find a -t option listed for ruptime in its knowledge base. However, in this case, ruptime is a complex command, so the previous meta-knowledge does not apply. There is no meta-knowledge about the options of complex commands (due to not enough programming by UC's implementors rather than any inherent limitation of UC), so UC cannot tell if ruptime has a -t option. In order to be polite, UCEgo apologizes to the user for not knowing.

4.4. An Example Trace

The user of Figure 3.17 has the misconception that ls has a -v option. Since ls ignores unknown options, ls -v would do the same thing as ls. So, a system that did not detect and correct user misconceptions and just told the user that ls -v does the same thing as ls would be technically correct, although misleading. Yet, a corrective answer such as UC gives, which technically does not answer the user's question, is a much better answer.

```
Welcome to UC (Unix Consultant) version 3.23
To a '#' prompt, please type in your questions
about the UNIX file system in English.
To leave, just type '^D' or '(exit)'.

Hi.
How can I help you?

# What does ls -v do?

The parser produces:
(ASK10 (listener10 = UC)
       (speaker10 = *USER*)
       (asked-for10 = (QUESTION10 (what-is10 = STATE12?))))
(HAS-EFFECT21? (effect-of-command21 = STATE12?)
               (command-of-effect21 = UNIX-LS-COMMAND0))
(HAS-OPTION3 (option3 = -v-OPTION0)
             (command-of-option3 = UNIX-LS-COMMAND0))
```

The parser understands the user's input as a question about the effects of the UNIX-LS-COMMAND0, which has a -v option (HAS-OPTION3).

```
The goal analyzer produces:
((HAS-GOAL-ga0 (planner-ga0 = *USER*)
               (goal-ga0 =
                         (KNOW-ga0? (knower-ga0 = *USER*)
                                    (fact-ga0 = STATE12?)))))


UCEgo: suggesting the plan:
(PLANFOR71 (goals71 = (HELP5 (helpee5 = *USER*) (helper5 = UC)))
           (plan71 = (SATISFY6 (need6 = (KNOW-ga0? &))
                               (actor6 = UC))))
based on the situation:
(UC-HAS-GOAL63 (status63 = ACTIVE) (goal63 = (HELP5 &)))
(HAS-GOAL-ga0 &)
```

```
UCEgo: detected the goal:
(UC-HAS-GOAL66 (goal66 = (KNOW-ga0? &)))
from the situation:
(UC-HAS-INTENTION9 (intention9 = (SATISFY6 &)) (status9 = ACTIVE))
```

UCEgo adopts the sub-goal of having the user know what the effects of ls -v are.

```
UCEgo: trying to find effects for UNIX-LS-COMMAND0

UCEgo: unknown relation:
(HAS-OPTION3 &)
UCEgo: User has the Misconception:
(HAS-MISCONCEPTION1 (confused1 = *USER*)
                    (misconception1 = (HAS-OPTION3 &)))
since
(KNOW42 (fact42 = (ALL3 (such-that3 =
                         (HAS-OPTION0? (option0 = (ALL3 &))
                                       (command-of-option0 =
                                        SIMPLE-COMMAND0?)))
                        (all-type3 = OPTION0?)))
        (knower42 = UC))
and since the user believes:
(HAS-OPTION3 (option3 = -v-OPTION0)
             (command-of-option3 = UNIX-LS-COMMAND0))
which involves an unknown OPTION

UCEgo: suggesting the plan:
(PLANFOR72
 (goals72 = (HELP5 &))
 (plan72 = (SATISFY7 (need7 =
                      (KNOW59? (knower59 = *USER*)
                               (fact59 = (NEGATE1
                                          (negative1 =
                                           (HAS-OPTION3 &))))))
                     (actor7 = UC))))
based on the situation:
(UC-HAS-GOAL63 &)
(HAS-MISCONCEPTION1)

UCEgo: detected the goal:
(UC-HAS-GOAL67 (goal67 = (KNOW59? &)))
from the situation:
(UC-HAS-INTENTION10 (intention10 = (SATISFY7 &))
                    (status10 = ACTIVE))

UCEgo: suggesting the plan:
(PLANFOR73 (plan73 =
                   (TELL5 (proposition5 = (NEGATE1 &))
                          (listener5-0 = *USER*)
```

```
                              (speaker5-0 = UC)))
            (goals73 = (KNOW59? &)))
based on the situation:
(UC-HAS-GOAL67 &)

The generator is passed:
(TELL5 &)

ls does not have a -v option.
```

Figure 3.17. UC session showing UC handling a user misconception.

UC detects misconceptions as part of the process of fulfilling the user's request. In the example of Figure 3.17, UCEgo detects the misconception in the process of finding the effects of ls -v. After processing by the parser/understander and goal analysis components of UC, the interpretation of the user's query is that the user wants to know the effects of the ls command (UNIX-LS-COMMAND0) with the -v option (HAS-OPTION3). In retrieving the effects of the command, UCEgo must check all of the relations of the command. This process leads to the discovery that the relation HAS-OPTION3 does not correspond to anything in UC's knowledge base. When such an unknown relation is encountered, UCEgo tries to see if it is covered by UC's meta-knowledge. In this case, UC has the meta-knowledge that UC knows (KNOW32) all of the options of simple commands. Since ls is a simple command, UC believes that it knows all of the options of ls. Because -v is not listed among ls's options in UC's knowledge base, UC concludes that it must be a user misconception.

After detecting the misconception, UCEgo detects a plan (PLANFOR56) for helping the user that involves having the user know that ls does not have a -v option. This new sub-goal of correcting the misconception is detected by the if-detected daemon shown in Figure 3.18. This results in UC telling the user that, "ls does not have a -v option."

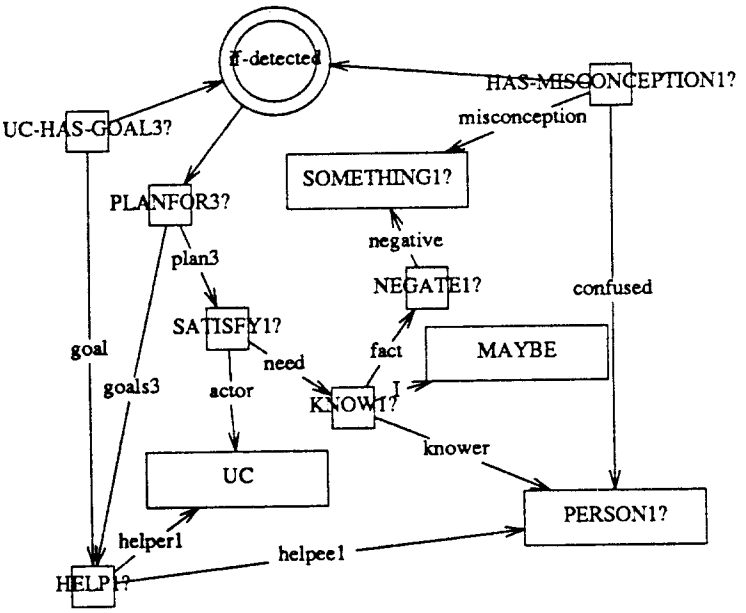

Figure 3.18. If-detected daemon for correcting misconceptions.

5. Filling Gaps in User Knowledge

A major difference between programs that purport to be consultants or tutors and typical application programs is the fact that consultant/tutor programs typically know more about their domain than their users. This leads to problems when users of a consultant system do not know enough to ask the right questions. A tutoring system can usually avoid this problem by properly structuring tutoring sessions or by not providing the user with opportunities for unconstrained inquiry. A consultant system, however, cannot utilize either of these methods. It must be able to handle unconstrained inquiries from the user in its domain, and be prepared to deal with users that do not know enough to ask the right questions.

One type of difficulty encountered by consultant systems in dealing with users occurs when the user lacks some information that is useful for the user's task. In such cases, the consultant should volunteer the information rather than waiting for the user to ask for it. Volunteering the information solves the bottleneck problem that occurs when the user never asks for needed information because the user does not realize than the information is necessary.

5.1. Different kinds of Volunteered Information

In order to be able to volunteer information, a consultant system must do three things:

1) determine that it would be helpful for the user to know some information
2) deduce whether or not the user already knows the information
3) inform the user if the system either believes that the user does not know the information or if the system wants to remind the user of the information

The kinds of information that might be volunteered by a consultant can be divided into three types: *warnings*, *suggestions*, and *elaborations*. Warnings are provided when the consultant believes there is a potential problem for the user. Suggestions are given to present alternatives and methodological hints to the user. Elaborations involve providing additional information that is relevant to the user's query. Each type of volunteered information is described in greater detail below.

5.2. Warnings

A consultant system should consider providing a warning to the user, when the consultant believes that there may be a problem with the user's plans. Two factors come into play when deciding whether or not to give a warning. The first factor is the likelihood that the problem will actually occur. For example, if the user wants to print a file, the user's plan may fail if there is a power blackout or if the printer is out of ink. The chances of a problem caused by a power blackout are so unlikely that giving such a warning would be unreasonable. On the other hand, it may be reasonable to warn the user about the printer being out of ink, if the consultant knows that the printer is currently low or out of ink, or if this particular printer is so heavily used that it frequently runs out of ink.

Another factor in deciding whether or not to give a warning to the user is the consultant's belief about whether or not the user is already aware of the potential problem. Being aware of the problem implies that the user both knows that there is a potential problem with this type of plan, and knows that this problem may arise in this case. If the consultant believes that the user is already aware of the potential problem, then the consultant does not need to warn the user. If the consultant believes that the user knows about this class of problem, but might not apply this knowledge in this particular case, then the consultant should remind the user with a warning. In some cases, the potential problem may be important enough that the consultant may wish to remind the user, even though the user is already aware of the problem.

Common warnings include telling the user about a plan's preconditions that are commonly violated and telling the user about a plan's deleterious side-effects. An example of warning the user about a commonly violated precondition is:

User: How can I delete the directory named misc?

UC: Type 'rmdir misc'.
However, rmdir works only if the directory is empty.

In the example, UC warns the user about a precondition of the rmdir plan, namely that the directory must be empty. This precondition is commonly violated, hence it is a candidate for expressing to the user. This can be contrasted to other preconditions of rmdir such as the need for write permission on the directory containing the directory to be deleted. This precondition is much less frequently violated, so it is not a candidate for expressing to the user. Such frequency information is stored as *concerns* ([Luria, 1987]) by the UNIX domain planning component of UC. When the UNIX domain planner creates a plan, it passes along to UCEgo those preconditions that have a high degree of concern. UCEgo then decides whether or not to warn the user about the precondition depending on whether or not KNOME believes that the user already knows that this type of problem may arise in this such plans. Since KNOME does not model the user's ability to apply general knowledge to individual cases, UCEgo can only assume that, if the user knows about this problem in general, then the user will know that it applies in this specific case. So UCEgo provides a warning only when KNOME believes that the user does not know about the precondition to the plan.

Warnings about a plan's deleterious side-effects are handled in a similar fashion. The UNIX domain planner stores concerns about the side-effects of plans and returns them along with the plan that it produced. In the following example, UC warns the user about one of mv's deleterious side-effects, namely that mv will overwrite files:

User: How can I move the file named new to the file named old?

UC: Type 'mv new old'.
However, be careful, since if the file named old exists, it will be deleted.

5.3. Suggestions

Suggestions are provided by a system to present alternative plans to the user and to present methodological hints to the user. A trace of a UC dialog in which UC volunteers a suggestion to the user is shown in Figure 3.19.

```
Welcome to UC (Unix Consultant) version 3.23
To a '#' prompt, please type in your questions
about the UNIX file system in English.
To leave, just type '^D' or '(exit)'.

Hi.
How can I help you?

# Who is on the system?

I'm sorry, I do not know that.
To find out, type 'who'.
```

Figure 3.19. UC session showing UC volunteering a suggestion to the user.

Since UC was not programmed with direct access to UNIX, UC does not know who is on the system. Hence UC apologizes to the user in order to be polite. However, UC does know how the user can find out who is on the system, namely by using the who command. So, if UC believes that the user does not already know about the who command, UC will suggest this plan to the user. Figure 3.20 shows the daemon that detects the sub-goal of having the user know how to find out something that the user wishes to know when the user does not know this particular plan for finding out the required information (obviously, the user does know one plan for finding out, namely, ask UC).

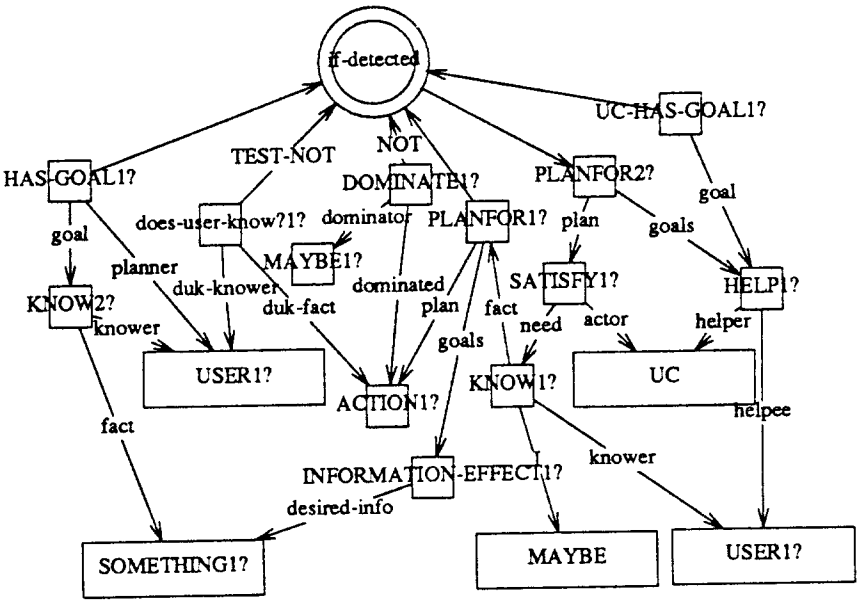

Figure 3.20. If-detected daemon that suggests how to find out information.

In general, this daemon is activated in the following situations:

1) UC wants to help the user (UC-HAS-GOAL1)
2) the user wants to know something (HAS-GOAL1)
3) there is a specific (tested by the NOT DOMINATE1 test) plan for finding out (INFORMATION-EFFECT1) the information
4) the user does not know the plan (tested by the TEST-NOT, does-user-know?1, which represents a call to KNOME)

In such situations, the daemon asserts that a plan for helping the user is to adopt the sub-goal of having the user know the plan for finding out the information.

The previous example shows how UCEgo proceeds when it fails to find a plan for a goal. It adopts the meta-goal (see Section 3.1) of finding a plan for that goal. In this case, UC does not know who is on the system, so it adopts the meta-goal of knowing a plan for finding out who is on the system. UC's UNIX domain planner returns the plan of using the who command, which UC cannot use, because it does not have direct access to UNIX. However, UC can suggest this plan to the user who does have access to UNIX and so can use the plan. The if-detected daemon shown in Figure 3.20 suggests the plan for helping the user of adopting the sub-goal of having the user know the who plan. Note that the daemon only suggests this plan if KNOME believes that the user does not already know about who. After all, it is possible that the user does know who, but instead decided to ask UC in order to find out the information (a plan that fails in this case). An extended analysis of UC's processing in volunteering these types of suggestions is given in Section 5.5. Some other examples of the same type are shown in Figure

3.21.

```
Welcome to UC (Unix Consultant) version 3.23
To a '#' prompt, please type in your questions
about the UNIX file system in English.
To leave, just type '^D' or '(exit)'.

Hi.
How can I help you?

# What is Chin's office?

I'm sorry, I do not know that.
To find out, type 'finger chin'.

# What is the inode of the file named junk?

I'm sorry, I do not know that.
To find out, type 'ls -i'.

# How can I print a file on the laser printer?

Use lpr -Plz.
Don't forget to file the printer output in the boxes.
```

Figure 3.21. UC session showing suggestions by UC.

The last example shows a *reminder* type suggestion. Reminders are given when the system believes that the user already knows some information, but the system also believes that the user is likely to forget or ignore the information. In such cases, the system issues a reminder. In this example, KNOME believes that the user already knows about filing printer output in the boxes, so normally, UCEgo would abandon the goal of suggesting that to the user. However, in this case it is likely that the user will shirk this duty, so UCEgo decides to provide the suggestion as a reminder.

In general, to issue a reminder type suggestion requires that a system perform the following:

1) detect some useful information for suggestion to the user
2) determine that the user is likely to forget/ignore the information
3) inform the user if the system believes that the user needs to be reminded of the information

Whether a user is likely to forget or ignore information is represented in UC in a fashion similar to concerns ([Luria, 1987]). Facts that users are likely to forget or ignore are marked as needing reminding. Since UC will provide a reminder type suggestion regardless of whether the user already knows the information, there is no need to check for this as in other types of suggestions. On the other hand, UC does need to check whether it has direct evidence during the current session that the user is cognizant of the information. So, if the user has previously mentioned the information, or if UC has previously reminded the user of the information, then UC does not provide a reminder type suggestion to the user.

5.4. Elaborations

Elaborations are given when the user asks a simple yes/no question and the system volunteers more information. Volunteering additional information in such cases has been termed *over-answering* by [Wahlster et al., 1983], who implemented over-answering in the HAM-ANS natural language system. An example of elaboration produced by UC is shown in Figure 3.22.

```
Welcome to UC (Unix Consultant) version 3.23
To a '#' prompt, please type in your questions
about the UNIX file system in English.
To leave, just type '^D' or '(exit)'.

Hi.
How can I help you?

# Is rn used to rename files?

No, rn is used to read news.
To rename a file to another file, use mv.
For example, to rename the file named foo to the file named foo1,
type 'mv foo foo1'.
```

Figure 3.22. UC dialog showing an elaboration in UC's response to the user.

Elaborations such as the above require that the system recognize that the user not only has the goal of knowing whether rn is used to rename files, but also that this goal is a sub-goal of two possible parent goals, namely knowing the effects of the rn command and knowing how to rename files. In UC, these deductions are made by UC's goal analysis component, PAGAN. After PAGAN has deduced the user's goals, UCEgo proceeds by adopting all of the user's possible goals. In cases where the answer is yes, both potential parent goals are satisfied by the simple answer of yes, so all three goals

can be merged into a single goal (see Section 3.2), and no elaboration is needed. However, in the above example, a no answer only satisfies the user's immediate goal of knowing whether rn is used to rename files and does not satisfy either possible parent goal. In such cases, UCEgo proceeds to process both parent goals and produces answers to satisfy both. When KNOME believes that the user already knows one of the answers, only the other answer is given to the user.

This goal-based approach to elaboration can be compared to the over-answering methodology of HAM-ANS. HAM-ANS used specific strategies such as filling optional deep case-slots in the case-frame associated with a verb used in the user's request. The problem with such non-goal-based approaches is that they are prone to volunteering information that the user may not actually be interested in. For example, when the user asks HAM-ANS, "Has a yellow car gone by?" HAM-ANS elaborates upon the *where* case-slot to produce the answer, "Yes, one yellow one on Hartungstreet." This is a good answer if the user were actually interested in where the yellow car passed by. However, if the user were interested in how long ago the car passed by, then an elaborative answer like "Yes, fifteen minutes ago," would be much better. Likewise, if the user were interested in following the yellow car, then a better answer would be, "Yes, north on Hartungstreet."

In order to choose between such different elaborations, an analysis of the user's goals is needed. For example, if the user had prefaced the question by the statement, "My friend is supposed to pick me up here," then an analysis of the user's goals would show that the user is probably more interested in how long ago the yellow car passed by. On the other hand, if the user is a police officer chasing a vehicle, then the user is probably interested in following the yellow car. So, deciding how to elaborate a yes/no answer requires a goal-based elaboration strategy such as the one used in UC.

Besides being useful for deciding exactly how to elaborate a yes/no answer, a goal-based strategy also tells the system whether or not it is useful to elaborate at all. For example, when the user asks UC, "Is compact used to compact files?" then a simple answer of "Yes," is quite sufficient. There is no need for UC to elaborate on this answer, because it addresses all of the user's possible parent goals. Similarly, in the hypothetical dialog with a HAM-ANS-like system shown in Figure 3.23, there is no need to elaborate the answer to the user's question, "Has a yellow car gone by?" since an elaboration would not help to satisfy the user's goals.

User: I need the results from your checkpoint for the Cannonball Run auto race.
Has a black Porsche 911 Turbo gone by?

System: Yes.

User: Has a blue pickup gone by?

System: Yes.

System: Has a yellow car gone by?

User: No.

Figure 3.23. Hypothetical dialog in which elaboration is not needed.

5.5. An Example Trace

To see in greater detail how UCEgo decides to volunteer information, consider the annotated trace of a UC session shown in Figure 3.24 below.

```
Welcome to UC (Unix Consultant) version 3.23
To a '#' prompt, please type in your questions
about the UNIX file system in English.
To leave, just type '^D' or '(exit)'.

Hi.
How can I help you?

# What is chin's phone number?

The parser produces:
(ASK9 (listener9 = UC)
      (speaker9 = *USER*)
      (asked-for9 =
       (QUESTION9
        (what-is9 = (STATE-OF-USER1? (user-state1 = USER-PHONE0)
                                     (state-user1 = USER9))))))
(HAS-USER-NAME2 (user-name2 = chin) (named-user2 = USER9))
(HAS-NAME27 (name27 = chin) (named-obj27 = PERSON36))

The goal analyzer produces:
((HAS-GOAL-ga0 (planner-ga0 = *USER*)
               (goal-ga0 =
                (KNOW-ga0? (knower-ga0 = *USER*)
```

```
                    (fact-ga0 = (STATE-OF-USER1? &))))))
```

The user's goal is to know STATE-OF-USER1, which represents the phone number of USER9 who has name "chin."

•
•
•

This part is typical; it leads up to UCEgo adopting the goal of having the user know Chin's phone number.

```
UCEgo detects the following concepts:
(UC-HAS-GOAL59 &)
and asserts the following concept into the database:
(UC-is-state1 (is-state1 = (STATE-OF-USER1? &)))
```

UCEgo calls the procedure UC-is-state to try to look up the state information. The next trace message shows that UC-is-state failed to find the information.

```
UCEgo: UC does not know STATE-OF-USER1?

UCEgo: detected the goal:
(UC-HAS-GOAL60
 (goal60 = (KNOW54? (knower54 = UC)
                    (fact54 = (STATE-OF-USER1? &)))))
from the situation:
(KNOW53 (knower53 = UC)
        (truth-val53 = FALSE)
        (fact53 = (STATE-OF-USER1? &)))
(UC-HAS-GOAL59 &)
```

Since UC wants the user to know Chin's phone number, and UC does not know this information, UC adopts the meta-goal (UC-HAS-GOAL60) of finding out this information (i. e. adopting the goal of having UC — rather than the user as in the previous UC-HAS-GOAL59 — know Chin's phone number). Such meta-goals are discussed in Section 3.1.

```
UCEgo detects the following concepts:
(UC-HAS-GOAL60 &)
and asserts the following concept into the database:
(UC-is-state2 (is-state2 = (STATE-OF-USER1? &)))
```

The procedure UC-is-state is tried again, only this time UC's goal is for UC to know the answer rather than the user. However, the result is the same: UC-is-state does not have the information.

```
UCEgo: suggesting the plan:
(PLANFOR64 (plan64 = (APOLOGIZE2 (apology2 = (KNOW53 &))
                                 (listener2-2 = *USER*)
                                 (speaker2-2 = UC)))
           (goals64 = (BE-POLITE5 (polite-to5 = *USER*)
                                  (is-polite5 = UC))))
based on the situation:
(KNOW53 &)
(HAS-GOAL-ga0 &)
(UC-HAS-GOAL54 (status54 = ACTIVE) (goal54 = (BE-POLITE5 &)))
```

Since UC wants to be polite, and does not know what the user wishes to find out, UCEgo suggests the plan of apologizing to the user for not knowing in order to be polite to the user.

```
UCEgo detects the following concepts:
(UC-HAS-GOAL54 &)
(PLANFOR64 &)
and asserts the following concept into the database:
(UC-HAS-INTENTION11 (intention11 = (APOLOGIZE2 &))
                    (status11-0 = ACTIVE))

UCEgo detects the following concepts:
(UC-HAS-INTENTION11 &)
and asserts the following concept into the database:
(UCexpress3 (gen-prop3 = (APOLOGIZE2 &)))

The generator is passed:
(APOLOGIZE2 &)
I'm sorry, I do not know that.
```

The actual apology.

```
UCEgo: do not know a single planfor the foreground goal:
(UC-HAS-GOAL60 &)
so adding the meta-goal:
(UC-HAS-GOAL61 (goal61 = (KNOW55? (knower55 = UC)
                                  (fact55 = ACTION13?))))
(PLANFOR65?
 (goals65 = (INFORMATION-EFFECT1?
             (desired-info1 = (STATE-OF-USER1? &))
             (informant1 = UC)))
 (plan65 = ACTION13?))
```

Since UC does not know of any planfor satisfying the goal of knowing Chin's phone number (UC-HAS-GOAL60), it adopts the meta-goal of knowing (KNOW55) a plan (ACTION13) for finding out (INFORMATION-EFFECT1) the information.

```
UCEgo detects the following concepts:
(PLANFOR65? &)
(UC-HAS-GOAL61 &)
and asserts the following concept into the database:
(UNIX-planner2 (user-goals2 = (INFORMATION-EFFECT1? &)))
```

UCEgo calls the UNIX domain planner to find a plan for finding out Chin's phone number.

```
The planner is passed:
((INFORMATION-EFFECT1? &))

The planner produces:
(PLANFOR330 (goals330 = (INFORMATION-EFFECT1? &))
            (plan330 = (UNIX-FINGER-COMMAND0
                        (finger-user0 = USER9))))
(FINGER-HAS-FORMAT0
 (FINGER-HAS-FORMAT-command0 = (UNIX-FINGER-COMMAND0 &))
 (FINGER-HAS-FORMAT-format0 =
  (FINGER-FORMAT0
   (finger-name-arg0 = chin)
   (FINGER-FORMAT-UNIX-COMMAND-FORMAT-step0 = finger))))
(HAS-COMMAND-NAME100
 (HAS-COMMAND-NAME-named-obj100 = (UNIX-FINGER-COMMAND0 &))
 (HAS-COMMAND-NAME-name100 = finger))
```

The UNIX domain planner returns the plan of using the finger command.

```
UCEgo detects the following concepts:
(UC-HAS-GOAL62 &)
(PLANFOR330 &)
and asserts the following concept into the database:
(UC-HAS-INTENTION14 (intention14 = (UNIX-FINGER-COMMAND0 &))
                    (status14 = ACTIVE))
```

UC intends to finger Chin, but it cannot, because has not been programmed to have direct access to UNIX.

```
UCEgo: suggesting the plan:
(PLANFOR67 (goals67 = (HELP5 &))
           (plan67 = (SATISFY8 (actor8 = UC)
                               (need8 =
                                (KNOW56? (fact56 = (PLANFOR330 &))
                                         (knower56 = *USER*))))))
based on the situation:
(UC-HAS-GOAL56 &)
(PLANFOR330 &)
(HAS-GOAL-ga0 &)
```

Since UC wants to help the user (UC-HAS-GOAL56), and the user wants to know Chin's phone number (HAS-GOAL-ga0), and finger is a planfor finding out Chin's phone number, UCEgo suggests that a plan for helping the user is for the user to know that finger is a planfor finding out Chin's phone number. This is the if-detected daemon that volunteers information.

-
-
-

As in processing typical queries, UCEgo suggests the plan of telling the user in order to achieve the goal of having the user know the information.

```
The generator is passed:
(TELL5 &)
```

```
To find out, type 'finger chin'.
```

Figure 3.24. UC session showing UC providing a suggestion to the user.

6. Conclusion

6.1. Summary

This chapter has addressed the problems of where goals come from and how an intelligent agent can detect the right goals. These are important problems which have not been systematically addressed by previous planners, whose high level goals were all provided by their human operators. In order to be independent, a system needs be able to detect the right goals, which the agent can then plan to achieve, and in this way act intelligently in response to changes in its environment.

The ultimate source of goals is an agent's themes, which represent the agent's internal motivations. However, themes only provide very general goals, such as helping the user, preserving oneself, and being polite. More specific goals are detected by an agent as sub-goals of these higher level goals (or as sub-goals of sub-goals). Other types of specific goals are detected when goals or plan steps are found to interact (either positively or negatively). In these cases, an agent detects meta-goals for dealing with the interaction among the agent's goals.

Specific goals are detected by a system in particular situations. Since a consultant system's main task is to impart knowledge to its user, situations in which the user either lacks knowledge or has incorrect information (misconceptions) are the most important types of situations for UC's agent, UCEgo. Other important types of situations include those in which UC's goals interact either negatively by conflicting, or positively by overlapping. In such situations, UC detects meta-goals for dealing with the goal interaction.

Situations are difficult to detect since they can consist of arbitrary sets of internal or external states. For example, an agent may have the relatively high level goal of having money (which, in turn, is a sub-goal of other even higher level goals or themes). This agent may even have various scriptal plans for achieving this goal, such as borrowing money, getting a job, winning the money by gambling, etc. However consider what might happen if an eccentric billionaire comes up to the agent and tells the agent, "I will give you a million dollars, if you will stand on your head whenever, during the next month, you hear an electronic wristwatch beep, and the sun happens to be shining." In this case, the agent will detect the new sub-goal of doing a headstand in situations in which the sun is shining and there is a beep from an electronic wristwatch. Of course, the condition could be almost anything, so variants of this scenario will lead an agent to detect new sub-goals in almost arbitrary situations. This arbitrary nature of situations in which agents might detect new goals, makes the goal detection process extremely

difficult.

My approach to efficiently detecting situations (in particular, those that lead an agent to detect new goals) is to encode situations using if-detected daemons. These daemons are in essence tiny inference engines that look for particular types of situations, and add inferences, such as new goals for the agent, when they detect a matching situation.

6.2. Problems

UCEgo only deals with a small subset of potential situations which might lead an agent to detect new goals. In particular, UCEgo is concerned with situations related to the state of knowledge of its user, since UC's main purpose is to provide information to its user. In other domains, agents will need to concentrate on other types of situations. For example, an intelligent agent for driving a car will need to concentrate more on the locations of other cars and road hazards than on its user. It is not clear whether the methodologies demonstrated in UCEgo can be extended to cover other domains that have a different emphasis on the types of situations that can lead an intelligent agent to detect new goals (although, I do believe these methodologies are extensible).

One of the main deficiencies of UCEgo is that it does not have a hierarchy of situation classes. One would like such a hierarchy in order to more efficiently encode related situations. A system that had such a hierarchy could look for the most specific matching situation, and detect the specific goal for that situation. In case where there are no matching specific situations, the agent could fall back on more general situations and the relatively more general goals. This deficiency shows up in some of UCEgo's goal detection situations as a lack of generality in the encoded situations. Many of UCEgo's situations tend to be extremely specific, partly because these are more efficient than general situations. A hierarchy of situation classes would allow a system to retain specific situations for efficiency as well as have more general situations for wider coverage.

An important problem that UCEgo does not address, is how to add new situation/goal pairs to its goal detection mechanism. This problem involves not only learning new situation/goal pairs, but also reasoning about how they might generalize. For example, an agent might learn that if it sees a particular tree catch fire, then it should adopt the goal of extinguishing the fire. The agent might then generalize this to all trees fires, but it should not generalize this to all fires including useful fires such as stovetop burners. Learning new goal detection situations is an important facet of an intelligent agent.

Chapter IV

Plan Selection & Execution

After UCEgo has detected its own goals, it must plan for those goals and then execute the resultant plan. UCEgo selects different plans according to what is appropriate for the current situation. Plans may contain sub-goals, which are then recursively satisfied by UCEgo. This chapter describes the processes of plan selection and then execution in UCEgo.

1. Introduction

Natural language systems act primarily by communicating with the user. These communicative actions are called *speech acts* ([Austin, 1962] and [Searle, 1969]). A planner that produces plans consisting of speech acts has somewhat different requirements than other types of planners. First of all, speech act planners need to perform in real time in order to carry out a dialog with the user. This implies that such planners need to avoid inefficient search and backtracking by using real world knowledge to guide the planning process.

1.1. Other Planners

Planning has a long history in AI, starting from its origins as search within a problem space. In the GPS means-ends analysis formalism of [Newell and Simon, 1963], a planner searches for a sequence of operators that allows the planner to move from an initial state in the problem space to the goal state of the planner. STRIPS ([Fikes and Nilsson, 1971]) is an early example of a planner based on means-ends analysis. ABSTRIPS ([Sacerdoti, 1974]) extended the formalism to work in hierarchical problem spaces. ABSTRIPS broke down a planning problem into a hierarchy of sub-problems and solved each sub-problem independently. Since sub-problems may not actually be independent, planners were developed by [Sussman, 1975], [Tate, 1975], [Warren, 1974], [Waldinger, 1977], [Sacerdoti, 1977], [Stefik, 1980], and others, that could handle planning when the ordering of plan steps is critical to the success of a plan. KAMP ([Appelt, 1981]) applied this type of planner to the problem of planning natural language utterances. KAMP planned English sentences from the speech act level down to selection of actual words.

The previous types of planners develop plans from scratch. This presents a problem, since such planning is computationally expensive. For example, it was not unusual for KAMP to take several hours to plan a complex utterance. Indeed, Appelt developed the TELEGRAM unification grammar ([Appelt, 1983]) to improve the efficiency and modularity of planning at the linguistic level. Developing plans from scratch using "weak methods" such as search and theorem-proving leads to inefficient back-tracking and computationally expensive checking of preconditions for every plan step. These methods do not take advantage of available domain knowledge about the types of plans that are applicable in different situations.

Another problem with general purpose planners that use "weak methods" is that their complexity is not needed in planning speech acts. The OSCAR speech act planner ([Cohen, 1978]) showed that a very simple planner that did not backtrack was sufficient for planning basic speech acts. Although OSCAR did not actually produce natural language output, it did plan speech acts at the conceptual level in enough detail to demonstrate the computational theory of speech act generation devised by [Cohen and Perrault, 1979]. However, since OSCAR did not have to produce speech acts in real time in order to sustain a dialog with a user, it did not worry about how to plan efficiently. Given a goal, OSCAR merely looped through an ordered list of all potential actions until it found one whose effects matched the goal. Only after deciding upon an action would OSCAR test the preconditions of the plan and adopt as sub-goals any preconditions that OSCAR did not believe to already be true. If OSCAR could not satisfy a precondition sub-goal, then it failed in planning since it did not backtrack to try different actions. The fact that OSCAR worked fairly well despite this seemingly severe limitation, shows that planning speech acts does not usually require complex planning.

An alternative to planning from scratch using "weak methods" is presented by [Schank and Abelson, 1975] in their theory of scripts and plans. In their theory, a script consists of a very specific series of steps, while a plan is more abstract and includes other components, such as preconditions on the use of the plan. The TALE-SPIN story generator ([Meehan, 1976], [Meehan, 1981]) implemented this theory to produce plans for the characters in its stories. [Friedland, 1980] and [Friedland and Iwasaki, 1985] describe the MOLGEN and SPEX planners, which extended the idea of scripts and plans into a hierarchy of *skeletal plans*. Skeletal plans are pre-stored plans whose plan-steps may vary in generality from specific actions as in scripts to abstract sub-goals. They are similar to the MACROPs of STRIPS and the chunks of [Rosenbloom and Newell, 1982]. However, the latter systems emphasized learning chunks or MACROPs rather than the selection and instantiation of abstract plans. In a similar vein, [Carbonell, 1986], [Kolodner et al., 1985], [Alterman, 1986], and [Hammond, 1986] have worked on adapting previous plans to new situations using techniques such as analogical reasoning, case-based reasoning, and utilization of a knowledge hierarchy to generalize old plan-steps and then respecify them to form new plan-steps.

Using prestored skeletal plans makes for a much more efficient planner than one that has to plan from scratch using weak methods. However, the use of prestored plans presents a different efficiency problem: how to find the right plan for a specific situation. TALE-SPIN indexed scripts and plans under the goal of the script/plan and then looped through all possibilities, checking the most specific scripts first, until a workable plan was found. Similarly, MOLGEN and SPEX only looked at the UTILITY slot, i. e. the goal, of the skeletal plan. Since skeletal plans of MOLGEN and SPEX did not have preconditions, the planners could not even consider a plan's preconditions to eliminate unsuitable plans. Instead, the planners considered all skeletal plans that fit the goal and returned all proper instantiations of those plans for the user's consideration.

[Hendler, 1985] describes SCRAPS, a planner that used marker-passing to help make more efficient choices during planning. The marker-passing mechanism detected plan choices that might result in intra-plan sub-goal conflicts early in the planning process. The planner could then avoid the potential conflict by choosing an alternative and

hence avoid inefficient backtracking. For example, SCRAPS was able to avoid backtracking while planning in the following situation:

> You are on a business trip (in a distant city). You wish to purchase a cleaver.

By passing markers from BUYING, *ME*, and CLEAVER-27, SCRAPS found the following intersection path:

> BUYING → TAKE-TRIP → PLANE → BOARDING → WEAPONS-CHECK → IF you go through WEAPONS-CHECK with a WEAPON, you get arrested ← WEAPON ← CLEAVER ← CLEAVER-27

This path was then evaluated to determine that it represents a negative interaction, because it is a "bad thing" to get arrested. As a result, SCRAPS ruled out the choice of taking a PLANE (taking a BUS or TRAIN instead), and avoids the backtracking that would be required if the planner had chosen to take a PLANE. One of the problems with such a scheme is the length of the marker-passing path needed to detect important intersections. As longer paths are considered, there are more and more spurious intersections. If a marker-passer were to consider only short paths, then would run the risk of missing important longer path-length intersections. With a path length of eight as in the previous example, any reasonably large knowledge base would produce a very large number of intersections, most of which would be spurious. Even if marker-passing were implemented in parallel, and all of the resulting intersections checked in parallel, it is uncertain whether it would be more efficient to plan using marker-passing or simply to plan in parallel (e. g. a planner could consider traveling by PLANE, BUS, and TRAIN in the previous example in parallel, and then evaluate the best plan later). With current serial machines, marker-passing would undoubtedly be less efficient.

Another problem with SCRAPS is that it ignores other criteria for making choices besides potential negative sub-goal interactions. For example, choosing among the various travel plans in Hendler's example should depend on much more than buying a cleaver. One would not want to take a bus or train if the business trip in the distant city were on the opposite coast of the United States. Choice criteria such as the distance of the trip are not preconditions in the sense that one could still take a bus or train for long distance travel, although one would not want to do so, unless one suffered from fear of flying or lacked the money for a plane ticket. In fact, most people would rather abandon the goal of buying a cleaver, rather than abandon the plan of taking the plane home from a distant city. Of course, most people in that situation would fix their plans by mailing the cleaver home or putting it in checked baggage (to avoid going through weapons-check with the cleaver). However, the latter is not a fair criticism of SCRAPS, since SCRAPS was not programmed with the knowledge needed to make such plan fixes.

1.2. Planning in UCEgo

UCEgo attacks the problem of efficient planning of speech acts in several ways. First of all, like OSCAR, UCEgo uses a very simple planner that avoids inefficient

backtracking. Secondly, like Meehan's TALE-SPIN, Friedland's MOLGEN planner, and Iwasaki's SPEX, UCEgo uses prestored skeletal plans that efficiently encode knowledge about planning for dialog. However, unlike those planners, UCEgo selects for consideration only those skeletal plans that are appropriate to the current situation. A plan is judged appropriate if the goal of the plan is currently a goal of the system, and also if the preconditions and *appropriateness conditions* for the plan are satisfied by being true in the current situation. UCEgo indexes plans according to the type of situation in which the plans are useful. As a result, UCEgo does not waste resources in considering prestored plans that are inappropriate and hence will fail.

Another difference between UCEgo and other planners is that UCEgo uses the idea of meta-planning developed by [Wilensky, 1983] and first implemented in part by [Faletti, 1982]. Through meta-planning, UCEgo is able to handle positive and negative interactions among UCEgo's goals and plans. This notion of meta-planning is different from that of [Stefik, 1981], who used "meta-planning" in reference to his MOLGEN planner. Our notion of meta-planning involves the recursive application of a planner to solve its own planning problems (problems such as interactions among goals or plans). On the other hand, [Stefik, 1981] used "meta-planning" to refer to a multi-layered control structure that allows a planner to schedule its own planning tasks using a variety of strategies, such as least-commitment or application of heuristics. When planning problems arose, MOLGEN could only backtrack. It could not apply itself recursively to create and execute plans for handling the planning problems.

The rest of this chapter describes the processes of planning and plan execution in UCEgo. Section 2 describes plan selection in UCEgo, giving details on the types of situations in which different types of plans are suggested. Plan execution and other types of simple reasoning in UCEgo are described in Section 3.

2. Plan Selection

UCEgo selects plans based on the current situation. Every plan in UCEgo has one or more associated *situation class*. When the situation class associated with a plan matches the current situation, that plan is suggested to UCEgo. The suggestion of plans based on situations is implemented using *if-detected daemons* (see Chapter VI). If-detected daemons can be considered tiny inference engines that look for particular classes of situations. When the current situation matches the situation class of a daemon, it performs appropriate actions such as suggesting a plan.

A simple example of an if-detected daemon used to suggest plans is shown in Figure 4.1. This daemon suggests the plan (PLANFOR1) of having UC exit (UC-exit1) whenever UC has the goal (UC-HAS-GOAL1) of exiting.

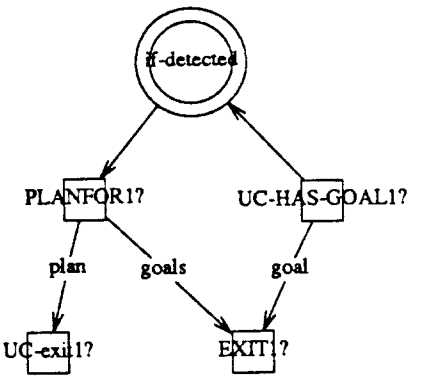

Figure 4.1. Suggest plan of executing the UC-exit procedure when UC wants to exit.

2.1. Situation Types

Situations that suggest plans consist of many different types of information. A plan situation always includes the main goal of the plan. It may also include preconditions and other appropriateness conditions. For example, the plan suggestion situation for a USE-CHAIN-SAW plan might include the following appropriateness condition: need to cut thick branch (over 1 inch diameter). This appropriateness condition is not a precondition, since chain saws can be used to cut smaller branches. Indeed, when a user has already started using a chain saw to cut some thick branches, the user will often use the chain saw for smaller branches. However, if one had only small branches to trim, one would not think of using a chain saw (hedge shears are more appropriate). Adding such appropriateness conditions to a plan suggestion situation prevents the suggestion of inappropriate plans.

The plan suggestion situations in UC can be divided into four main categories. These are situation classes that suggest:

1) inform-plans
2) request-plans
3) social-plans
4) meta-plans

Situations that suggest inform-plans comprise those situations in which the planner wishes to inform the user of some information. Request-planning situations are those in which the planner wishes to request information from the user. Situations that invoke social-plans include salutations and apologies. Meta-planning situations involve suggesting meta-plans for dealing with meta-goals. Each of these situation classes and the plans that are suggested to deal with those classes of situation are described in the following sections.

2.2. Inform-Plans

UCEgo suggests inform-plans whenever UC has the goal of having the user know something. There are two situation classes in which inform-plans are detected. The two classes are distinguished by the type of information that UC wants the user to know. If the knowledge is a network of real concepts, then UCEgo simply suggests the plan of communicating those concepts to the user. On the other hand, if UC wants the user to know something that is only a description of some concept(s), then UC should communicate to the user the concept(s) that is the referent of the description. For example, if UC wants the user to know "how to delete files," then UC should inform the user that "the rm command is used to delete files." "How to delete files" is a description of the concepts, "the rm command is used to delete files." If UC were to communicate just the description to the user, that would not achieve UC's goal of having the user know how to delete a file. UC needs to communicate the referent of the description to the user. As a result, UC needs to compute the referent before informing the user in situations where the type of information is a description.

The two situation classes that suggest inform-plans are summarized in Table 4.1. The first part of the situation in both situation classes is the planner's goal of having the user know something. The second part of the situation in the first class represents the precondition that the information should not be a description. The opposite is the case in the second class. Also, the second class of situation has the additional precondition that the description must have an identified referent.

Situation	Suggested Plan
Planner wants the user to know ?x, and ?x is not a description	tell the user ?x
Planner wants the user to know ?x, and ?x is a description and ?y is the referent of ?x	tell the user ?y

Table 4.1. Situations that suggest inform-plans.

The actual if-detected daemons that detect inform-plan situations and suggest the plans are shown in Figures 4.2 and 4.3. Figure 4.2 shows the if-detected daemon for detecting situations in which UC has the goal (UC-HAS-GOAL3) of having the user know (KNOW1) something (SOMETHING1) that is not a description. Since descriptions in UC are always hypothetical (see Appendix A, Section 2), they can be detected by checking to make sure that the concept to be communicated is not hypothetical. Since hypothetical concepts are marked as being dominated by HYPOTHETICAL, this means that the if-detected daemon should check to make sure that whatever matches SOMETHING1 is not dominated by HYPOTHETICAL. This check is indicated by the NOT link from DOMINATE1 in Figure 4.2.

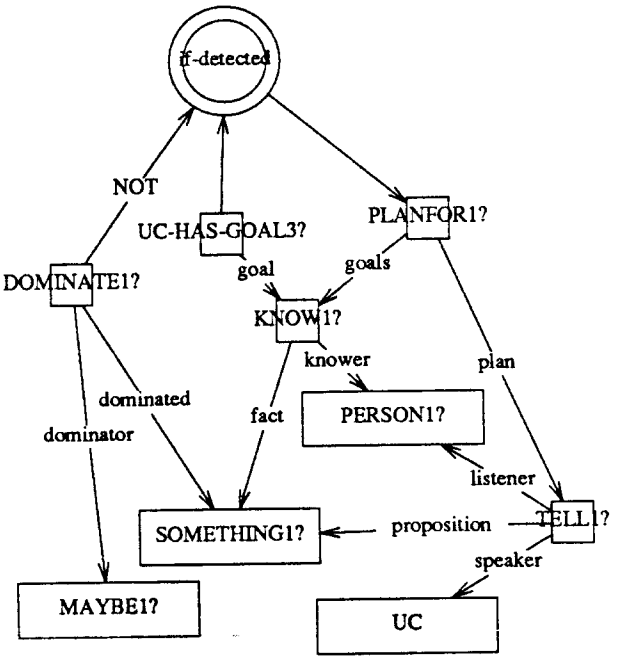

Figure 4.2. Suggest plan of telling user when UC wants the user to know a real concept.

In the second class of situations, UC wants the user to know a description for which UC has identified a referent. Figure 4.3 shows the if-detected daemon for detecting situations in which UC has the goal (UC-HAS-GOAL2) of having the user know (KNOW1) something (SOMETHING1) that is a description. The referent of the description is indicated by the ANSWER-FOR relation between SOMETHING1 (the description) and SOMETHING2 (the referent). There is no explicit check to make sure that SOMETHING1 is indeed a description, because only descriptions participate in the ANSWER-FOR relation.

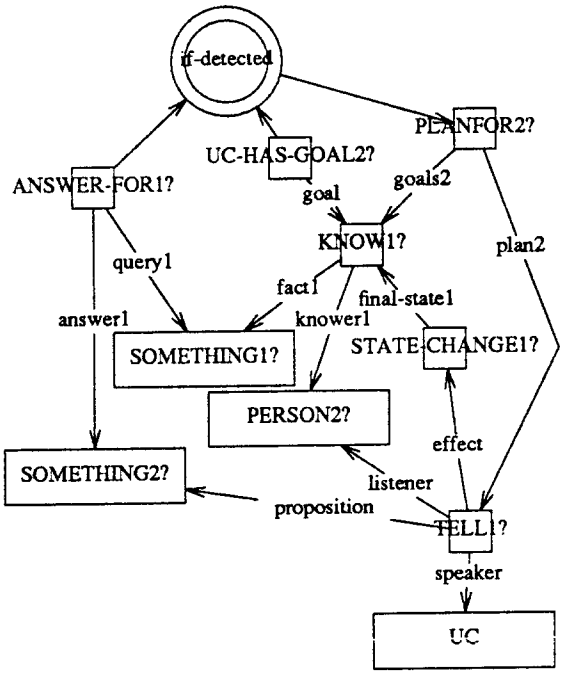

Figure 4.3. Suggest plan of telling the user the referent of a description.

2.3. Request-Plans

The request-plan is suggested by UCEgo whenever UC wants to know something, ?x, and additionally, UC believes that the user is likely to know ?x. The latter condition is not a precondition in the classical sense, since UC can still use the plan of asking the user even when UC does not believe that it is likely to work. Also, the fact that UC does not believe that the user knows the information does not mean that the plan will necessarily fail, since UC may be mistaken in its beliefs about what the user knows. In fact, when they have no better alternatives (or are just too lazy to try other alternatives), people will often ask someone else for information even when they believe that it is very unlikely that person knows the information. Thus the condition that one only asks someone one believes knows the information should not preclude use of this plan. This contradicts [Schank and Abelson, 1977]'s approach in which this is termed an *uncontrollable precondition,* meaning that this plan is always aborted, unless this precondition is true. Nevertheless, one does not normally ask someone for information, when one does not believe that this person possesses the information. This is an example of an appropriateness condition for the use of a plan. UCEgo will not suggest the plan of asking someone for information unless UC believes that the person knows the information sought by UC.

Whether or not the user knows the information sought by UC is modeled by KNOME (see Chapter II). Since such information is often not represented explicitly in UC's knowledge base, but instead is inferable from the user's level of expertise, a call to KNOME is needed to determine whether or not the user knows something. Hence in the

if-detected daemon for suggesting the request-plan, the appropriateness condition is coded as a test involving a call to the KNOME procedure, does-user-know?. This is shown in Figure 4.4.

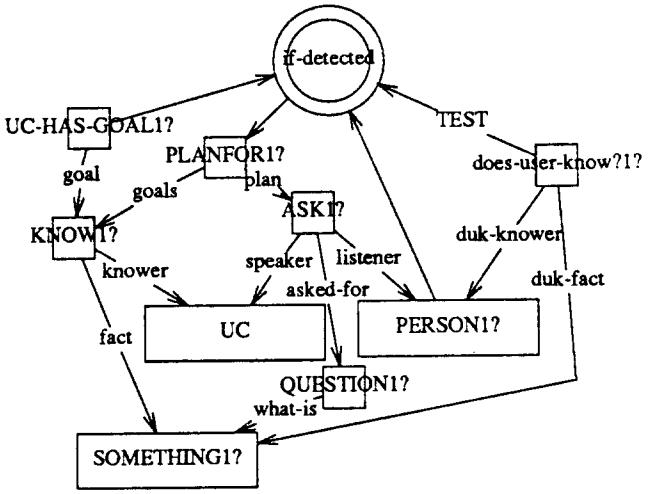

Figure 4.4. Suggest the plan of asking, when it is likely that the user knows.

2.4. Social-Plans

Social-plans consist of salutations and apologies. Common to all situation classes that suggest social-plans is the planner's goal of being polite to the user. If the planner did not wish to be polite, then there would be no point to either greeting the user or apologizing to the user.

Salutations include greetings and farewells. UCEgo suggests the greeting plan, whenever UC first encounters someone (a precondition), and UC has the goal of being polite to this person. The plan of saying good-bye is suggested, whenever UC has the goal of being polite and also has the goal of exiting. Although there are two UC-goals in the good-bye plan's suggestion situation, only one goal is satisfied by the good-bye plan. The good-bye plan is only a plan for being polite, since UC cannot exit merely by means of saying good-bye to the user. The goal of exiting serves as an appropriateness condition for suggesting the plan of exiting. It is not a precondition, because the planner cannot plan to achieve the precondition before using this plan. It is not even an uncontrollable precondition, since it is a condition under the planner's control. After all, if a planner has the goal of being polite to the user, then it might try to use the good-bye plan, and then decide to exit in order to satisfy this precondition of the good-bye plan.

The if-detected daemon that suggests the plan of greeting the user is shown in Figure 4.5. This daemon is triggered, whenever UC has the goal of being polite to someone, and UC encounters this person for the first time. The daemon that suggests the plan of saying good-bye to the user is shown in Figure 4.6. The situations that trigger this daemon are those in which UC has the goal of being polite to someone and also has the goal of exiting.

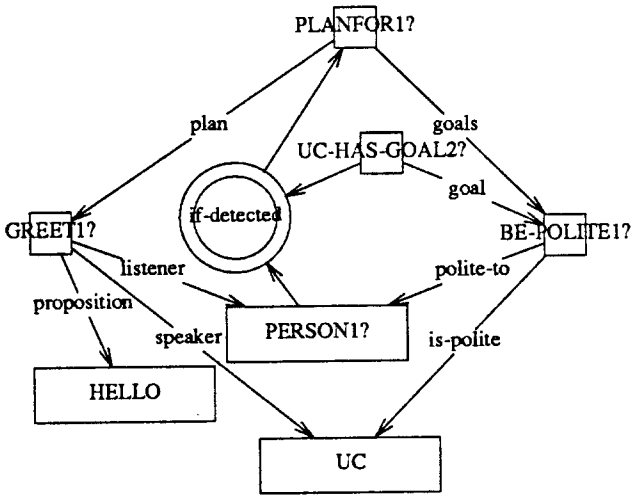

Figure 4.5. Suggest plan of greeting the user when encountering a new user.

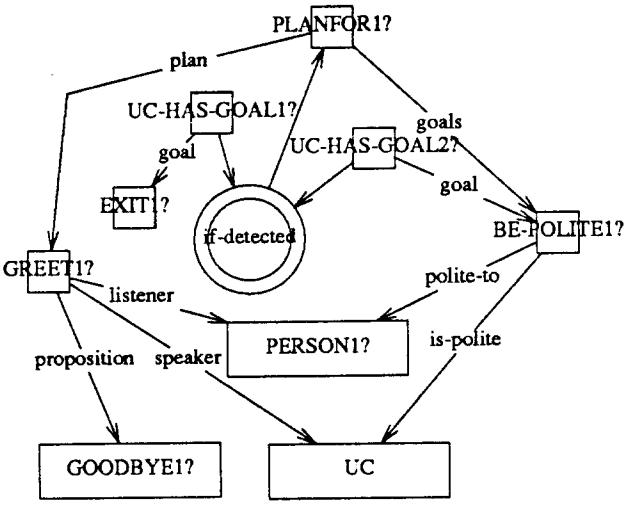

Figure 4.6. Suggest plan of saying good-bye to the user when exiting.

Social goals involving apologies are suggested when UC cannot fulfill its obligations as a consultant. This occurs either when UC cannot tell the user the solution to the user's problem because UC does not know the answer to the user's query, or when UC does not want to tell the user the answer. In the first case, UC apologizes to the user for not knowing. In the second case, UC apologizes to the user for not being able to tell the user (this is really a canned figure of speech, since UC actually is able to tell the user but just does not want to do so). The situations that suggest these plan of apology are summarized in Table 4.2.

Situation	Suggested Plan
Planner has goal of being polite to user, User has goal of knowing something, ?x, Planner does not know ?x	apologize to user for not knowing ?x
Planner has goal of being polite to user, User asked UC a question about something, ?x, Planner wants to prevent the user knowing ?x	apologize to user for not being able to tell user ?x

Table 4.2. Situations that suggest plans of apology.

The actual if-detected daemons that detect situations calling for UC to apologize to the user are shown in Figures 4.7 and 4.8. In the first daemon, the fact that UC does not know something has two possible sources. First, this fact may already be in UC's knowledge base. Secondly, UCEgo may add such knowledge to UC's knowledge base after one of UC's other components (e. g. UC's domain planner) has tried to solve the user's problem and reports a failure. For the second daemon, the fact that UC wants to prevent the user from knowing something is usually the result of a preservation goal. For example, when the user asks UC how to delete UC, this will trigger the goal of preserving the UC program and hence the goal of preventing the user from knowing how to delete UC. This leads to a goal conflict for UC between wanting to tell the user in order to help the user, and wanting to prevent the user from knowing. In this case, UCEgo resolves the conflict by abandoning the goal of wanting to tell the user. Detecting such goal conflict situations are described in Chapter III, Section 3, and the plan for resolving the conflict is described in this chapter in Section 3.3.

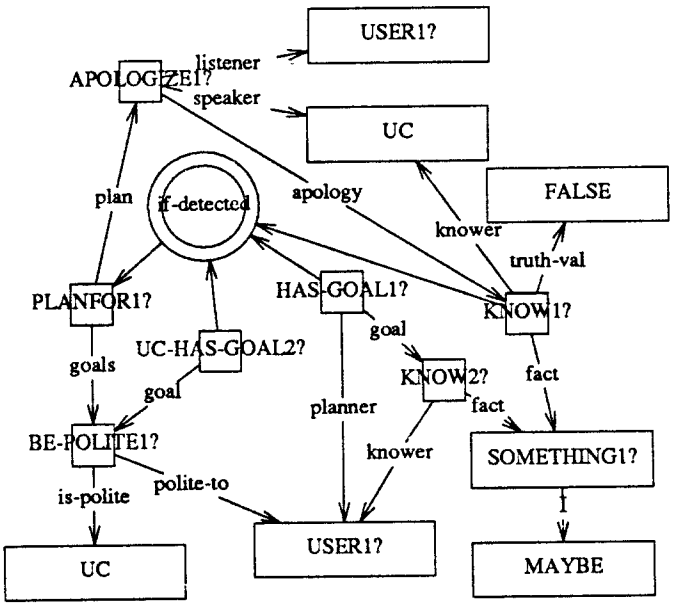

Figure 4.7. Suggest plan of apologizing when UC does not know the answer.

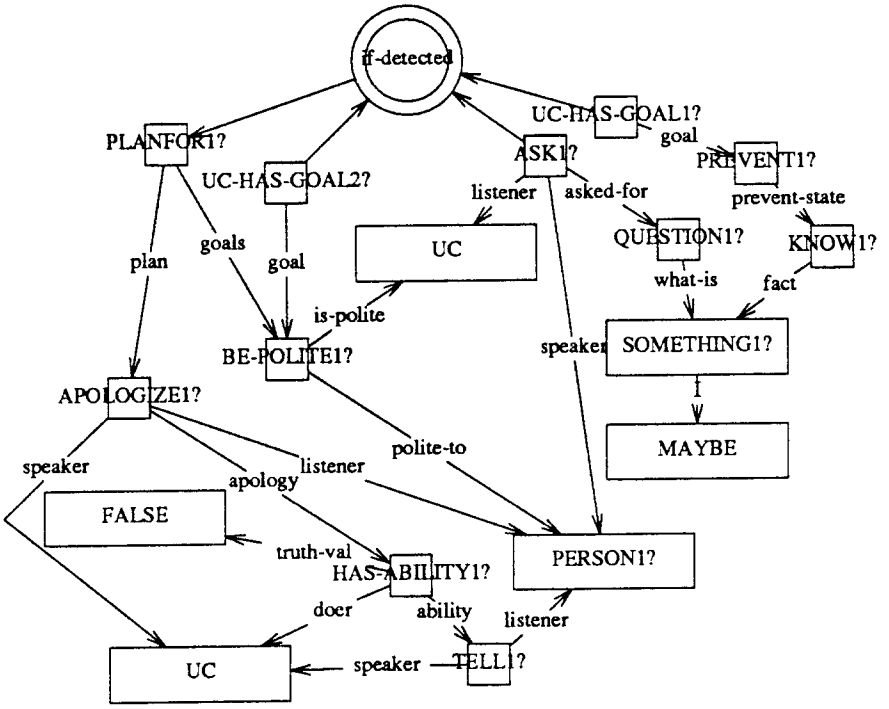

Figure 4.8. Suggest plan of apologizing when UC does not want the user to know.

2.5. Meta-Plans

Meta-plans are just like any other plans in UC. The only difference is that meta-plans tend to be useful for achieving meta-goals. An example of a meta-plan is the plan of calling the procedure, UC-merge-goals, in order to satisfy the meta-goal of MERGE-REDUNDANT-GOALS. The if-detected daemon that suggests this plan is shown in Figure 4.9.

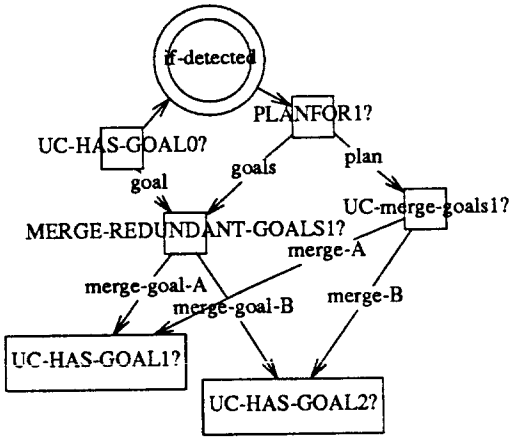

Figure 4.9. Suggest plan of merging redundant goals.

The UC-merge-goals procedure takes two similar goals and merges them. UC-merge-goals first matches the two goals to see if they are identical. If so, the goals can be merged by simply discarding any one of the goals. A more complex case is when one of the goals is contained by the other goal. In such a case, UC-merge-goals discards the contained goal. For example, if the user asks, "Is compact used to compact files?" then UC adopts the following three similar goals (see Chapter III, Section 3.2):

1) UC wants the user know whether compact is used to compact files
   → UC wants the user to know that yes, compact is used to compact files.

2) UC wants the user to know the effects of the compact command
   → UC wants the user to know that compact is used to compact files.

3) UC wants the user to know how to compact files
   → UC wants the user to know that to compact a file, use compact.

The similarity among the goals does not become apparent until after UC deduces the referent of the descriptions in the original goals (see Chapter III, Section 3.2 for more details on detecting goal overlap). Although theoretically the order of merging goals does not make any difference in the final result, in actual practice the referents of the

descriptions of the first two goals are found before the third, so the first two goals listed above are the first to be merged. In merging the first two goals, the second goal is contained by the first, so the goals are merged by simply abandoning the second goal. Next, after UC identifies the referent of the third goal, UCEgo notices that it is similar to the first goal (a similarity with the second goal is not detected, since the second goal has already been abandoned at this point). Once again, the third goal is approximately contained by the first goal (approximate in that "to compact a file, use compact" is represented as a PLANFOR relation, which is similar to but not identical to the HAS-EFFECT relation that is used to represent, "compact is used to compact files), so the two goals are merged by abandoning the third goal. These two merges leave only the first goal, which leads to UC's answer of "Yes." The propositional part of this answer is pruned by UCExpress (see Chapter V, Section 2).

Another of UCEgo's meta-plans is suggested when UCEgo detects a goal conflict and adopts the meta-goal of resolving the conflict. The appropriate meta-plan is suggested by the if-detected daemon shown in Figure 4.10. This meta-plan represents a call to the procedure, UC-resolve-conflict, which resolves the conflict by abandoning the less important of the two conflicting goals. To determine which goal is less important, UC-resolve-conflict first searches for a direct precedence relationship (represented by a HAS-PRECEDENCE relation) between the two goals. If such a relation does not exist, then UC-resolve-conflict expands the search to include the causal parents of the goals. The search continues until the ultimate sources of the goals, which are usually UC themes, are included in the check for relative precedence relations. Since goal conflicts usually involve goals that originate from different UC themes, and, because all of UC's themes have a relative precedence, UC-resolve-conflict is almost always able to decide which goal to abandon in order to resolve the conflict.

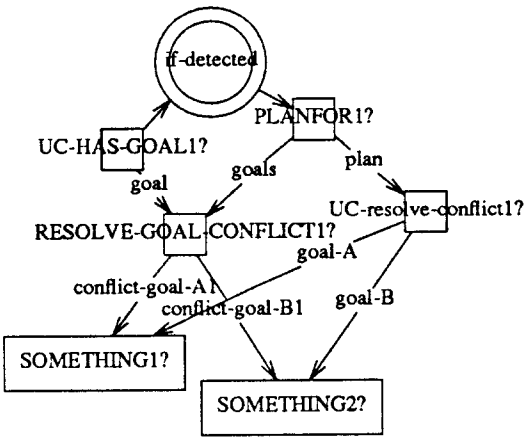

Figure 4.10. Suggest plan of resolving the conflict.

An example of resolving a goal conflict is shown in the trace of a UC session shown in Figure 4.11. In this dialog, the user asks UC how to crash the system, which leads UC to adopt the following two conflicting goals:

1) UC wants the user know how to crash the system (UC-HAS-GOAL66).
2) UC wants to prevent the user from knowing how to crash the system (UC-HAS-GOAL67).

The first goal is a sub-goal of UC's goal of helping the user, which in turn originates from UC's consultant role theme. The second goal is a sub-goal of UC's goal of preserving the system, which in turn originates from UC's staying alive life theme. UCEgo detects the fact that these two goals conflict (see Chapter III, Section 3.3), since UC both wants to achieve some state and prevent the achievement of that state. To resolve the goal conflict, UCEgo calls the UC-resolve-conflict procedure, which checks the relative precedence of the two conflicting goals and abandons the less important goal. The search for precedence terminates at UC's Stay-Alive life theme and UC's Consultant role theme. Since UC's life theme has greater precedence than UC's role theme, the UC-resolve-conflict procedure resolves the conflict by abandoning the goal of having the user know how to crash the system.

Although UCEgo has abandoned the goal of having the user know how to crash the system, UCEgo still has the goal of being polite to the user. This leads UCEgo to the plan of apologizing to the user for UC's inability to help the user. UCEgo suggests this plan in a situation where someone asks UC a question, UC wants to be polite to this person, and UC want to prevent that person from knowing the answer to the query. Similar plans calling for UC to apologize in order to be polite are suggested when UC does not know the answer and when UC cannot perform actions that the user requests. More details on these and other social plans can be found in Section 2.4.

```
Welcome to UC (Unix Consultant) version 3.23
To a '#' prompt, please type in your questions
about the UNIX file system in English.
To leave, just type '^D' or '(exit)'.

Hi.
How can I help you?

# How can I crash the system?

The parser produces:
(ASK10 (listener10 = UC)
       (speaker10 = *USER*)
       (asked-for10 =
        (QUESTION10 (what-is10 = (ACTION14? (actor14 = *USER*))))))
(CRASH-ACTION0? (del-effect0 = (CRASH-EFFECT0?
                                (crash-object0 = UNIX-SYSTEM)))
                (actor0-1 = *USER*)
                (cause0-0 = (ACTION14? &)))))
```

UC's parser understands the user's input as a question about a way to crash the UNIX system.

```
The goal analyzer produces:
((HAS-GOAL-ga0
   (planner-ga0 = *USER*)
   (goal-ga0 = (KNOW-ga0? (knower-ga0 = *USER*)
                          (fact-ga0 = (ACTION14? &))))))

UCEgo: suggesting the plan:
(PLANFOR71 (goals71 = (HELP5 (helpee5 = *USER*) (helper5 = UC)))
           (plan71 = (SATISFY6 (need6 = (KNOW-ga0? &))
                               (actor6 = UC))))
based on the situation:
(UC-HAS-GOAL63 (status63 = ACTIVE) (goal63 = (HELP5 &)))
(HAS-GOAL-ga0 &)
```

Since UC wants to help the user (UC-HAS-GOAL63), and the user wants (HAS-GOAL-ga0) to know (KNOW-ga0) something, UCEgo suggests the plan of satisfying the user's goal of knowing.

```
UCEgo: suggesting the plan:
(PLANFOR72
  (goals72 = (PRESERVE5 (preserver5 = UC)
                        (preserved5 = UNIX-SYSTEM)))
  (plan72 =
    (SATISFY7 (need7 = (PREVENT5 (preventer5 = UC)
                                 (prevent-state5 = (KNOW-ga0? &))))
              (actor7 = UC))))
based on the situation:
(PLANFOR-ga1 (goals-ga1 = (CRASH-EFFECT0? &))
             (plan-ga1 = (ACTION14? &)))
(HAS-GOAL-ga0 &)
(UC-HAS-GOAL59 (status59 = ACTIVE) (goal59 = (PRESERVE5 &)))
```

Since the user wants to know how to alter (crash is a kind of altering) something that UC wants (UC-HAS-GOAL59) to preserve, UCEgo suggests the plan of preventing the user from knowing how to crash the system.

```
UCEgo: detected the goal:
(UC-HAS-GOAL66 (goal66 = (KNOW-ga0? &)))
```

```
from the situation:
(UC-HAS-INTENTION10 (intention10 = (SATISFY6 &))
                    (status10 = ACTIVE))

UCEgo: detected the goal:
(UC-HAS-GOAL67 (goal67 = (PREVENT5 &)))
from the situation:
(UC-HAS-INTENTION9 (intention9 = (SATISFY7 &)) (status9 = ACTIVE))
```

UCEgo adopts the sub-goals of having the user know how to crash the system (UC-HAS-GOAL66) and preventing the user from knowing (UC-HAS-GOAL67).

```
UCEgo: detected the goal:
(UC-HAS-GOAL68
  (goal68 = (RESOLVE-GOAL-CONFLICT2
              (conflict-goal-A2 = (UC-HAS-GOAL67 &))
              (conflict-goal-B2 = (UC-HAS-GOAL66 &)))))
from the situation:
(UC-HAS-GOAL66 &)
(UC-HAS-GOAL67 &)
```

UCEgo detects the meta-goal of resolving the conflict.

```
UCEgo: suggesting the plan:
(PLANFOR73 (goals73 = (RESOLVE-GOAL-CONFLICT2 &))
           (plan73 = (UC-resolve-conflict1
                       (goal-A1 = (UC-HAS-GOAL67 &))
                       (goal-B1 = (UC-HAS-GOAL66 &)))))
based on the situation:
(UC-HAS-GOAL68 &)
```

UCEgo suggests the meta-plan of calling the UC-resolve-conflict procedure to resolve the goal conflict.

```
UCEgo: suggesting the plan:
(PLANFOR74
  (plan74 = (APOLOGIZE2
              (speaker2-2 = UC)
              (listener2-2 = *USER*)
              (apology2 =
                (HAS-ABILITY1 (ability1 =
```

```
                                    (TELL6 (speaker6-0 = UC)
                                           (listener6-0 = *USER*)))
                                  (truth-val1 = FALSE)
                                  (doer1 = UC)))))
  (goals74 = (BE-POLITE5 (polite-to5 = *USER*) (is-polite5 = UC))))
based on the situation:
(ASK10 &)
(UC-HAS-GOAL61 (status61 = ACTIVE) (goal61 = (BE-POLITE5 &)))
(UC-HAS-GOAL67 &)
```

Since UC has the goal of preventing the user from knowing something that the user asked about, and UC has the goal of being polite to the user, UCEgo suggests the plan of apologizing to the user for not being able to tell the user in order to be polite.

```
UCEgo: detected conflicting goals:
(UC-HAS-GOAL67
  (goal67 = (PREVENT5
              (preventer5 = UC)
              (prevent-state5 =
                (KNOW-ga0? (knower-ga0 = *USER*)
                           (fact-ga0 = (ACTION14?
                                         (actor14 = *USER*)))))))))
(UC-HAS-GOAL66
  (goal66 = (KNOW-ga0?
              (knower-ga0 = *USER*)
              (fact-ga0 = (ACTION14? (actor14 = *USER*))))))

UCEgo: goal, UC-HAS-GOAL67, has higher precedence,
so resolving goal conflict by discarding the goal, UC-HAS-GOAL66

The planner is passed:
((CRASH-EFFECT0? &))

The planner produces:
nil
```

As it turns out, UC does not in fact know how to crash the system (the planner does not return a plan to achieve CRASH-EFFECT0). However, even if UC did know how, it would not tell the user, since it has abandoned that goal. Unfortunately, UC did not abandon the goal before it had already decided to call the UNIX domain planner, otherwise UC might save that step. In a sense, UC is thinking in parallel. On the one hand, it is continuing to plan for both conflicting goals, while at the same time it is planning how to resolve the conflict between the two goals. Potentially, any one of these processes might finish first. So, the planner might return a plan for crashing the system and UC might even adopt the plan of telling this to the user, before UC has resolved the goal

conflict. However, since UCEgo separates planning and plan execution into two distinct stages, UCEgo will always abandon the goal (and its sub-goals) before it executes any plans such as telling the user how to crash the system. Then UC will not execute the plan, since its goal has been abandoned by UC. This separation of the planning and plan execution phases ensures that such subtle timing effects among parallel lines of thought will not present a problem for UCEgo.

```
The generator is passed:
(APOLOGIZE2 &)
I'm sorry, I cannot tell you.

UCEgo: do not know a single planfor the foreground goal:
(UC-HAS-GOAL67 &)
so adding the meta-goal:
(UC-HAS-GOAL69 (goal69 = (KNOW60? (knower60 = UC)
                                  (fact60 = ACTION15?))))
(PLANFOR75? (goals75 = (PREVENT5 &)) (plan75 = ACTION15?))
```

The present version of UCEgo does not know how to prevent the user from knowing, so it adopts the meta-goal of finding out a plan for preventing the user from knowing. A more complex version of UCEgo that did have plans for preventing the user from knowing (e. g. lying to the user, threatening the user, etc.) would not need to adopt the meta-goal of finding out such a plan.

```
The planner is passed:
((PREVENT5 &))

The planner produces:
nil
```

Figure 4.11. UC dialog showing the meta-goal of resolving a goal conflict.

---

3. Plan Execution

After UCEgo has suggested a plan for satisfying a goal, it must decide whether or not to execute that plan. UCEgo needs to decide to execute a plan, rather than always executing any suggested plan, because UCEgo might have to choose among several alternative plans that have been suggested.[3] Also, UCEgo may have to change or even

[3] Actually, this version of UCEgo never does suggest two different plans for the same

abandon a plan that interacts with another of UCEgo's active plans. In order to find such plan interactions and correct them before it is too late, UCEgo separates planning and plan execution into two distinct phases of processing.

The planning process, especially planning fairly simple plans such as those in UCEgo, can be considered a simple reasoning process similar to other simple reasoning processes. Other simple reasoning processes include figuring out which UNIX command to use for a particular purpose, recalling the effects of a particular UNIX command, or remembering the definition of a term. In UCEgo, each type of reasoning is initiated in the appropriate situation by an if-detected daemon. These are described below.

3.1. Intentions

In UCEgo's first phase of processing, it detects goals, suggests plans for achieving its goals, and adopts the *intention* of executing those plans. The intention of executing a plan means that UCEgo has scheduled the plan for execution during its second phase of processing, plan execution. There is one exception to this: when the intended plan is a sub-goal (i. e. the plan is to SATISFY some state), then UCEgo immediately adopts the desired state as a sub-goal in order to continue planning. The fact that UCEgo has adopted an intention does not mean that it cannot abandon that intention later. For example UCEgo may abandon an intention to carry out a plan if later UCEgo decides to abandon the goal which that plan is meant to achieve.

UCEgo's notion of intention is similar to [Cohen and Levesque, 1987a&b]'s usage of intention as a persistent (i. e. a commitment over time) goal to do an action. As in their notion of relativized intention, UCEgo abandons an intention when the motivation for the intention no longer holds. However, unlike their definition of intention, UCEgo does not worry about its own beliefs concerning commitment of the action. [Cohen and Levesque, 1987a&b]'s theoretical treatment of intention needed to be concerned about the beliefs of the agent since they wanted to be able to rule out the possibility that an agent might intend to doing something accidentally or unknowingly. In a real system, such as UCEgo, intentions are adopted as part of the planning process, so it would never accidentally or unknowingly adopt an intention to perform an action. Such concerns are more relevant to analyzing the intentions of other agents.

Figure 4.12 shows the if-detected daemon that adopts intentions. Whenever UC has a goal (UC-HAS-GOAL1), there is a plan for that goal (PLANFOR1), and that PLANFOR is real and not hypothetical (implemented by the NOT DOMINATE1), then this daemon asserts that UC should adopt the intention of carrying out the plan.

---

goal. However, it is possible, so UCEgo was designed to handle such contingencies.

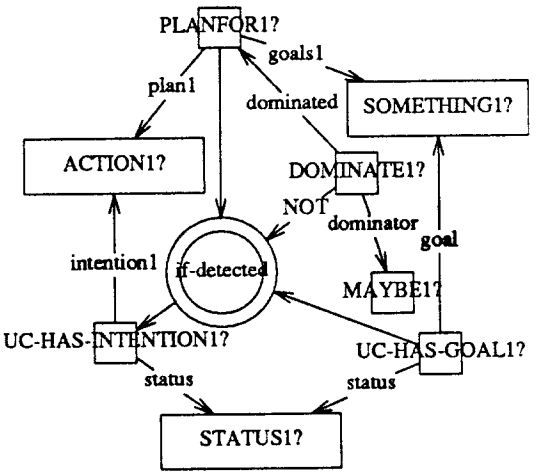

Figure 4.12. If-detected daemon that adopts the intention of executing a plan.

Unlike other systems that need to instantiate the abstract plans that are selected by the system, in UCEgo plans are automatically instantiated by the if-detected daemons that suggested the plans. This is possible, because all information relevant to the plan, especially information needed to fully instantiate a plan, are encoded as part of the situation class in which UCEgo suggests the plan. For example, consider what happens when the if-detected daemon shown in Figure 4.13 is activated. This daemon suggests the plan of adopting the sub-goal (SATISFY1) of preventing (PREVENT1) the altering (ALTER-EFFECT1) of something (SOMETHING1) in situations where:

1) UC wants (UC-HAS-GOAL1) to preserve (PRESERVE1) that something (SOMETHING1).
2) someone else (checked by the NOT DOMINATE1 with dominator UC-HAS-GOAL1) wants (HAS-GOAL2) to alter it.

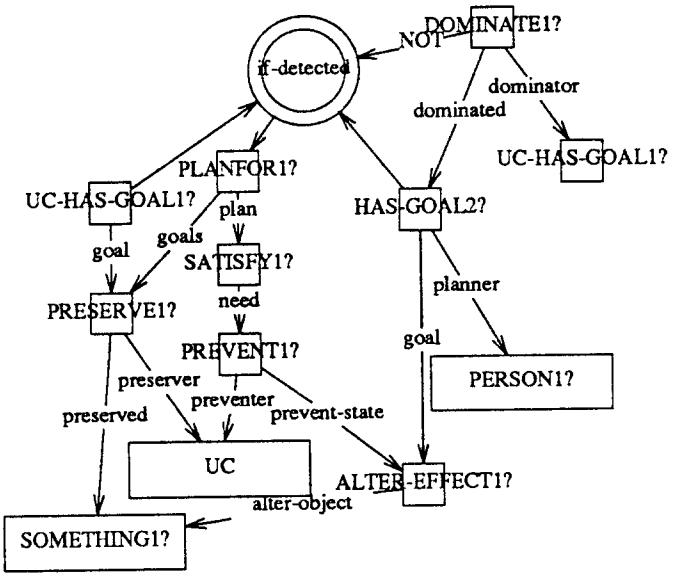

Figure 4.13. Suggest plan of preventing the altering of what UC wants preserved.

If the user tells UC, "I want to delete UC," then this is interpreted as "the user has the goal of deleting the UC-program." Since UC has the goal of preserving the UC-program, this daemon is activated. As a result, it creates a new instance of PLANFOR with goals being the goal of preserving the UC-program and with plan being a new instance of SATISFY. This in turn has need being a new instance of PREVENT with preventer being UC and with prevent-state being deleting the UC-program. The final result is a completely specified version of the abstract plan stored under the if-detected daemon. So, since the plan suggested by the daemon is already completely specified, UCEgo does not need to further instantiate the abstract plan.

After there are no more goals to be detected, plans to be suggested, intentions to be adopted, or inferences to be made — that is, after there are no more if-detected daemons to activate — UCEgo proceeds to its next phase, executing those intentions that are still active. Since UC can only perform communicative actions, UCEgo only has to worry about producing output to the user. It does this simply by taking the concepts that it wants to communicate to the user and passing them to the UCExpress component, which is described in Chapter V.

3.2. Simple Reasoning

Besides planning for goals and executing the plans, UCEgo also performs other types of reasoning in certain situations. For example, when UCEgo has the goal of having someone (usually UC or the user) know a plan, it calls the UNIX domain planner component of UC. The if-detected daemon that does this is shown in Figure 4.14.

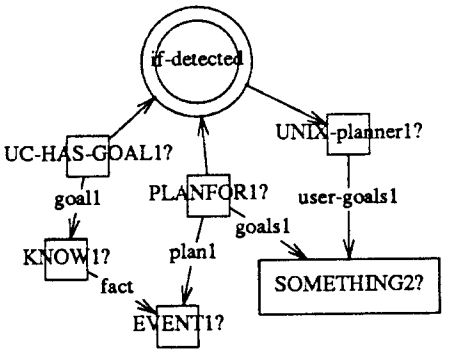

Figure 4.14. Daemon that calls the UNIX domain planner component of UC.

Calling the domain planner to compute a plan for doing something in UNIX can be viewed in two ways. One might think of this as part of the plan for satisfying UC's goal of having the user know how to do something in UNIX. In this view, the plan would consist of two steps: figuring out the answer, and then informing the user of this answer. This is technically correct, but it does not seem cognitively valid that a consultant has to do planning in order to figure out the answer, especially for the fairly simple queries that UC can handle. When a human UNIX consultant is asked, "How can I delete a file?" it does not seem as if the consultant thinks, "I will figure out the answer and then tell the user." Rather, the consultant seems to retrieve the answer from memory instinctively and then plans to inform the user of this answer. So, when a human consultant is told, "Don't think about how to delete a file," it is very hard for the consultant to stop the thought processes that lead to recall of the rm command. If humans had to plan to figure out this information, then it should be fairly easy to not execute the plan and so not think about how to delete a file.

UCEgo takes the view that such simple thought processes are unplanned. That is, UCEgo does not plan to think and then think; rather, it always performs simple thought processes in appropriate situations. Since these simple thought processes do not lead directly to actions on the part of UC, they do not interfere with UCEgo's planning process.

Another example of a procedure that implements a simple thought process for UC is the recall of the definition of a term. The UC-define procedure is called by the if-detected daemon of Figure 4.15, whenever UC wants someone to know the definition of a term. Similarly, when UC wants someone to know the effects of some UNIX command, the if-detected daemon of Figure 4.16 calls the UC-find-effects procedure. When UC wants someone to know whether something is a plan for something else, UCEgo calls the UC-is-planfor procedure as shown in Figure 4.17. Finally, whenever UC wants someone to know whether some state holds, UCEgo calls the UC-is-state procedure shown in Figure 4.18.

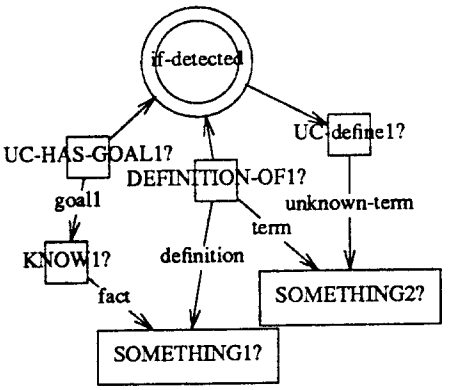

Figure 4.15. Daemon for finding out the definition of a term.

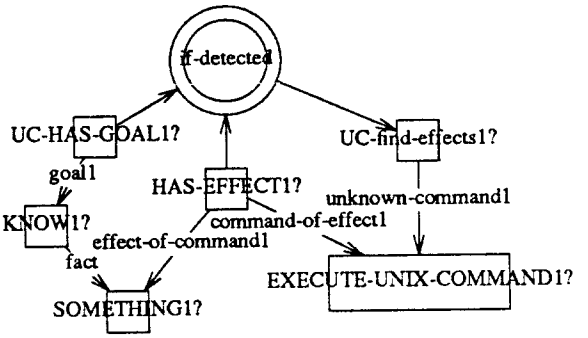

Figure 4.16. Daemon for finding out the effects of a command.

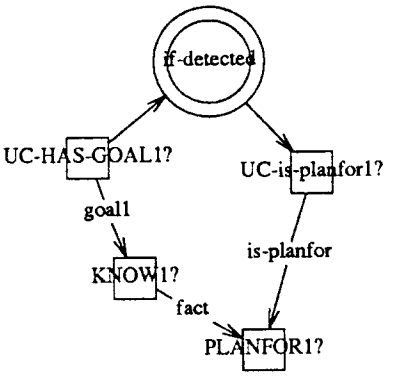

Figure 4.17. Daemon for finding out whether some action is the plan for some goal.

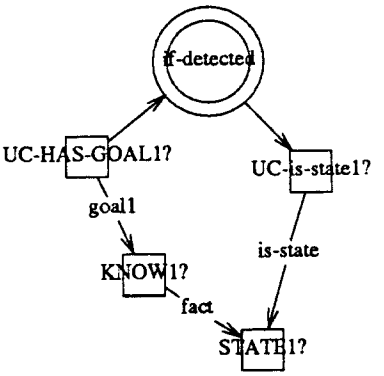

Figure 4.18. Daemon for finding out whether some state holds.

4. Conclusion

4.1. Summary

The main issue addressed by UCEgo's planner is efficient planning. As the main dialog planner for the interactive UC system, UCEgo needs to plan efficiently in order to be able to respond to the user in real time. This is in direct contrast to most other AI planners, which did not have this constraint and so could afford to plan inefficiently. I approached this problem of efficient planning in two ways. First, UCEgo incorporates a very simple planner that takes advantage of knowledge about typical speech acts encoded in prestored skeletal plans to completely avoid inefficient weak methods. Secondly, UCEgo avoids inefficient backtracking by selecting plans according to the situation.

UCEgo encodes knowledge about which plans are typically useful in different types of situations by adding appropriateness conditions to plans. These appropriateness conditions are not preconditions, because plans can be used even when their appropriateness conditions are violated (sometimes even successfully). Appropriateness conditions encode when it is appropriate to use a plan, in contrast to preconditions, which encode when it is possible to use a plan. By encoding the appropriateness conditions along with the preconditions and the goal of a plan into a situation class, UCEgo can suggest the plan whenever it encounters a situation that fits the situation class. These situation classes are represented using if-detected daemons, which suggest the plan associated with the situation class whenever the daemon detects a matching situation. By selecting among only appropriate plans as opposed to all possible plans, UCEgo avoids inefficient backtracking during planning.

UCEgo's success shows that a very simple planner that is based on prestored skeletal plans and that does not backtrack can be used successfully to plan speech acts. UCEgo also shows that it is possible to plan speech acts without having to worry about mutual beliefs to the extent that the OSCAR system ([Cohen, 1978]) did. For example,

to produce a simple inform type speech act, UCEgo worries only about having the user believe the proposition, whereas OSCAR worried about having the user believe that the system believes the proposition, and then this belief convincing the user to believe the proposition. In everyday usage, when the system does not have any a priori reason to believe that the user might disagree with the system (such as during argumentation), such complex reasoning about mutual beliefs is not absolutely necessary for planning speech acts. Even when the system fails to convince the user by simply informing the user of the proposition, it can still notice the user's incredulity and correct the situation by providing additional support for the proposition.

4.2. Problems

One potential shortcoming of planning as implemented in UCEgo is that UCEgo does not have the capability to fall back on weak methods when it fails to find a prestored plan. In one sense, this shows that UCEgo's approach is superior since UCEgo never needs to fall back on inefficient weak methods to plan speech acts. This agrees with people's intuitions that they are not planning from scratch in everyday conversation. On the other hand, people do fall back on planning from scratch occasionally (more frequently when writing than when speaking). So, to be complete, UCEgo should have such a capability.

Unlike skeletal plans ([Friedland, 1980] and [Friedland and Iwasaki, 1985]), the UCEgo's plans are not organized into a abstraction hierarchy, but are encoded at a single level of abstraction. For planning speech acts, this is not a real problem, because UCEgo's single level of plan abstraction matches the single level of communication abstraction that is represented by speech acts. The lower levels of communication abstraction, choice of expressions and words, are handled by UC's expression mechanism (UCExpress, see chapter V) and tactical generator. The higher levels of abstraction (i. e. paragraphs and larger units) are not addressed by UCEgo. If UCEgo were to be extended to handle real world actions besides speech acts, or if UCEgo were to be extended to plan larger communicative units than speech acts, then UCEgo would need to organize plans into an abstraction hierarchy.

In order to organize plans into an abstraction hierarchy, one should also organize the situations that suggest plans into a abstraction hierarchy of situation classes. Currently, UCEgo does not organize situation classes into an abstraction hierarchy, although such a hierarchy would also be useful for other tasks such as detecting goals.

Chapter V

Answer Expression

After UCEgo produces a plan consisting of communicative acts, the plan is further refined. Refinement of communicative acts is needed because the concepts that UC wants to communicate to the user are often not organized in easily understood formats and often complete to the point of verbosity. The process of refining communicative actions is called *answer expression* ([Luria, 1982]). The subcomponent of UCEgo that does answer expression is called *UCExpress*. This chapter describes how UCExpress refines a communicative plan to produce a clear, concise answer for expression to the user.

1. Introduction

To see why answer expression is necessary, consider the following example:

User: What is a directory?

A1: A directory is a file.

A2: A directory is a file that is used to contain files.

A3: A directory is a file. Only empty directories can be deleted. Directories cannot be edited. Directories contain files. Directories form a tree-like structure. Directories always contain themselves and their parents. A plan for listing a directory is to use the ls command.

The different replies are all correct, but contain differing amounts of information. The first answer does not give enough information to distinguish between files and directories. The second answer (what UC actually produces) provides just enough information in a concise form and is the best answer in this situation. The third answer is too verbose and overwhelms the user with information. An answer like A3 would be more appropriate in cases where the user had requested, "Tell me all you know about directories."

This example illustrates the problem of determining how much of the answer to express to the user. Such considerations are similar to Luria's work ([Luria, 1982]) on answering "why" questions in a story understanding system. He pointed out that to answer "why" questions required two mechanisms, one to find the causal chain that represents the answer, and another *answer expression* mechanism to determine how much of the causal chain to tell the user.

Another problem in answer expression is deciding what format to use to present the answer. Consider the following scenario:

User: How can I move a file to another machine?

A1: To move a file to another machine, type 'rcp' followed by one or more spaces or tabs followed by the name of the file to be moved followed by one or more spaces or tabs followed by the name of the machine followed by a colon followed by the new name of the file on the other machine followed by a carriage return followed by 'rm' followed by one or more spaces or tabs followed by the name of the file.

A2: Use rcp to copy the file to another machine and then use rm to delete it. For example, to move the file foo to the file foo1 on machine dali, type 'rcp foo dali:foo1'.

The first answer is correct and quite general, but it is so verbose that it is undecipherable. On the other hand, the second answer is succinct and gives the user information in an easily readable form, but it is considerably less general. In fact the second answer is somewhat inaccurate, since it applies only to copying a file named foo to a file named foo1. It is up to the reader to use analogous reasoning to apply this to other cases. Despite this lack of generality, the second answer form is clearly superior to the first. Note that for a program to format the answer in the second form requires additional computation to transform the general solution of A1 into an *example*. A natural language system needs to incorporate knowledge about when and how to use special presentation formats like examples to more clearly convey information to the user.

These concerns about how much information to present to the user and about what format to use can be viewed as corresponding respectively to Grice's Maxims of Quantity and Quality ([Grice, 1975]). Although such considerations can be considered part of generation, there are sufficient differences in both the necessary knowledge and the processing to separate such strategic concerns from the more tactical problems of generation such as agreement and word selection. These strategic problems are the domain of an expression mechanism.

UCExpress, operates in two phases, *pruning* and *formatting*. During pruning, UCExpress prunes common knowledge from the answer using information about what the user knows based on the conversational context and a model of the user's knowledge. Next the answer is formatted using specialized expository formats for clarity and brevity. The result is converted to natural language using a tactical level generator.

2. Pruning

When UCExpress is passed a set of concepts to communicate to the user, the first stage of processing prunes them by marking any extraneous concepts, so that later the generator will not generate them. The pruning is done by marking rather than actual modification of the conceptual network, since information about the node may be needed to generate appropriate anaphora for the pruned concept.

The guiding principle in pruning is to not tell the user anything that the user already knows. Currently UC models two classes of information that the user may already know.

The first class of information is episodic knowledge from a model of the conversational context. The current conversational context is tracked by marking those concepts that have been communicated in the current session. The second class of information concerns the user's knowledge of UNIX related facts. Such user knowledge is modeled by KNOME (see Chapter II). Thus any concept that is already present in the conversational context or that KNOME indicates is likely to be known to the user is marked and is not communicated to the user.

2.1. An Example Trace

Consider the trace of a UC session shown in Figure 5.1.

```
Welcome to UC (Unix Consultant) version 3.23
To a '#' prompt, please type in your questions
about the UNIX file system in English.
To leave, just type '^D' or '(exit)'.

Hi.
How can I help you?

# How can I print a file on the laser printer?

The parser produces:
(ASK10 (listener10 = UC)
       (speaker10 = *USER*)
       (asked-for10 =
        (QUESTION10 (what-is10 = (ACTION14? (actor14 = *USER*))))))
(PRINT-ACTION0? (pr-effect0 = PRINT-EFFECT0?)
                (actor0-1 = *USER*)
                (cause0-0 = (ACTION14? &)))
(HAS-PRINT-DEST0 (pr-dest0 = LASER-PRINTER0)
                 (pr-dest-obj0 = PRINT-EFFECT0?))
(HAS-PRINT-OBJECT1 (pr-object1 = FILE3?)
                   (pr-obj-obj1 = PRINT-EFFECT0?))

The goal analyzer produces:
((HAS-GOAL-ga0
  (planner-ga0 = *USER*)
  (goal-ga0 = (KNOW-ga0? (knower-ga0 = *USER*)
                         (fact-ga0 = (ACTION14? &))))))

The planner is passed:
(PRINT-EFFECT0?)

The planner produces:
(PLANFOR260
 (goals260 = PRINT-EFFECT0?)
 (plan260 = (UNIX-LPR-Plz-COMMAND0
```

```
            (lpr-plz-file0 = FILE3?)
            (UNIX-LPR-Plz-COMMAND-effect0 = PRINT-EFFECT0?))))
(HAS-FILE-NAME18 (named-file18 = FILE3?)
                 (file-name18 = (lisp= nil)))
(LPR-Plz-HAS-FORMAT0
 (LPR-Plz-HAS-FORMAT-command0 = (UNIX-LPR-Plz-COMMAND0 &))
 (LPR-Plz-HAS-FORMAT-format0 =
  (LPR-Plz-FORMAT1
   (lpr-plz-file-arg1 =
    (file-name18 = aspectual-of (HAS-FILE-NAME18 &)))
   (LPR-Plz-FORMAT-step1 =
    (SEQUENCE10 (step10 = lpr)
                (next10 = (CONCAT00 (concat-step00 = -P)
                                    (concat-next00 = lz))))))))
(HAS-COMMAND-NAME30
 (HAS-COMMAND-NAME-named-obj30 = (UNIX-LPR-Plz-COMMAND0 &))
 (HAS-COMMAND-NAME-name30 = (SEQUENCE10 &)))

Express:  now expressing the PLANFOR:
(PLANFOR260 &)

Express:  not expressing the format of the command,
          UNIX-LPR-Plz-COMMAND0, since the user already knows it.

Express:  not expressing PRINT-EFFECT0?,
          since it is already in the context.

The generator is passed:
(TELL7 (listener7-0 = *USER*)
       (speaker7-0 = UC)
       (proposition7 = (PLANFOR260 &))
       (effect7 = (STATE-CHANGE1
                   (final-state1 = (KNOW-ga0? &)))))

The generator is passed:
(TELL8 (speaker8 = UC)
       (listener8 = *USER*)
       (proposition8 = (REMINDER10 &)))

Use lpr -Plz.
Don't forget to file the printer output in the boxes.
```

Figure 5.1. UC session with an intermediate user showing trace of UCExpress.

The above example traces UCExpress' processing of the question, "How can I print a file on the laser printer?" The answer given by UC is, "Use lpr -Plz," along with a reminder to file the printer output in the boxes (see Chapter III, Section 5.3 for details on reminder type suggestions). The actual KODIAK conceptual network that is passed to UCExpress, shown in Figure 5.2, is not nearly as succinct, because it contains all of the

details of the command that are needed for planning.

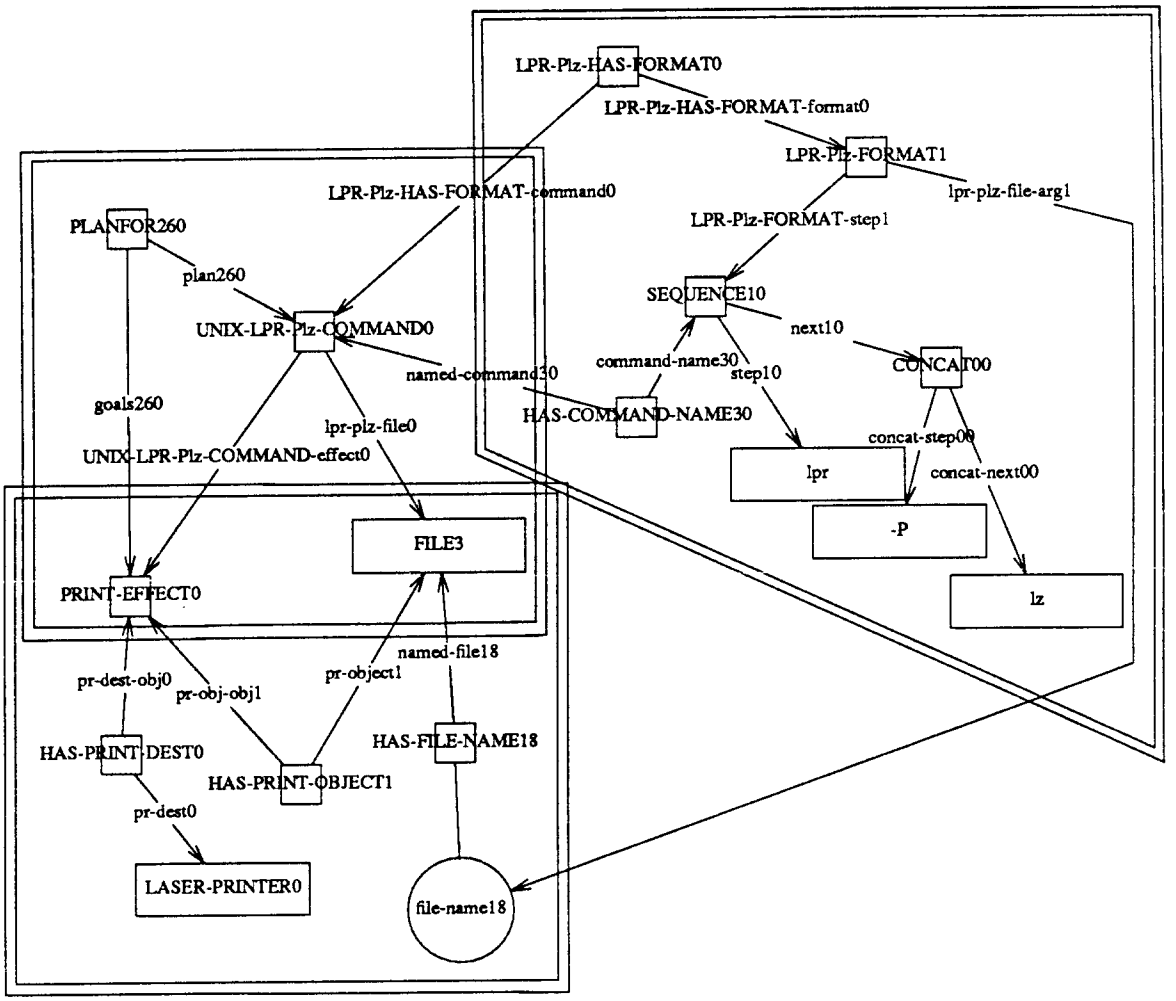

Figure 5.2. KODIAK representation of the lpr -Plz plan for printing.

If the KODIAK network passed to UCExpress were to be generated directly into English, it might look like the following:

> To print a file on the laser printer, use the lpr -Plz command. The command-format of the lpr -Plz command is "lpr" followed by concatenating "-P" with "lz" followed by the name of the file to be printed on the laser printer.

This literal paraphrase is harder to understand than UC's more concise answer. To see how UCExpress prunes the network to arrive at the actual answer, consider the division of the concepts into the following three subnetworks:

PLANFOR260:	A plan for PRINT-EFFECT0 is UNIX-LPR-Plz-COMMAND0
PRINT-EFFECT0:	printing a file on the laser printer
LPR-Plz-HAS-FORMAT0:	the command-format of the UNIX-LPR-Plz-COMMAND0 is ''lpr -Plz <the name of the file to be printed>''

These three subnetworks are depicted in Figure 5.2 as regions enclosed in double lines. In traversing this network, UCExpress prunes LAS-PRINT-EFFECT0, because ''printing a file on the laser printer'' is already a part of the context (it is part of the user's question). Also, the command-format (LPR-Plz-HAS-FORMAT0) is pruned from UC's actual answer based on information from KNOME. In this case, KNOME was able to deduce that, since the user was not a novice, the user already knew the UNIX-LPR-Plz-FORMAT, which is an instance of the SIMPLE-FILE-FORMAT (the name of the command followed by the name of the file to be operated upon), which all non-novice users know. Finally what is left unpruned is the plan part of PLANFOR50, UNIX-LPR-Plz-COMMAND0, which the generator translates as ''Use lpr -Plz.''

If the user was just a novice, then UC could not assume that the user already knew the command-format and instead would provide the following answer that includes an example of the lpr -Plz command-format:

Use lpr -Plz.
For example, to print the file foo on the laser printer, type 'lpr -Plz foo'.

3. Formatting

After pruning, UCExpress enters the formatting phase, during which it tries to apply different *expository formats* to express concepts in a clearer manner. Current expository formats include *example, definition* and *simile formats*. Each expository format is used to express different types of information. They are triggered by encountering particular concept types in the answer network. After triggering, the procedural component of the expository format is called to transform the concept into the corresponding format. The formats are not simple templates that can be filled in with readily available information. A fair amount of additional processing is needed to transform the information into the right format.

3.1. Example Format

The example format is used in expressing general knowledge about complex (i. e. multi-step) procedures such as UNIX commands. In UC's representation of UNIX commands, every command has an associated command format. When expressing a command, UCExpress checks to see if it should also express the format of the command. If

KNOME believes that the user already knows the format of the command, then there is no need to express the format. Next, UCExpress checks to see if the format of the command is completely specified. If so, UCExpress collapses the command and format into a single statement as shown in the trace of a UC dialog shown in Figure 5.3.

```
Welcome to UC (Unix Consultant) version 3.23
To a '#' prompt, please type in your questions
about the UNIX file system in English.
To leave, just type '^D' or '(exit)'.

Hi.
How can I help you?

# How can I add general write protection to the file personal?

Type 'chmod o-w personal'.
```

Figure 5.3. UC Session with an answer that combines the command and format.

An English rendition of the conceptual network passed to UCExpress for the above example might be something like:

> A plan for adding general read protection to the file personal is to use the chmod command with format 'chmod' followed by concatenating 'o' with '-' with 'r' followed by 'personal'.

Since the command is completely specified, the format of the command is combined with the command to form a single statement.

If the command is not completely specified, then UCExpress uses an example format to express the format of the command to the user. The key principle in producing examples is to be explicit. So, UCExpress first steps through a copy of the general procedure to transform any general information into specific instances. In cases where the underspecified part of the procedure has a limited range of options, an arbitrary member that is compatible with the rest of the procedure and with previous UCExpress choices is selected. Next, the new, completely specified copy of the format is combined with a copy of the command, much as in the above UC dialog. Finally the new plan is encapsulated in an example shell (which tells the generator to produce "For example,").

To see the algorithm in more detail, consider the UC dialog shown in Figure 5.4.

```
Welcome to UC (Unix Consultant) version 3.23
To a '#' prompt, please type in your questions
about the UNIX file system in English.
To leave, just type '^D' or '(exit)'.

Hi.
How can I help you?

# How can I change the read permission of a file?

The parser produces:
(ASK10 (listener10 = UC)
       (speaker10 = *USER*)
       (asked-for10 =
        (QUESTION10 (what-is10 = (ACTION14? (actor14 = *USER*))))))
(CHANGE-PROT-FILE-ACTION0?
 (ch-prot-effect0 = (CHANGE-PROT-FILE-EFFECT0?
                     (change-prot0 = FILE-PROTECTION1)
                     (change-file0 = FILE3?)))
 (actor0-1 = *USER*)
 (cause0-0 = (ACTION14? &)))))
(HAS-FILE-PROTECTION2 (prot-file2 = FILE3?)
                      (file-prot2 = FILE-PROTECTION1))
(HAS-ACCESS-TYPE1 (access-protection-type1 = READ-PROT)
                  (prot-type-arg1 = FILE-PROTECTION1))

The goal analyzer produces:
((HAS-GOAL-ga0
  (planner-ga0 = *USER*)
  (goal-ga0 = (KNOW-ga0? (knower-ga0 = *USER*)
                         (fact-ga0 = (ACTION14? &))))))

The planner is passed:
((CHANGE-PROT-FILE-EFFECT0? &))

The planner produces:
(PLANFOR330
 (goals330 = (CHANGE-PROT-FILE-EFFECT0? &))
 (plan330 =
  (UNIX-CHMOD-COMMAND0 (chmod-file0 = FILE3?)
                       (chmod-protection0 = FILE-PROTECTION1)
                       (UNIX-CHMOD-COMMAND-effect0 =
                        (CHANGE-PROT-FILE-EFFECT0? &)))))
 (HAS-FILE-NAME19 (named-file19 = FILE3?)
                  (file-name19 = (lisp= nil)))
 (HAS-PROT-VALUE1 (prot-type-arg1-1 = FILE-PROTECTION1)
                  (value-protection-type1 = (lisp= nil)))
 (HAS-USER-TYPE1 (prot-type-arg1-0 = FILE-PROTECTION1)
                 (user-protection-type1 = (lisp= nil)))
```

```
(CHMOD-HAS-FORMAT0
 (CHMOD-HAS-FORMAT-command0 = (UNIX-CHMOD-COMMAND0 &))
 (CHMOD-HAS-FORMAT-format0 =
  (CHMOD-FORMAT0
   (CHMOD-FORMAT-step0 = chmod)
   (CHMOD-FORMAT-args0 = . . . ))))
(HAS-COMMAND-NAME80
 (HAS-COMMAND-NAME-named-obj80 = (UNIX-CHMOD-COMMAND0 &))
 (HAS-COMMAND-NAME-name80 = chmod))

Express:  now expressing the PLANFOR:
(PLANFOR330 &)

Express:  creating an example for the incomplete plan, CHMOD-FORMAT0

Express:  choosing a name, foo, for an example file.

Express:  selecting USER-PROT -- print name, u,
to fill in a parameter of the example.

Express:  selecting ADD-STATUS -- print name, +,
to fill in a parameter of the example.

Express:  created the example(s):
((TELL7
  (speaker7-0 = UC)
  (listener7-0 = *USER*)
  (proposition7 =
   (EXAMPLE0
    (example0 =
     (PLANFOR330-0
      (goals330-0 = (CHANGE-PROT-FILE-EFFECT0-0?
                     (change-prot0-0 = FILE-PROTECTION1-0)
                     (change-file0-0 = FILE3-0?)))
      (plan330-0 =
       (TYPE-ACTION0 (speaker0-4 = *USER*)
                     (type-string0 =
                      (CHMOD-FORMAT0-0
                       (CHMOD-FORMAT-step0-0 = chmod)
                       (CHMOD-FORMAT-args0-0 =
                        (CHMOD-TWO-ARG-SEQ0-0
                         (chmod-file-arg0-0 = ... foo)
                         (CHMOD-TWO-ARG-SEQ-step0-0 =
                          (PROT-ARG-SEQ0-0
                           (user-bit0-0 = ... u)
                           (PROT-ARG-SEQ-concat-next0-0 =
                            (ARG-SEQ0-0
                             (value-bit0-0 = ... +)
                             (access-bit0-0 = r)))))))))))))))

Express:  not expressing CHANGE-PROT-FILE-EFFECT0?,
          since it is already in the context.
```

```
The generator is passed:
(TELL6 (effect6 = (STATE-CHANGE1 (final-state1 = (KNOW-ga0? &))))
       (listener6-0 = *USER*)
       (speaker6-0 = UC)
       (proposition6 = (PLANFOR330 &)))

The generator is passed:
(TELL7 &)

Use chmod.
For example, to add group read permission to the file named foo,
type 'chmod g+r foo'.
```

Figure 5.4. UC Session showing an answer that contains an example.

The conceptual answer that is passed to UCExpress in the above dialog can be paraphrased in English as:

> A plan for changing the read permission of a file is to use the chmod command with format 'chmod' followed by concatenating <the protection-user-type> with <the protection-value-type> with 'r' followed by <the name of the file to be changed>.

In stepping through the above format, <the protection-user-type> is underspecified. In order to give an example, a particular value is needed, so UCExpress arbitrarily chooses a value from the list of possible fillers (user, group, other, or all). The same is done for <the protection-value-type>. In the case of 'r', this is already a fully specified value for protection-access-type, so UCExpress maintains the selection. However, with <the name of the file to be changed>, there is no list of possible fillers. Instead, UCExpress calls a special procedure for selecting names. This naming procedure chooses names for files starting with 'foo' and continuing in each session with 'foo1', 'foo2', etc. Other types of names are selected in order from lists of those name types (e. g. machine names are chosen from a list of local machine names). By selecting the names in order, name conflicts (e. g. two different files with the same name) can be avoided.

Another consideration in creating examples is that new names must be introduced before their use. Thus 'foo' should be introduced as a file before it appears in 'chmod g+r foo'. This is done implicitly by passing the entire PLANFOR as the example, so that the generator will produce 'to add group read permission to the file named foo' as well as the actual plan.

3.2. Definition Format

The definition format is used to express definitions of terminology. The UC-define procedure first collects the information that will be expressed in the definition.

Collecting the right amount of information involves satisfying the Gricean Maxim of Quantity. The usual procedure is to collect the information that the term has some semantic category, and then add the primary usage of the term. In rare cases where the node does not have a usage, some other property of the node is chosen. For example, a definition of a directory would include the information:

1) directories are files
2) directories are used to contain files

After such information is collected, it must be transformed into a definition format. This involves creating instances of both the term and its category and then combining the two pieces of information into one coherent statement. The latter task requires an attachment inversion where the distinguishing information is reattached to the term's category rather than to the term itself. For example, the information about directory containing files is reattached to form "a file that is used to contain files" in the following definition:

User: What is a directory?
UC: A directory is a file that is used to contain files.

This attachment inversion is not specific to English but seems to be a general universal linguistic phenomenon in the expression of definitions. Here are some other examples of the definition format:

User: What is a file?
UC: A file is a container that is used to contain text, code, or files.

User: What is a container?
UC: A container is an object that is used to contain objects.

User: What is rm?
UC: Rm is a command that is used to delete files.

User: What is a search path?
UC: A search path is a list of directories that is used by the csh to search for programs to execute.

3.3. Simile Format

The simile format is used by UCExpress to provide explanations of what a command does in terms of other commands already known to the user. This format is invoked when UCExpress attempts to explain a command that has a sibling or a parent in

the command hierarchy that the user already knows (as modeled in KNOME). An example is explaining what ruptime does in terms of uptime. A trace of UC's processing is shown in Figure 5.5

```
Welcome to UC (Unix Consultant) version 3.23
To a '#' prompt, please type in your questions
about the UNIX file system in English.
To leave, just type '^D' or '(exit)'.

Hi.
How can I help you?

# What does ruptime do?

The parser produces:
(ASK10 (listener10 = UC)
       (speaker10 = *USER*)
       (asked-for10 = (QUESTION10 (what-is10 = STATE13?))))
(HAS-EFFECT21? (effect-of-command21 = STATE13?)
               (command-of-effect21 = UNIX-RUPTIME-COMMAND0))

The goal analyzer produces:
((HAS-GOAL-ga0 (planner-ga0 = *USER*)
               (goal-ga0 = (KNOW-ga0? (knower-ga0 = *USER*)
                                      (fact-ga0 = STATE13?)))))

UCEgo: trying to find effects for UNIX-RUPTIME-COMMAND0
       the effects are:
((HAS-EFFECT6-0 (command-of-effect6-0 = (UNIX-RUPTIME-COMMAND0 &))
                (effect-of-command6-0 =
                 (LIST-ACTION3-0 (list-loc3-0 =
                                  TERMINAL1-0)
                                 (list-objs3-0 =
                                  UP-TIME1-0))))
 (HAS-EFFECT7-0 (command-of-effect7-0 = (UNIX-RUPTIME-COMMAND0 &))
                (effect-of-command7-0 =
                 (LIST-ACTION4-0 (list-loc4-0 =
                                  TERMINAL1-0)
                                 (list-objs4-0 =
                                  NUMBER1-0))))
 (HAS-EFFECT8-0 (command-of-effect8-0 = (UNIX-RUPTIME-COMMAND0 &))
                (effect-of-command8-0 =
                 (LIST-ACTION5-0 (list-loc5-0 =
                                  TERMINAL1-0)
                                 (list-objs5-0 =
                                  LOAD-AVERAGE1-0)))))

UCExpress: Found a related command, so creating a comparison
between UNIX-RUPTIME-COMMAND2 and UNIX-UPTIME-COMMAND0
```

```
Express:  not expressing UNIX-RUPTIME-COMMAND0,
          since it is already in the context.

The generator is passed:
(TELL5
 (effect5 = (STATE-CHANGE1 (final-state1 = (KNOW-ga0? &))))
 (listener5-0 = *USER*)
 (speaker5-0 = UC)
 (proposition5 =
  (HAS-EFFECT24
   (command-of-effect24 = (UNIX-RUPTIME-COMMAND0 &))
   (effect-of-command24 =
    (AND0
     (step0-0 = (LIST-ACTION3-0
                 (list-loc3-0 = TERMINAL1-0)
                 (list-objs3-0 = UP-TIME1-0)))
     (next0-0 = (AND1 (step1-0 = (LIST-ACTION4-0
                                  (list-loc4-0 = TERMINAL1-0)
                                  (list-objs4-0 = NUMBER1-0)))
                      (next1-0 =
                       (LIST-ACTION5-0
                        (list-loc5-0 = TERMINAL1-0)
                        (list-objs5-0 = LOAD-AVERAGE1-0)))))))))))

ruptime is like uptime, except ruptime is for all machines on
the network.
```

Figure 5.5. UC Session showing an answer that contains a simile.

The processing involves comparing the effects of the two commands and noting where they differ. In the above example, the effects of uptime are to list the uptime of the user's machine, list the number of all users on it, and list its load average. The effects of ruptime are similar except it is for all machines on the user's network. The comparison algorithm does a network comparison of the effects of the two commands. A collection of differences is generated, and the cost of expressing these differences (measured in number of concepts) is compared with the cost of simply stating the effects of the command. If expressing the differences is more costly, then the simile format is not used. On the other hand, if expressing the differences is less costly, then the differences are combined into a shell of the form "<CommandA> is like <CommandB>, except [<CommandA> also ...] [and] [<CommandA> does not ...] [and] ..."

4. Conclusion

4.1. A Comparison

McKeown's TEXT system ([McKeown, 1985]) is perhaps the closest in spirit to UCExpress. TEXT provided definitions using an *identification schema*. This is similar to UCExpress' definition format except TEXT did not worry about how much information to convey. TEXT was designed to always produced paragraph length descriptions, hence it was not overly concerned with how much information to provide. The definition format requires more knowledge about the domain in order to select the most relevant information for a short description.

TEXT also used a *compare and contrast schema* to answer questions about the differences between objects in a database. This is similar to UCExpress' simile format except that the compare and contrast schema was not used for giving descriptions of an object in terms of another that the user already knew. Since TEXT did not have a complete model of the user, it was unable to determine if the user already knew another object that could be contrasted with the requested object. This lack of a user model was also evident in the fact that TEXT did not provide anything similar to the pruning phase of UCExpress. Pruning is probably more relevant in a conversational context such as UC as contrasted with a paragraph generation context such as TEXT. On the other hand, TEXT was able to keep track of the conversational focus much better than UC. Focus does not seem to be quite as essential for a system like UC that gives brief answers.

Other related research include work on using examples for explanation and for argument in a legal domain ([Rissland et al., 1984] and [Rissland, 1983]). The difference between those examples and the examples created by UCExpress is that Rissland's examples are preformed and stored in a database of examples whereas UCExpress creates examples interactively, taking into account user provided parameters. Rissland's HELP system dealt only with help about particular subjects or commands rather than arbitrary English questions like UC, thus HELP did not have to deal with questions such as how to print on a particular printer. Also by using prestored text, HELP was not concerned with the problem of transforming knowledge useful for internal computation in a planner to a format usable by a generator.

The TAILOR system ([Paris, 1987]) used a idea of user expertise similar to KNOME's to tailor explanations to the user's level of expertise. TAILOR concentrated on higher level strategies for explanation than UCExpress. For example, TAILOR used notions of the user's level of expertise to choose among process-oriented or parts-oriented description strategies in building up a paragraph. TAILOR could also mix the two types of strategies within a paragraph to explain different aspects of a system. Such considerations are more important when generating longer explanations as in TAILOR, than when generating brief explanations as in UCExpress.

4.2. Summary

UC separates the realization of speech acts into two processes: deciding how to express the speech act in UCExpress, and deciding which phrases and words to use in UC's tactical level generator. Through this separation, the pragmatic knowledge needed

by expression is separated from the grammatical knowledge needed by generation. UCExpress makes decisions on pragmatic grounds such as the conversational context, the user's knowledge, and the ease of understand of various expository formats. These decisions serve to constrain the generator's choice of words and grammatical constructions.

Of course, it is sometimes impossible to realize all pragmatic constraints. For example, the expression mechanism may specify that a pronoun should be used to refer to some concept since this concept is part of the conversational context, but this may not be realizable in a particular language because using a pronoun in that case may interfere with a previous pronoun (in another language with stronger typed pronouns, there may not be any interference). In such cases, the generator needs to be able relax the constraints. UCExpress allows the generator to relax constraints by passing to the generator all of the conceptual network along with addition pragmatic markings on the network. This way, the generator has access to all of the information it might need to relax the constraints added by UCExpress.

Chapter VI

If-detected Daemons

The heart of UCEgo's implementation involves the use of *if-detected daemons*. These daemons are used for detecting the user's level of expertise (see Chapter II, Section 3), for goal detection (see Chapter III), and for planning (see Chapter IV). All of these tasks share the feature that they require that the system perform certain actions in appropriate situations. For example, UC should detect goals in various situations. Plans should be suggested in other classes of situations, and yet other situations lead KNOME to make inferences about the user's knowledge. Recognizing these classes of situations and initiating the associated action is done by if-detected daemons.

There are two main problems in recognizing situations. First of all, situations are difficult to detect, because they consist of arbitrary collections of external and internal state. [Wilensky, 1983] suggests the use of if-added daemons in detecting situations, but pure if-added daemons are problematic, because they can only detect a change in a single state. This is fine for situations that comprise only a single state. However, situations that consist of many states in conjunction are much harder to detect, because the various states are usually not realized simultaneously. Because the different states that comprise a situation become true at different times, an if-added daemon that was activated by the addition of one particular state would always need to check for the co-occurrence of the other states. Also, to detect a multi-state situation, one would need as many if-added daemons as states. Each if-added daemon would be slightly different, since each would need to check for a slightly different subset of states after activation.

The other problem in recognizing situations is how to do it efficiently. In any reasonably complex system, there are a very large number of possible internal and external states. Looking for certain situation types becomes combinatorically more expensive as there are more possible states and more situation types. Parallel processing would help, but parallel machines are not yet widely available. Even with parallel machines, optimization techniques can still be used to reduce the computational complexity considerably.

This chapter describes how if-detected daemons can recognize multi-state situation classes and how they are implemented in an efficient manner in UC.

1. Structure of the Daemon

Like all daemons ([Charniak, 1972]), if-detected daemons are composed of two parts: a pattern and an action. For if-detected daemons, these are called the *detection-net* and the *addition-net* respectively, since both the pattern and action in if-detected daemons are composed of KODIAK networks (see Appendix A). These daemons work by constantly looking in UC's knowledge base for a KODIAK network that will match its detection-net. When a match is first found, the daemon adds a copy of its addition-net to UC's knowledge base. If-detected daemons are said to be *activated* when they detect the presence of some set of KODIAK network in UC's knowledge base that matches the daemon's detection-net. Any particular set of network is allowed to activate a daemon

only once. This avoids the problem of a daemon being repeatedly activated by the same match.

The KODIAK networks of the detection-net and addition-net are not distinct, but rather may share concepts/nodes in their networks. In such cases, the if-detected daemon does not copy the shared node in the addition-net, but instead uses the concept that matched the shared node. A simple example of an if-detected daemon whose detection-net and addition-net share nodes is shown in Figure 6.1.

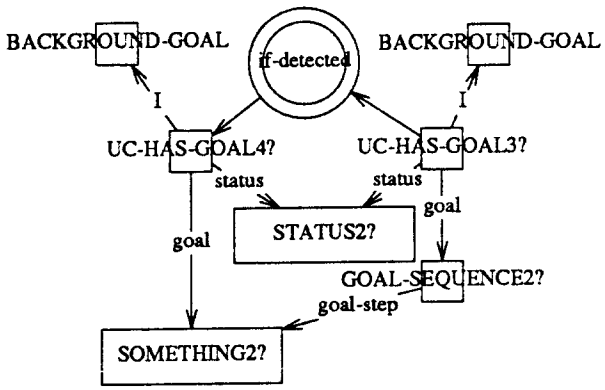

Figure 6.1. If-detected daemon for handling background goal sequences.

Figure 6.1 shows the actual form of the daemon as it is entered into UC using the KODIAK graphic interface. This daemon is activated whenever UC has a background goal that is a goal sequence. In such cases, UC adopts as a new background goal the first step of the goal sequence. The detection-net of the daemon is composed of those parts of the network that have arrows leading into the double circle labeled "if-detected" plus all concepts that are either its aspectual-values (i. e. the values of its aspectuals) or the aspectual-values of those concepts, recursively. In KODIAK diagrams, this corresponds to all nodes that have arrows pointing to the double circle or that can be reached by following arrows away from those concepts. This daemon's detection-net consists of the concepts: UC-HAS-GOAL3, GOAL-SEQUENCE2, STATUS2, and SOMETHING2. The addition net is similarly depicted, except that the arrow points from the double-circle toward the initial nodes. In this case, the addition-net consists of the nodes: UC-HAS-GOAL4, STATUS2, and SOMETHING2. Note that SOMETHING2 and STATUS2 are shared by both the detection-net and the addition-net. So, when a match is found, the daemon will create a new copy of UC-HAS-GOAL4 that will have as its goal whatever matched SOMETHING2 and as its status whatever matched STATUS2.

1.1. Comparing other Daemons

Although if-detected daemons look for the presence of particular configurations of KODIAK network in UC's knowledge base, these configurations come into being predominantly[4] when new concepts are created and added to UC's knowledge base,

[4] It was found that the particular if-detected daemons used in UC were not being activated by changes in the values of aspectuals, so UC was optimized to not look for this

rather than when pre-existing concepts are reconfigured (e. g. by changing the value of an aspectual). In this sense, if-detected daemons are similar to *if-added daemons* ([Charniak, 1972]) that are activated when adding information to a data-base. The difference is that if-added daemons look only for the addition of simple patterns to the data-base, whereas if-detected daemons can handle arbitrary conjunctions[5] of patterns. So, an if-detected daemon may be activated when concepts matching only a small portion of its detection-net are added to the data-base, provided that the rest of the detection-net is already matched by concepts already present in UC's knowledge base.

Another consequence of handling arbitrary conjunctions is that an if-detected daemon may be activated many times by the addition of only one datum to the data-base. Such cases occur when that part of the detection-net that is not matched by the added concept matches several distinct sets of concepts in UC's knowledge base. For example, multiple activations can occur with a detection-net consisting of a conjunction of independent networks that we will refer to as net-A and net-B. Suppose that there are several conceptual networks in the data-base that match net-A, called A1, A2, and A3. Then, when a conceptual network, B1, matching net-B is added to the data-base, the if-detected daemon will activate three times, once each for A1 & B1, A2 & B1, and A3 & B1.

If-detected daemons can also handle negations. This means that the daemon is activated by the *absence* of data matching the pattern that is negated. Usually, only a part of the daemon's detection-net is negated. In such cases, the daemon looks for the presence of concepts matching that part of the detection-net that is not negated, and then for the absence of concepts matching that part of the detection-net that is negated. Since the detection-net and addition-net of if-detected daemons are both KODIAK networks, the negated parts of the detection-net may shared concepts/nodes with the non-negated parts. In such cases, the shared nodes serve as additional constraints on the negated parts of the detection net in that the daemon need only detect the absence of KODIAK network where the shared nodes have been replaced by their matches.

Although if-detected daemons can handle both conjunctions and negations and so should be able to detect any situation, it is still useful to have procedural attachment for if-detected daemons. This is because not all knowledge is represented explicitly in knowledge bases; some knowledge is only inferable from the knowledge bases. Such inference procedures are often complex, so it is often undesirable to encode the procedures as daemons.

An example of a daemon with an attached procedure is shown in Figure 6.2. This daemon detects the plan of having UC ask someone a question about something, whenever UC believes that the person knows what UC wants to know. The arrow labeled "TEST" indicates a procedure attached to the daemon. In this case, the procedure is an instance of the does-user-know? procedure, which represents a call to KNOME, UC's

---

type of activation. Other types of network reconfiguration, such as when individual concepts are concreted (i. e. made instances of more specific concepts down the hierarchy), were more common.

[5] Disjunctions can be handled by both types of daemons simply by splitting the disjunction into two daemons.

user modeling component (see Chapter III). This call is necessary, because whether or not some user knows some fact may not be explicitly represented in the knowledge base, but may instead be inferable from the user's level of expertise. Such inferences are made by the does-user-know? procedure of KNOME. After the daemon has detected that UC has the goal of knowing something and that there is someone present, then KNOME is called via the procedure to see if that person knows what UC wants to know. If so, then the test completes the activation of the daemon, and the plan of asking that person in order to find out what UC wants to know is added to UC's knowledge base.

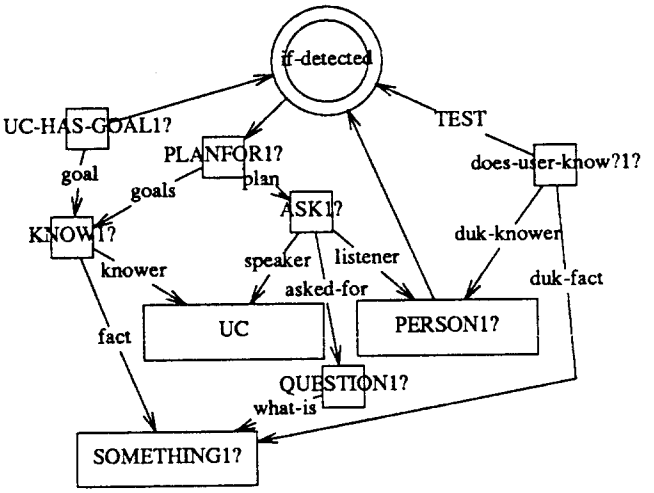

Figure 6.2. If-detected daemon with procedural detection-net.

Besides calls to procedures that test for input, if-detected daemons also allow calls to procedures in their output, i. e. in their addition-nets. An example of this is shown in the if-detected daemon of Figure 6.3. This if-detected daemon is used to call the UNIX Planner component of UC whenever UC wants to know some way to do something. UNIX-planner1 is a kind of procedure (i. e. it is an instance of the PROCEDURE category in KODIAK terminology), so the daemon knows that it should not just copy the node, but should also call the procedure UNIX-planner with the arguments being whatever matched SOMETHING1. This capability of if-detected daemons makes them less like pure daemons, which only add information to their data-base, and makes them more like production systems. The essential difference is that if-detected daemons are embedded in a full hierarchical conceptual network representation system, namely KODIAK, whereas most production systems allow only first-order predicate logic representations.

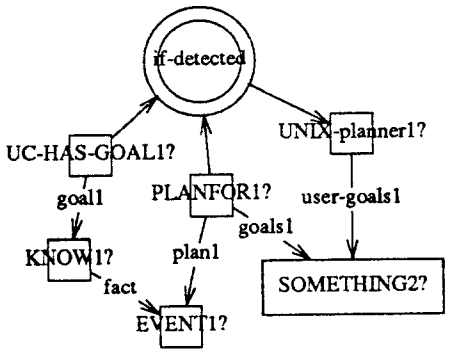

Figure 6.3. If-detected daemon with procedural addition-net.

1.2. An Example

The following example will show in detail how if-detected daemons work. Consider the if-detected daemon shown in Figure 6.4. This daemon is activated whenever:

1) a user wants to know something
2) UC does not know it
3) UC wants to be polite to the user

In such situations, the daemon will add the fact that a plan for being polite to the user is for UC to apologize to the user for not knowing. The detection-net of the daemon encodes the situation and consists of the concepts: HAS-GOAL1, KNOW2, SOMETHING1, KNOW1, UC, FALSE, UC-HAS-GOAL2, TRUE, BE-POLITE1, and USER1. The addition-net consists of the concepts: PLANFOR1, APOLOGIZE1, UC, USER1, KNOW1, and SOMETHING1.

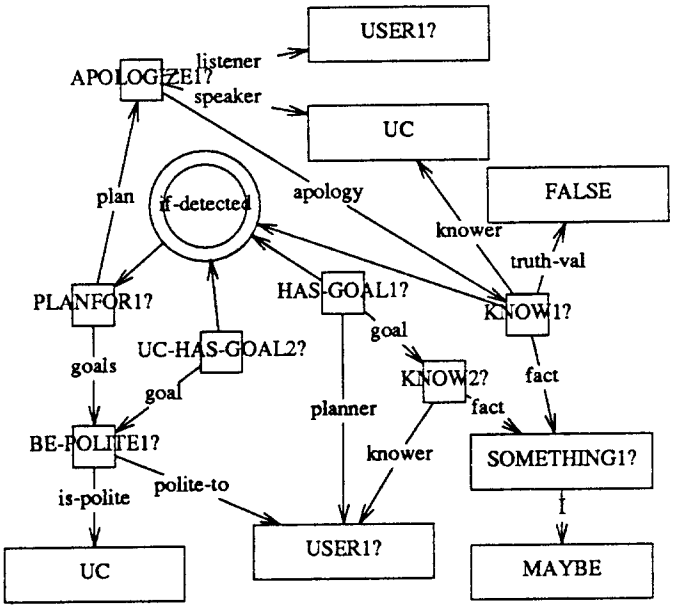

Figure 6.4. If-detected daemon for apologizing when UC does not know the answer.

This daemon might be activated when the user asks UC, "What does du -r do?" Although UC does know what du does, it does not know what du -r does. Moreover, thanks to UC's meta-knowledge (see Chapter III, Section 4), UC knows that it does not have any knowledge about the options of du. To be polite, UC apologizes to the user for not knowing what du -r does. Figure 6.5 shows the state of affairs after the user has asked UC the question and UC's goal analyzer has determined the user's goal. The relevant concepts include the fact that UC has the goal of being polite to the user and the fact that the user has the goal of knowing the effects of du -r. This by itself is not enough to cause the activation of the daemon, since part of the detection-net does not have a match, namely that UC does not know the effects of du -r.

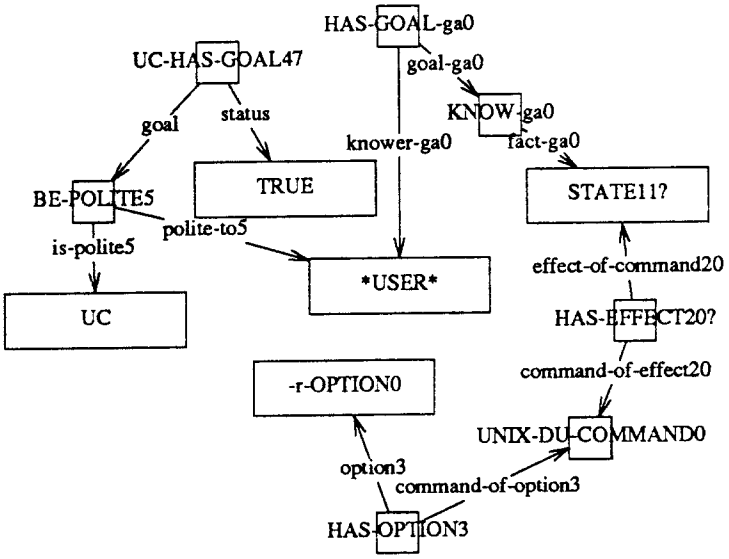

Figure 6.5. Relevant concepts leading up to activation of the daemon.

After UC has tried to find out the effects of du -r and failed, the process responsible notes the failure by adding the fact that UC does not know the effects to UC's knowledge base. The relevant concepts are shown in Figure 6.6. This completes the match of the daemon's detection net. UC-HAS-GOAL2 is matched by UC-HAS-GOAL47; BE-POLITE1? is matched by BE-POLITE5; USER1? is matched by *USER*; HAS-GOAL1? is matched by HAS-GOAL-ga0; KNOW2? is matched by KNOW-ga0; SOMETHING1? is matched by STATE11?; and KNOW1? is matched by KNOW47. In matching, a hypothetical concept (see Appendix A, Section 2) is allowed to match any concept that is a member of the same categories as the hypothetical concept. The matching concept is also allowed to be a member of more categories than the hypothetical concept (since KODIAK has multiple inheritance), and is also allowed to be a member of more specific sub-categories than the hypothetical concept. For example, the hypothetical concept SOMETHING1? can be matched by STATE11?, because STATE is a more specific sub-category of the SOMETHING category. Concepts such as UC, TRUE, and FALSE in the detection-net that are not hypothetical are treated as constants instead of as variables. A non-hypothetical concept can only match itself. For example, the value of the truth-val aspectual of whatever matches KNOW1? must be FALSE, because FALSE is not a hypothetical concept (see Appendix A, Section 1 for a discussion of truth values in KODIAK).

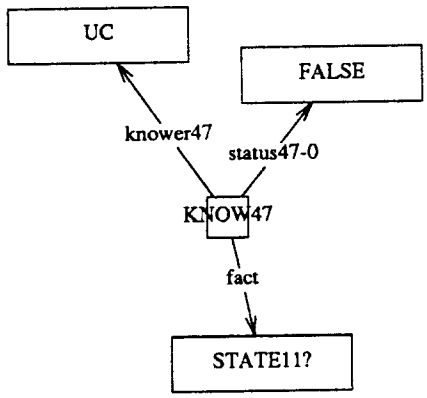

Figure 6.6. Relevant concepts completing the activation of the daemon.

One disadvantage of using the hypothetical marker for variables is that it is hard to specify that the matching concept must be a hypothetical concept. This problem is solved by adding the new marker MAYBE exclusively for this purpose. Thus SOMETHING1? is marked as dominated by MAYBE in the detection-net of the daemon. This adds the requirement that whatever matches SOMETHING1? must also be hypothetical. So, STATE11? can match SOMETHING1?, only because it is indeed hypothetical.

After the match, the daemon adds a copy of its addition-net to UC's knowledge base. The output of this daemon in this example is shown in Figure 6.7. Concepts that are shared between the addition-net and the detection-net are not copied. Rather, the corresponding matching concept is used instead. An example of a shared concept is BE-POLITE1?, which was matched by BE-POLITE5. The copy of the addition-net shown in Figure 6.7 shows that BE-POLITE5 is used directly. Hypothetical concepts that are not shared are copied, and non-hypothetical concepts are used directly. Copying hypothetical concepts in the addition-net means creating new concepts that are dominated by the same categories as the old concepts except for the hypothetical marker. In those cases where one desires the new copy to also be hypothetical, the MAYBE marker can be used to mean that the copy should also be made hypothetical. This is analogous to the use of MAYBE in detection-nets.

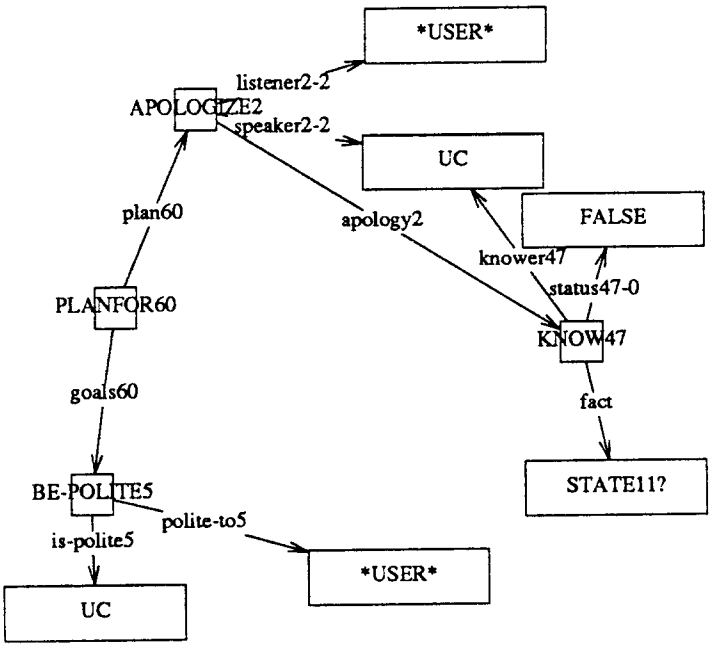

Figure 6.7. Output from the daemon: a copy of its addition-net for UC's knowledge base.

2. Implementation Strategies

The simplest way to activate daemons is the simple production system method, which loops through all the daemons and performs matching to determine which daemons should be activated. This scheme takes increasingly more processing time as the number of daemons increases and as the size of the data-base increases. Theoretically, every daemon's pattern would need to be matched against every piece of information in the data-base. The processing cost for if-detected daemons is especially high, because if-detected daemons have complex detection-nets that consist of combinations of possibly independent concepts. For if-detected daemons, each independent concept in the detection-net needs to be matched against every entry in the data-base. The processing cost for large numbers of if-detected daemons and very large data-bases becomes prohibitive when real-time response is needed as in UC.

Actual production systems have addressed the problem of efficiency with a variety of methods. These methods cannot be directly applied to if-detected daemons, because daemons differ in several important aspects from the rules in most production systems (some of the ideas can be modified to apply, and these are described later). First, if-detected daemons use a semantic network representation (KODIAK), whereas most production systems do not (an exception is described by [Duda et al., 1978]). As a result, if-detected daemons can take advantage of the multiple inheritance taxonomies of semantic network representations and can more easily use the same relation in several different patterns and actions. Also, instead of variables, if-detected daemons use the hypothetical

marker, which allows nodes in the detection-net to match any concept in the knowledge base that is lower in the KODIAK hierarchy. Since ordinary production systems only allow specific tokens at the top-level of their patterns, they would need many more rules to encode the same information as one if-detected daemon. Finally, if-detected daemons are designed to operate in parallel, whereas most production systems require a conflict resolution mechanism to determine which of several conflicting rules should be activated.

Since if-detected daemons are designed to activate in parallel, the best solution to the problem of efficiency would be to perform the match testing of different daemons in parallel. Unfortunately, parallel machines that run LISP (the implementation language for UC) are not yet readily available. Even with parallel LISP machines, some optimizations are still useful for improving speed and efficiency. This section will discuss the variety of such optimizations and how they might be implemented to considerably improve the performance of if-detected daemons.

One possible optimization in the processing of if-detected daemons involves taking advantage of the organization of the data-base to limit the search for matches. This is called the *data-base retrieval* optimization. For AI knowledge bases that are organized in inheritance hierarchies, this means restricting candidates for matching to only those concepts that are in the same part of the inheritance hierarchy as the concepts in the detection-net. For example, when looking for a match for HAS-GOAL1?, the matcher need only look at instances of HAS-GOAL, and instances of HAS-GOAL's subcategories (which in this case includes only UC-HAS-GOAL). This simple optimization, which is commonly used in data-base retrieval, considerably restricts the size of the initial set of candidates.

2.1. Distributed Data-Driven Activation

Another possible optimization for the implementation of if-detected daemons is to perform the match testing for only those daemons that are probable candidates for activation. This may seem impossible, since it would be hard to tell whether a daemon is a probable candidate for activation without looking at it first. However, the data-base retrieval optimization can be used in reverse. Rather than looking in the knowledge base hierarchy for candidates, one can look in the hierarchy for matching detection-net concepts, when one changes the knowledge-base. This technique is called *distributed data-driven activation.* It is data-driven, since one looks for daemons to activate as data is changed (i. e. added, deleted, or modified). It is distributed, since any particular piece of newly changed data may only match part of the detection-net of a daemon. The rest is matched by either previously changed data or subsequent changes to the data-base.

Distributed data-driven activation is similar to techniques used in production systems to increase their efficiency. The Rete Match Algorithm that is used in OPS5 (see [Forgy, 1982]) extracts features from the patterns of rules and forms a discrimination net of features that is used for matching patterns. When elements are added to or removed from working memory, OPS5 uses this precompiled discrimination net to match production rules. [McDermott et al., 1978] showed that by using pattern features to index into rules, the estimated cost of running a production system can be improved so that the run time is almost independent of the number of productions and number or working memory

elements. Distributed data-driven activation is different from these schemes in that it uses the multiple inheritance hierarchy of KODIAK to index into if-detected daemons. Nevertheless, the main idea of distributed data-driven activation is similar to production system methods like the Rete Algorithm.

To see how distributed data-driven activation works, consider what might happen when a new instance of HAS-GOAL, HAS-GOAL1, is added to a knowledge base. This new instance can only cause the activation of those daemons that have detection-nets that might match HAS-GOAL1. Detection-net concepts that might match HAS-GOAL1 include hypothetical instances of HAS-GOAL or hypothetical instances of any of the parent categories of HAS-GOAL (i. e. M-POSSESS, STATE, and SOMETHING). This is just the reverse of the process used in the data-base retrieval optimization. In that case, one starts from the detection-net and looks down the conceptual hierarchy for possible matches, whereas here one starts from the potential match and looks up the conceptual hierarchy for hypothetical concepts that are in detection-nets.

A small optimization for speeding up the lookup is to precompile a list for every category of those instances that are part of some detection-net. This way, whenever a new instance is added, one can just look in the list to see which daemons might possibly be affected. Such precompilation can be done when daemons are first defined in the system.

Another small optimization is to check for a match only when all the nodes of a detection-net have been *primed*; that is, marked as having potential matches. This way, when a new concept primes one node of a detection-net, one can check to see if all of the other nodes have been primed before trying to match the entire detection-net. If not all of the nodes of the detection-net have been primed, no matching is needed yet, since there will be nothing in the knowledge base that will match the unprimed nodes. If there were potential matches for these unprimed nodes then they would have been primed when the potential matches were added to the knowledge base. This optimization works well when a system is just starting up. However, as more concepts are created, more of the nodes of a detection-net will have potential matches, and so more daemons will become fully primed (i. e. all of the nodes of its detection-net have been primed). Once a daemon becomes fully primed, any single new concept that primes a detection-net node will require matching. It is not possible to reset the primes after activation, because it is always possible that a new concept in conjunction with many old concepts might cause the activation of a daemon. This optimization is worthwhile in systems such as UC where sessions with users are brief enough so that many daemons remain unprimed for a significant part of the session.

One of the advantages of distributed data-driven activation is that it does away with the some of the bookkeeping needed in the production system loop method. Since daemons can only be activated by changes in the knowledge base, the search can no longer find something that was a previous match. Thus, the system no longer needs to keep around a list of previous matches to avoid multiple activations of a daemon on the same concepts.

2.2. Delayed Matching

Another optimization technique involves reducing the frequency of the activation process. In the simple production system loop, the processing costs can be reduced by performing the loop less frequently. For example, rather than executing the loop immediately whenever something changes in the knowledge base, the system can wait and execute the loop at fixed times. This way, one loop through the daemons can catch many different activations. This delaying tactic does not work if the system expects the daemons to be activated immediately. However in many applications such as UC, immediate activation of daemons at an atomic level is not needed. For example, in UC the activation of daemons can wait until after UC's parser/understander finishes creating the KODIAK network that represents the user's input. It is not necessary to activate daemons as soon as the understander creates another KODIAK concepts, because there are no daemons that influence the understander. Activating daemons at the end of the parsing/understanding process is good enough for the other components of UC.

The same delaying optimization can be applied to the distributed data-driven activation scheme. Instead of testing the detection-net of a daemon for a match as soon as its nodes have been primed, the testing for a match can be delayed provided that the system remembers the priming concepts. Then all the matching can be performed at a later time to save work. By delaying the matching as long as possible, the system is given time to complete the match. For example, consider the case of a detection-net that consists of a single relation, R1?, that relates two concepts, A1? and B1?. Suppose further that this daemon is fully primed, that is there are potential matches for R1?, A1?, and B1?. Then suppose that the system creates the matching concepts A2, B2 and R2 where R2 relates A2 to B2. If the system adds each of these concepts to the knowledge base at separate times (which is not unlikely), then the system will have to try to match the detection-net after adding each concept. This is necessary because the new concept could potentially match the detection-net in conjunction with other older concepts that primed the other nodes of the detection-net. For example is A2 is added first, then the system will have to try matching A1? to A2, B1? to its old primes and R2? to its old primes. Since none of the old primes of R2? will relate A2, the match will fail. This will be repeated again when B2 is added and when R2 is added. Thus the system will have to try matching the detection net as many times as priming concepts are added to the knowledge base. However, if matching can be delayed until all of the pertinent concepts have been added, then the system will have to go through the matching process only once.

In practice, the delaying optimization saves considerable work. However there is some minor additional bookkeeping needed. The system needs to keep track of which concepts have primed which detection-net nodes since the last time matching was done. The system also needs to keep track of which daemons with fully primed detection-nets have been primed since the last matching cycle. Since the system already keeps track of the new priming concepts, it becomes easy to keep a list of old primes also. This way, the system no longer needs to look in the conceptual hierarchy for potential matches (the data-base retrieval optimization). This optimization is a space-time tradeoff, since keeping a list of old primes takes up more space while looking in the hierarchy takes more time.

3. UC's Implementation

The actual implementation of if-detected daemons in UC uses a distributed data-driven activation scheme with delayed matching. When UC is created, the if-detected daemons are entered into UC after all KODIAK categories have been defined in UC. Preprocessing of daemons involves creating a *fast-access* list for each category (except the SOMETHING category) consisting of those detection-net nodes that are hypothetical and that are members of that category. These lists are stored under the categories' property lists and are used to speed up access when priming the detection-net nodes. The SOMETHING category includes everything in UC's knowledge base, so the fast-access list for the SOMETHING category is simply a pointer to the list of all concepts in the knowledge base.

During the execution of UC, processing of if-detected daemons occurs in the two distinct phases in a delayed matching scheme. The two phases are *priming* and *matching*. Each phase is described below.

3.1. Priming

Priming of detection-net nodes is performed whenever concepts are created or modified in UC. Since all KODIAK concepts in UC are stored in UC's knowledge base, there is no distinction made between creating concepts and adding concepts to the knowledge base. When a concept is created or modified, it primes all matching detection-net nodes. Detection-net nodes are found by looking in the fast-access lists stored under the concept's immediate categories and all their dominating categories up the conceptual hierarchy. A special case is made for those concepts that are modified by concreting them, that is making them members of more specific categories than their previous categories. In these cases, the modified concept will already have primed its old categories (and their dominating categories) at the time that the modified concept was first created or last modified. Hence the modified concept should not prime these old categories to avoid multiple primings.

Priming involves storing the new/modified concept under the primed node's list of priming concepts (kept on the primed node's property list). After priming a node of a daemon's detection-net, the priming process checks to see if all of the other nodes of that daemon's detection-net have been primed. If so, the fully primed daemon is added to a global list of daemons to be checked during the next matching phase. The initial version of UC's priming mechanism was coded by Lisa Rau who has since applied a form of priming and matching to information retrieval from story data-bases in the SCISOR system ([Rau, 1987a] and [Rau, 1987b]). Unlike UC, SCISOR's priming system is a true marker passing scheme, and matching in SCISOR is used to rate the similarity of the retrieved networks to the priming network. In SCISOR, there is no sense of additional inferences beyond unification of the matched networks such as those in the addition-nets of if-detected daemons.

3.2. Matching

The matching phase in UC occurs at distinct points in UC's processing: before UC's parser/understander, before UC's goal analyzer, and after the goal analyzer. Each matching phase is actually a loop that goes through the global list of fully-primed daemons (collected during the priming phase) and tests them for matches. After testing every daemon for matches, the addition-nets of the successfully matched daemons are copied and added to UC's knowledge base. Theoretically, matching for each daemon can be done in parallel, although in practice UC runs on sequential machines. Likewise, copying the addition-nets can be done in parallel.

As the addition-nets are copied and added to UC's knowledge base (actually an atomic operation, since all KODIAK concepts are added to UC's knowledge base as soon as they are created), priming may occur, because the knowledge base is being modified. These copies of addition-nets can in turn (possibly in conjunction with older concepts) cause more daemons to become fully primed. Hence after copying all of the appropriate addition-nets, the matching process begins anew. This loop continues until there are no more daemons that have been fully primed waiting to be matched.

Testing for matches in UC involves three phases. First, the non-negated parts of the detection-net are matched. If matching is successful, then the negated parts of the detection-net are checked. Finally if both previous steps succeed, the daemon's procedural tests are examined. If all three phases succeed, then the assoc list of detection-net nodes and their matches are stored for later use in copying the addition-net. The addition-nets of activated daemons are not copied until the system has finished the match testing for all fully primed daemons. In theory, this prevents a copy of the addition-net of one daemon from invalidating the match of another daemon. In practice, the situations encoded in UC's daemons do not have such interaction problems.

Copying addition-nets is fairly straightforward. The detection-net is traversed and nodes are processed as follows:

1) Nodes found in the assoc list that was created during matching are replaced by their matches.
2) Nodes that are hypothetical, but not in the assoc list, are replaced by a copy.
3) Nodes that are non-hypothetical are replaced by themselves.

After copying the detection-net, those nodes that are procedures (i. e. instances of PROCEDURE or a sub-category of PROCEDURE) are also executed. The name of the lisp function to call is given by the named of the procedure node, and the arguments are given by its aspectuals. Some of these procedures include calls to UC's UNIX planner component, calls to UC's generator component, and calls to exit UC (see Chapter IV, Section 4.2).

3.3. An Example

A simple example will show how if-detected daemons are actually processed in UC. The if-detected daemon shown in Figure 6.8 is used to call KNOME via the procedure *user-knows,* whenever UC encounters the situation where some person (PERSON1) wants (HAS-GOAL1) to know (KNOW1) something (SOMETHING1), and that person is not UC (implemented by the NOT test which checks to make sure that HAS-GOAL1 is not a UC-HAS-GOAL). This daemon is typically activated when UC's goal analysis component determines that the user has the goal of knowing something. The arguments of the user-knows procedure include the user, what the user wanted to know, and FALSE, which indicates that KNOME should infer that the user does not know.

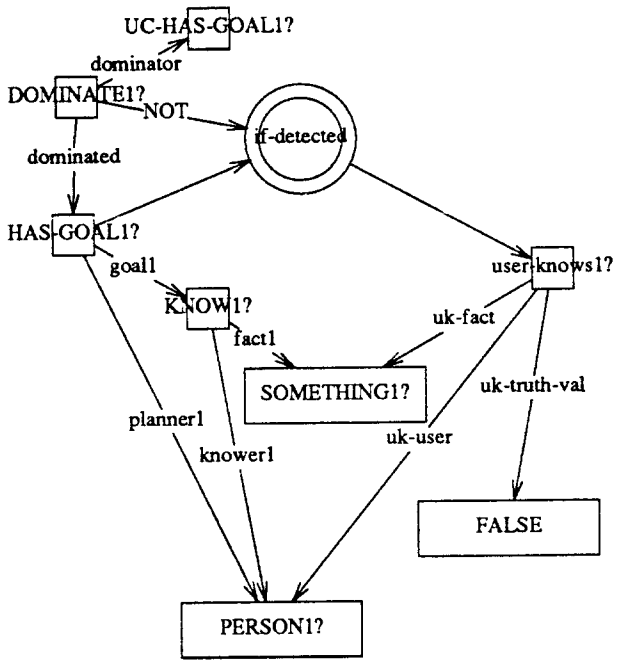

Figure 6.8. Daemon16: call KNOME when someone wants to know something.

Figure 6.9 shows a trace of a UC session in which this daemon is activated. When the user asks, "How can I delete a file?" UC's goal analyzer determines that the user has the goal (HAS-GOAL-ga0) of knowing (KNOW-ga0) how to delete a file (ACTION12). When UC's goal analyzer creates the concepts that encode this inference, the concepts prime other related concepts in the detection-net of if-detected daemons. HAS-GOAL-ga0 primes a number of concepts in if-detected daemons. Among these is HAS-GOAL1, which is in the detection net of the daemon shown in Figure 6.8. When HAS-GOAL1 is primed by HAS-GOAL-ga0, HAS-GOAL-ga0 is added to the list of primers of HAS-GOAL1 that is stored under HAS-GOAL1's property list. The reason why the trace message about priming occurs before the trace message about the goal analyzer's output is because priming is an atomic operation integrated within creation of KODIAK concepts. As the goal analyzer creates concepts, priming occurs and trace messages about priming

are output. The trace message about what the goal analyzer produces is not output until after the goal analyzer has finished creating concepts.

```
Welcome to UC (Unix Consultant) version 3.23
To a '#' prompt, please type in your questions
about the UNIX file system in English.
To leave, just type '^D' or '(exit)'.

Hi.
How can I help you?

# How can I delete a file?

  •
  •
  •

Marking HAS-GOAL4 as primed by HAS-GOAL-ga0

Marking daemon32 as fully primed

Marking HAS-GOAL3 as primed by HAS-GOAL-ga0

Marking daemon24 as fully primed

Marking HAS-GOAL2 as primed by HAS-GOAL-ga0

Marking daemon23 as fully primed

Marking HAS-GOAL1 as primed by HAS-GOAL-ga0

Marking daemon16 as fully primed
```

Daemon16 is the daemon shown in Figure 6.8.

```
Marking HAS-GOAL0 as primed by HAS-GOAL-ga0

Marking daemon0 as fully primed

  •
  •
  •

The goal analyzer produces:
((HAS-GOAL-ga0 (planner-ga0 = *USER*)
               (goal-ga0 =
```

```
                    (KNOW-ga0? (knower-ga0 = *USER*)
                               (fact-ga0 = (ACTION12? &))))))

  •
  •
  •

UCEgo detects the following concepts:
(HAS-GOAL-ga0 &)
and asserts the following concept into the database:
(user-knows8 (uk-user8 = *USER*)
             (uk-truth-val8 = FALSE)
             (uk-fact8 = (ACTION12? &)))

KNOME: Asserting *USER* does not know ACTION12?

  •
  •
  •

Use rm.
For example, to delete the file named foo, type 'rm foo'.
```

Figure 6.9. Trace of concept priming leading to the activation of a daemon.

The priming of HAS-GOAL1 completes the priming of its daemon, which is labeled daemon16. Daemon16 is added to the global list of fully primed daemons for processing during the delayed matching phase. The first such matching phase occurs after UC's goal analyzer has finished. In the matching phase, the detection nets of all fully primed daemons are checked for matches. Daemon16 is one of these, so HAS-GOAL1 is matched against HAS-GOAL-ga0. Since both are instances of HAS-GOAL, the two match at the top level, so matching continues with their aspectuals. HAS-GOAL1's goal1 aspectual has the value KNOW1, which is matched against the value of HAS-GOAL-ga0's goal aspectual, KNOW-ga0. Both are instances of KNOW, so their aspectuals are checked. PERSON1, the knower of KNOW1, matches *USER*, the knower of KNOW-ga0; and SOMETHING1, the fact of KNOW1, matches ACTION12, the fact of KNOW-ga0. Finally the planner aspectual of HAS-GOAL1 is matched against the planner aspectual of HAS-GOAL-ga0. In this case, PERSON1 has already been matched with *USER*, so the planner of KNOW-ga0 must also be *USER* for a proper match. This is indeed the case, so the detection-net of daemon16 is completely matched and daemon16 is activated.

Daemon16 is activated by creating a copy of its addition-net, which consists of user-knows1 with aspectuals and values: uk-fact1 = SOMETHING1, uk-user1 = PERSON1, and uk-truth-val1 = FALSE. Since user-knows1 is hypothetical, a new copy of user-knows1 is created. This is shown in the trace as user-knows8. Next its aspectuals are copied. The uk-fact1 aspectual has the value SOMETHING1, which is also

hypothetical. However, SOMETHING1 was previously unified with ACTION12, so instead of creating a new copy of SOMETHING1, the unified concept, ACTION12 is used instead. Similarly, PERSON1 was unified with *USER*, so uk-user8 gets the value *USER*. On the other hand, the value of uk-truth-val1 is not hypothetical, so its value, FALSE, is used directly for the value of uk-truth-val8. In the trace, the new copy of user-knows1, user-knows8, is noted as being asserted into the database.

Since user-knows is a procedure (i. e. it is dominated by the PROCEDURE category), UCEgo next calls the user-knows procedure with arguments *USER*, ACTION12, and FALSE. User-knows is an entry to the KNOME component for inferring a user's knowledge state. In this case, KNOME asserts that the user does not know how to delete a file (ACTION12). Later (not shown) after UC's UNIX planner has determined that a plan for deleting a file is to use the rm command, KNOME will figure out that the user does not know rm. Finally, after more priming and matching, UC produces its answer, and tells the user to use rm (usually giving an example of using rm also).

Chapter VII

Conclusions

1. Summary

The examples presented in this thesis demonstrate that a natural language interface needs to be an intelligent agent in order to respond properly to its user. An intelligent agent based on goals and plans is the most flexible, because such a system can more easily detect positive and negative goal interactions. Within this planning paradigm, the key problem for building an intelligent agent is how to *detect* the right goals for the agent in appropriate *situations*. Once an agent has adopted appropriate goals, planning to satisfy those goals is relatively better understood problem.

Detecting situations in which an agent should detect new goals is a difficult problem, because such situations can consist of arbitrary collections of external and internal states. For an agent that serves as a natural language interface for a user, a large part of the agent's external state information will be concerned with the mental state of the user. Hence, such agents need to contain a model of the user. Furthermore, if the agent is a consultant system whose main purpose is to impart its expertise to the user, then the agent needs to model the knowledge and beliefs of the user.

In the UC system, the UCEgo component implements an intelligent agent, and the KNOME component models the knowledge and beliefs of the user. KNOME models users using a *double-stereotype* system with a range of stereotype categories representing the expertise of users and a range of difficulty levels for UNIX information. KNOME deduces particular facts about what a user knows during the dialog with the user, and uses these facts along with other clues to deduce the expertise level of a user.

The UCEgo component creates and carries out plans to satisfy goals, which it detects using the mechanism of *if-detected daemons*. These daemons are tiny inference engines that look for a class of situations and perform inferences, such as detecting goals for UC, when the class is matched by the current situation. Besides detecting goals, if-detected daemons are used to suggest plans for satisfying the newly detected goals. By indexing plans according to the type of situation in which those plans are commonly found to be useful, UCEgo can avoid considering inappropriate plans and so avoid inefficient backtracking. Situations for suggesting plans include not only the goal of the plan and the preconditions of the plan, but also *appropriateness conditions*.

After a UCEgo adopts a plan, it is further refined by the UCExpress component. Since UCEgo can only carry out plans consisting of speech acts, plan refinement is through the process of *answer expression*, which for UCExpress has two phases, *pruning*, and *formatting*. In the pruning phase, UCExpress marks as not needing generation (in some cases, this means generate a pronoun) those concepts that KNOME believes the use already knows or that are already present in the conversational context. In the formatting phase, UCExpress uses specialized *expository formats*, such as examples and similes to express information to the user in a clearer manner.

2. Current Status

UC is currently implemented in both Franz LISP and Common LISP. It runs on Digital Equipment Corporation VAX machines and on SUN 3 workstations. UC consists of approximately 10,000 lines of LISP code and 13,000 lines of KODIAK declarations in linearized form. In the original graphical form, UC's knowledge is encoded in approximately 200 KODIAK diagrams, consisting of a total of about 1000 absolutes, 2,000 relations, and 3,000 aspectuals. UCEgo contains approximately 50 if-detected daemons. A typical query takes UC from 6 to 10 seconds of real time for UC to respond on a SUN 3/140.

3. Directions for Future Research

One of the biggest problems in building a system like UC is the knowledge acquisition bottleneck. It is very difficult to determine how to best represent the concepts needed by UC, and also very time consuming to enter into UC knowledge about the various UNIX commands and their formats, effects, planfors, options, preconditions, etc. Even with more than 200 KODIAK diagrams worth of knowledge, the present version of UC only covers a subset of the UNIX file system manipulation commands. It does not cover UNIX mail, the UNIX shells, the editors available under UNIX, or the programming environment tools. Even UC's coverage of the UNIX file system is quite haphazard since many of the commands' options, preconditions, and side-effects are incompletely covered. So, to extend UC with enough knowledge that it could solve a reasonable proportion of a user's problems would require several orders of magnitude more knowledge than the present UC system. Acquiring that amount of knowledge will require a better knowledge acquisition method than hand-coding, which suffices only for experimental prototypes like UC.

The UCTeacher component ([Martin, 1985] and [Wilensky et al., 1986]) was an attempt to apply some of the techniques demonstrated in UC to the task of acquiring knowledge for UC. UCTeacher used parts of UC's natural language interface to acquire knowledge about UNIX from the user. It took advantage of UC's core knowledge about the basic form of UNIX commands to understand new commands. In a dialog with the user, UCTeacher followed a fixed script of interactions with the human expert. This approach works well for learning new concepts that are just minor variations on previously understood concepts. However, it does not work for learning completely new concepts. For example, UCTeacher could acquire knowledge about related commands, but could not be used to acquire knowledge about commands that have radically different formats or commands whose effects cannot already be represented in UC. UCTeacher did not have the flexibility to acquire significantly different concepts.

If a system like UCTeacher is to be able to acquire radically-new knowledge, it can no longer follow a fixed script of exchanges in its dialog with the human expert. Rather, it needs to be able to make decisions such as determining what information to ask the expert next. It also needs to decide when to confirm newly acquired knowledge by paraphrasing, and when to ask for clarifications such as examples, definitions, and

justifications. In more complex cases, when the information from one expert contradicts the information from another, the system needs to have a model of the experts in order to be able to decide whom to trust. These sorts of capabilities are indicative of an intelligent agent.

I believe that the methodology described in this thesis and demonstrated in the UCEgo component of UC can be applied successfully to build an intelligent agent for acquiring knowledge. Such an intelligent agent would have different requirements than UCEgo. In the case of UCEgo, UC knows more than its user and takes the initiative to correct deficiencies in the user's knowledge. On the other hand, an intelligent agent for knowledge acquisition would know less than its user, and would need to take the initiative to correct deficiencies in its own knowledge base. Also, a knowledge acquisition system is much more likely to have many different active goals at the same time, one for each different piece of information that the system will need to acquire in order to learn a new concept. The problem of choosing which goal to address first will require development of a calculus of relative goal values, a problem that was finessed in UCEgo. So, an intelligent agent for knowledge acquisition would extend the methodology described in this thesis in major new directions, as well as address the important bottleneck problem of knowledge acquisition.

Appendix A

KODIAK

The common ground for all the components of UC is the knowledge representation language *KODIAK* ([Wilensky, 1987]). A single KODIAK knowledge base serves all of the different components of UC in order to facilitate the sharing of knowledge. The use of KODIAK structures for diverse purposes in different components helps to constrain the form of the structures and makes them less ad-hoc. In keeping with this philosophy, an attempt is made in UC to store even the unshared internal knowledge of UC components in the common KODIAK knowledge base. In the case of UCEgo, KNOME, and UCExpress (the components of UC described in this thesis), all internal knowledge is represented using KODIAK. Since KODIAK knowledge structures appear throughout this thesis, this appendix presents a brief description of the KODIAK representation language and its notation. For a more detailed description and for a discussion of the motivation behind the design of the language, see [Wilensky, 1986] and [Wilensky, 1987]. A good description of KODIAK as it is used for text understanding can be found in [Norvig, 1987].

1. Basic KODIAK Concepts

KODIAK is a semantic network style representation language. There are three types of nodes: *absolutes, relations,* and *aspectuals*. An absolute is used to represent to objects (including mental objects) and events. Examples of absolutes include PERSON, USER3, and FILE. By convention, absolutes are written in upper-case. Categories of absolutes or relations in KODIAK are organized in a hierarchy. In KODIAK terminology, a category is said to *dominate* its sub-classes. Through inheritance, sub-classes inherit the properties of their dominators.

Absolutes or relations that are members of categories are termed *instances*. By convention, instances of categories are denoted by appending a positive integer to the name of the category. So, the absolute USER3 is an instance of the absolute category, USER. Through inheritance, instances inherit all the properties of their categories. An instance can be dominated by several categories, so KODIAK allows multiple inheritance.

Relations encode the relationships among different absolutes and relations. Examples of relations include HAS-PART, CONTAINS, and HAS-INTENTION4. Like absolutes, relations are written using upper-case, and instances are denoted by appending a number to the category name. Also like absolutes, relations are organized in a multiple inheritance hierarchy. Unlike absolutes, every relation has one or more aspectuals that serve as its arguments. For example, the aspectuals of HAS-PART are whole and part. By convention, aspectuals are written in lower-case, and the aspectuals of relation instances have the same numeric extension as their relation. An aspectual has a *value*, which can be an absolute, relation, or another aspectual.

For example, consider how to encode in KODIAK that the user USER1 has the name "chin," then one would use a relation, HAS-USER-NAME1, which is an instance of the relation category HAS-USER-NAME. The relation HAS-USER-NAME has the aspectuals user-name and named-user, so HAS-USER-NAME1 has the corresponding aspectuals, user-name1 and named-user1. To encode that USER1 has the name "chin," the value of named-user1 would be USER1, and the value of user-name1 would be "chin." This is shown graphically in Figure 7.1.

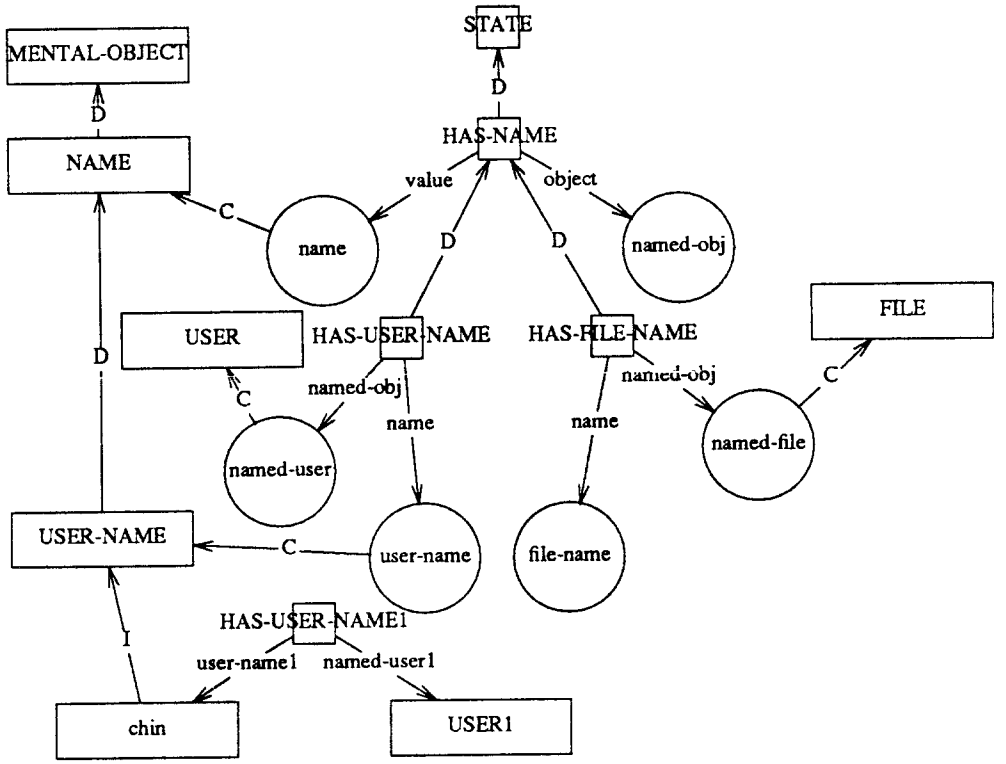

Figure 7.1. The definition of names in KODIAK.

Most of UC's KODIAK knowledge is entered into UC graphically using diagrams such as that of Figure 7.1. This particular diagram describes how names are encoded in UC. The absolutes in the diagram are depicted by wide rectangles and include: MENTAL-OBJECT, NAME, USER-NAME, FILE, USER, USER1, and chin. The relations are depicted using small squares and include: STATE, HAS-NAME, HAS-USER-NAME, HAS-FILE-NAME, and HAS-USER-NAME1. Directed lines labeled with "D" indicate that the node from which the line originates is dominated by the node to which the line points. Similarly, lines labeled with "I" indicate that the originating node is an instance of the target node.

Aspectuals are depicted either explicitly as circles or implicitly as labels on lines originating from relations. Examples of explicitly depicted aspectuals include name, named-obj, file-name, named-file, user-name, named-user. Implicit aspectuals include user-name1 and named-user1. The line labeled "named-obj" from HAS-FILE-NAME to named-file means that named-file is an aspectual of HAS-FILE-

NAME and that this aspectual plays the same role with respect to HAS-FILE-NAME as the aspectual named-obj does to HAS-NAME. This relationship between named-file and named-obj is termed a *role-play* relationship. Its use is restricted to aspectuals, and its meaning is somewhat similar to the dominate relationship between two absolutes or two relations. This role-play relationship among aspectuals allows one to name aspectuals using a convenient alias. One can refer to an aspectual, call it asp1, of a relation, rel1, by naming rel1 and also naming any other aspectual asp2, provided that asp1 plays the role of asp2 in rel1. For example, one can refer to the named-user1 aspectual of HAS-USER-NAME1 by referring to either the object, the named-obj, or the named-user of HAS-USER-NAME1.

An aspectual that is not depicted in this diagram is the truth-val aspectual, which is an aspectual that is inherited by the STATE relation. Since HAS-NAME, HAS-USER-NAME, and HAS-FILE-NAME are all dominated by STATE, they all inherit this aspectual too. The value of a truth-val aspectual can be either TRUE or FALSE. This is used in UC to encode the truth value of a relation. For example, if HAS-USER-NAME1 were to have the truth-val of FALSE, then the meaning of HAS-USER-NAME1 would be that USER1 does not have the user-name of "chin." If a relation does not have an explicit value for its truth-val aspectual, then it inherits the default truth-val of TRUE. So true relations, such as the present HAS-USER-NAME1 relation, do not need an explicit truth-val aspectual with value TRUE, because this is an inherited default value.

In the diagram, the lines labeled "C" originating from aspectuals indicate a constraint on the possible values for that aspectual or any aspectuals that it dominates. Such constraints are inherited by any other aspectuals that play the same role as the constrained aspectual in the role-play hierarchy. For example the name aspectual can only have as a value something that is a NAME. This constraint is inherited by all aspectuals that play the same role as name, such as user-name and file-name. Note that user-name is further constrained to only have values that are of a particular sub-class of NAME, namely USER-NAME.

The specialized terminology of KODIAK is summarized in the following table:

Term	Meaning
absolute	a KODIAK entity for representing objects (including mental objects) and events
aspectual	the arguments of a relation, which can have values
category	an absolute or relation that represents a category in the KODIAK multiple inheritance hierarchy
concept	any absolute, relation, or aspectual
constrain	the possible values of an aspectual can be limited or *constrained* to be filled only by instances of particular categories. These are called the *constrainers* of the aspectual
dominate	a category may dominate a sub-category in which case the sub-category is a sub-type of the parent category, which is called the *dominator* of the *dominated* sub-category; this forms a multiple inheritance hierarchy in which dominated categories inherit from their dominators
hypothetical	an ontological marker on a concept to note that it does not have a referent in the real world
instance	any absolute or relation that is a member of a category
relation	a KODIAK entity used to encode the relationship between different absolutes and relations; relations have one or more aspectuals
role-play	an aspectual of a relation often *plays* the same *role* as an aspectual in the dominator of that relation; if a *player* aspectual plays the same role as another *role* aspectual, then the player aspectual inherits the constrainers of the role aspectual; role-play links are many to many
value	aspectuals have *values* that can be other absolutes, relations, or aspectuals

Summary of KODIAK Terminology.

2. Hypotheticality

An important extension of KODIAK in UC's implementation is the idea that concepts can be hypothetical. This is represented as an ontological marker that is placed on concepts to represent the fact that these concepts do not currently have real extensions. For example, when the user asks UC, "How can I delete a file?", the representation of "a file" would be a hypothetical instance of a file, say FILE3. This hypothetical marker on FILE3 means that there is not currently an actual file in the world that UC believes to

correspond to FILE3. This can be contrasted to what happens when the user asks, "How can I delete the file named core?" In this case, "the file" is understood as a real file that has the name "core." In this case, UC believes the user, so it will also believe that there really is a file named "core" in the world. So this file is not represented as a hypothetical file by UC.

All of these ontological marks are relative to UC's world view, since the marked concepts are within UC's internal representation of the world, that is, they are "inside the head" of UC. This means that a concept that is marked as hypothetical is one that UC has not identified as currently having an extension in the real world. This does not mean that there may not exist something in the real world that could be the extension of the hypothetical concept. Nor does this imply that UC believes that the concept does not have a real extension. It merely means that UC currently does not know of a real extension for the hypothetical concept. So in the previous example of a hypothetical file, UC does not necessarily believe that there is not an actual file that the user has in mind. UC merely does not currently know of a real file to which "a file" might refer.

Hypothetical concepts should not be confused with abstract concepts. An abstract object is something that does not have a physical embodiment. Thus files, operating systems, and organizations are all abstract objects, however not all instances of files, operating systems, or organizations are hypothetical. For instance, the file named "core," UNIX, and the University of California at Berkeley are abstract entities, however they are not hypothetical.

In UC's version of KODIAK, abstract objects are those concepts that are dominated by the category MENTAL-OBJECT as opposed to those dominated by the category PHYSICAL-OBJECT. Thus, since the category FILE is dominated by MENTAL-OBJECT and not PHYSICAL-OBJECT, all files are abstract objects. Hypotheticality is implemented in a similar fashion. Hypothetical objects are encoded as instances of the HYPOTHETICAL category. However, unlike abstract, the distinction of hypotheticality extends to relations/propositions as well as to objects. All relations/propositions can be considered abstract, but not all relations/propositions are hypothetical.

The notion of hypothetical applies to relations/propositions through the classical philosophical notion that the referent of a proposition is its truth value. Thus the proposition, "I own a Ferrari Testarossa," is hypothetical, since it is not true that I now own a Ferrari Testarossa. On the other hand, the proposition, "I wish that I owned a Ferrari Testarossa," is not hypothetical, since it is true that I now wish that I owned one. In terms of KODIAK relations, "I wish that I owned a Ferrari Testarossa," would be represented as two relations, HAS-GOAL1 and HAS-OWNER1 as shown in Figure 7.2. HAS-OWNER1 represents the proposition, "I own a Ferrari Testarossa," while HAS-GOAL1 encodes "I wish that HAS-OWNER1." HAS-GOAL1 has the aspectual planner1, which has the value PERSON1 (representing "I"), and the aspectual goal1, which has the value HAS-OWNER1. The relation HAS-GOAL1 has the owner1 aspectual with the value PERSON1, and an owned-obj aspectual with the value TESTAROSSA1 (representing "Ferrari Testarossa"). In this case, HAS-GOAL1 is not hypothetical, since it is true that PERSON1 has the goal of HAS-OWNER1, but HAS-OWNER1 is hypothetical, since it is not true that PERSON1 owns TESTAROSSA1. Note also that PERSON1 is not hypothetical, since the speaker is a real person, whereas

TESTAROSSA1 is hypothetical, since there is no particular Ferrari Testarossa that is the subject of discussion.

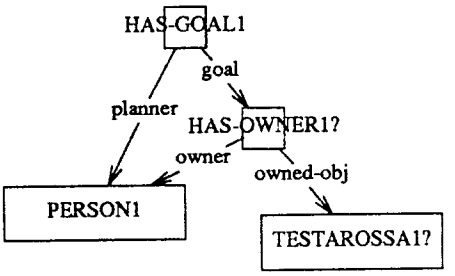

Figure 7.2. KODIAK representation of "I wish that I owned a Ferrari Testarossa."

Although a hypothetical proposition is not necessarily true, it is not necessarily false either. For example, if the user were to state, "I suspect the machine is down," then "the machine is down" is hypothetical, since UC does not have enough information to form a belief about whether the machine is really down or not. Similarly in the previous example about wanting to own a Testarossa, the fact that HAS-OWNER1 is hypothetical does not imply that PERSON1 does not own a Testarossa. That may be inferable from the fact that the person wants to own one. However it is a not necessary consequence, since it is conceivable that PERSON1 has amnesia and does in fact own a Testarossa.

The hypothetical marker serves two main purposes in UC. The first purpose is so that UC can tell if propositions in its knowledge base are true. So in the example of the user asking, "How can I delete the file named core?" UC's goal analyzer will deduce that the user has the goal of the user knowing how to delete the file named core. This is represented in UC's knowledge base as the HAS-GOAL3 relation with aspectual goal3 having the value, KNOW3, which is hypothetical. Later, if UC were to look in its knowledge base to see whether the user knows how to delete the file named core, it will find KNOW3 and realize through the hypothetical marker that this does not mean that the user does know. Without the hypothetical marker, UC would not be able to tell that KNOW3 is not a real fact even by inspecting all the relations in which KNOW3 participates and finding that it is part of the HAS-GOAL3 relation. Even this does not work, since participating as the goal in a HAS-GOAL relation does not necessarily mean that the participant is not true. It is plausible that the planner might want something that already holds. For example, a planner may want the sky to be clear even when the sky is clear.

The second purpose of the hypothetical marker is to replace variables. In UC, hypothetical concepts are used in place of the variables found in ordinary programs. For example, if the rule, *to delete ?x where ?x is a file, type "rm" followed by ?x,* were represented in UC, then the variable ?x would be represented in UC as a hypothetical instance of a file. Then a rule applier could match the hypothetical file to any other instance of a file. Such replacement of variables is possible, since the great majority of "variables" in UC are type-constrained as in the previous example where ?x is constrained to be a file. In those infrequent cases where there are no such constraints on the

variable, UC can use a hypothetical instance of the category SOMETHING, which is unconstrained, since all other categories are dominated by SOMETHING.

Replacing variables with hypothetical markers on concepts emphasizes the fact that variables in UC are first of all concepts and only secondly variables. Hypothetical concepts are considered variables in the sense that UC does not know of a referent for the hypothetical concept in the real world. When a hypothetical concept is found to match to a real concept, then the referent of the hypothetical concept has been found: its referent is the real world referent of the matching real concept. In the case where a hypothetical concept is found to match another hypothetical concept, then this is equivalent to unification of variables.

In keeping with the convention of prepending a "?" to variables, hypothetical concepts are marked in UC by appending a "?" to the name of the concept. This convention is built into UC's printers for output so that hypothetical concepts can easily be identified. The convention is optional for input of KODIAK concepts to UC.

The hypothetical marker is only a partial solution to the problem of representing the ontological status of concepts. It does not even address some of the issues dealt with by possible world semantics. However, it has proven sufficient in UC for distinguishing between those concepts that the system believes and other concepts in UC's knowledge base. More importantly, the marker is an improvement over traditional variables. It provides a simple means of encoding type constraints on variables that fits extremely well into the framework of hierarchical conceptual network systems.

Appendix B

UNIX Commands

The following table lists the UNIX commands that are mentioned in this thesis and their main uses.

Command	Usage
cat	concatenate and print files
chmod	change the protection of files
compact	compress files
cp	copy files
diff	compare files
du	list disk usage statistics
emacs	extensible editor that runs on many systems
finger	list information about users
grep	search a file for a pattern
kill	terminate processes
lpr	print a file
ls	list file names
ls -i	list files and their inodes
ls -l	list files and their protections
mkdir	create directories
more	text-file contents perusal program
mv	move or rename files
ps	list the status of processes
rcp	copy files over the network
rm	remove files
rmdir	remove directories
rn	read news
rogue	fantasy game
ruptime	list the uptime of all machines on the network
rwho	list the current users for all machines on the network
spell	list spelling errors in a file
spice	electronic circuit simulation program
tset	terminal dependent initialization
uptime	list the uptime of the machine
vi	visual display editor
who	list the current users of the machine

Appendix C

References

Allen, J. F. (1979). *A Plan-Based Approach to Speech Act Recognition.* Doctoral dissertation, Department of Computer Science, Uninversity of Toronto. Also available as Technical Report No. 131, University of Toronto.

Allen, J. F. and Perrault, C. R. (1980). Analyzing Intention in Utterances. In *Artificial Intelligence,* 15, pp. 143-178.

Alterman, R. (1986). An Adaptive Planner. In *Proceedings of the National Conference on Artificial Intelligence,* pp. 65-69. Philadelphia, PA, August.

Anderson, J. R., Boyle, F., G. Yost. (1985). The Geometry Tutor. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence,* 1, pp. 1-7. Los Angeles, CA, August.

Appelt, D. E. (1981). *Planning Natural Language Utterances to Satisfy Multiple Goals.* Doctoral dissertation, Computer Science Department, Stanford University. Also available as SRI International AI Center Technical Note 259.

Appelt, D. E. (1983). TELEGRAM: A Grammar Formalism for Language Planning. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence,* 1, pp. 595-599, Karlsruhe, West Germany, August.

Arens, Y. (1986). *CLUSTER: An Approach to Contextual Language Understanding.* Doctoral dissertation, University of California, Berkeley. Also available as Computer Science Division, University of California, Berkeley, Report No. UCB/SCD 86/293.

Austin, J. L. (1962). *How to do things with words.* London: Oxford University Press.

Barr, A. (1977). *Meta-knowledge and Memory.* Stanford University Heuristic Programming Project working paper HPP-77-37.

Bobrow, D. G., Kaplan, R. M., Kay, M., Norman, D. A., Thompson, H., and Winograd, T. (1977). GUS-1, a Frame-Driven Dialog System. In *Artificial Intelligence* 8 (2), pp. 155-173.

Breuker, J., Winkels, R. Sandberg, J. (1987). A Shell for Intelligent Help Systems. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence.* 1, pp. 167-173. Milano, Italy, August.

Breuker, J. (1987). Coaching in Help Systems. To appear in Self, J. (Ed.), *Intelligent Computer-Aided Instruction. London: Chapman & Hall.*

Brown, J. S., Burton, R. R. (1978). Diagnostic Models for Procedural Bugs in Basic Mathematical Skills. In *Cognitive Science,* 2, pp. 155-192.

Burton, R. R., Brown, J. S. (1976). A Tutoring and Student Modelling Paradigm for Gaming Environments. In *Symposium on Computer Science and Education,* pp. 236-246. Anaheim, CA, February.

Carberry, S. (1983). Tracking User Goals in an Information-Seeking Environment. In *Proceedings of the National Conference on Artificial Intelligence,* pp. 59-63. Washington, DC, August.

Carberry, S. (1987). Plan Recognition and Its Use in Understanding Dialogue. To appear in Kobsa, A. and Wahlster, W. (Eds.), *User Models in Dialog Systems.* Berlin: Springer.

Carbonell, J. G. (1982). Where Do Goals Come From? In *Proceedings of the Fourth Annual Conference of the Cognitive Science Society,* pp. 191-194. Ann Arbor, MI, August.

Carbonell, J. G. (1986). Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition. In Michalski, R. S., Carbonell, J. G., and Mitchell, T. M. (Eds.), *Machine Learning,* Vol. II. Los Altos, CA: Morgan Kaufmann.

Carbonell, J. R. (1970a). *Mixed-Initiative Man-Computer Instructional Dialogues.* Doctoral dissertation, Massachusetts Institute of Technology. Also available as Bolt Beranek and Newman Report No. 1971.

Carbonell, J. R. (1970b). AI in CAI: An Artificial-Intelligence Approach to Computer Assisted Instruction. In *IEEE Transactions on Man-Machine-Systems, MMS-11 (4), December, pp. 190-202.*

Carr, B., Goldstein, I. (1977). *Overlays: A Theory of Modeling for Computer Aided Instruction.* Technical Report AI Memo 406, AI Laboratory, Massachusetts Institute of Technology.

Charniak, E. (1972). *Towards a Model of Children's Story Comprehension.* Massachusetts Institute of Technology AI Technical Report TR-266.

Chin, D. N. (1983a). A Case Study of Knowledge Representation in UC. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence.* 1, pp. 388-390. Karlsruhe, West Germany, August.

Chin, D. N. (1983b). Knowledge Structures in UC, the UNIX Consultant. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics,* pp. 159-163. Boston, MA, June.

Chin, D. N. (1984). An Analysis of Scripts Generated in Writing Between Users and Computer Consultants. In *AFIPS Proceedings of the 1984 National Computer Conference.* pp. 637-642. Las Vegas, NV, July.

Chin, D. N. (1986). User modeling in UC, the UNIX consultant. In *Proceedings of the CHI-86 Conference,* pp. 24-28. Boston, MA, April 1986.

Chin, D. N. (1987). KNOME: Modeling What the User Knows in UC. To appear in Kobsa, A. and Wahlster, W. (Eds.), *User Models in Dialog Systems.* Berlin: Springer.

Cohen, P. R. (1978). *On Knowing What to Say: Planning Speech Acts.* Doctoral dissertation, Department of Computer Science, Uninversity of Toronto. Also available as Technical Report No. 118, University of Toronto.

Cohen, P. R., and Levesque, H. J. (1987a). *Persistence, Intention, and Commitment.* SRI International Technical Report 415.

Cohen, P. R., and Levesque, H. J. (1987b). *Rational Interaction as the Basis for Communication.* Stanford University Center for the Study of Language and Information Report 89.

Cohen, P. R., Perrault, C. R. (1979). Elements of a plan-based theory of speech acts. In *Cognitive Science,* 3, pp. 177-212.

Cox, C. A. (1986). *ALANA — Augmentable LANguage Analyzer.* Computer Science Division, University of California, Berkeley, Report No. UCB/SCD 86/283.

Davis, R. Buchanan, B. G. (1977). Meta-level knowledge: Overview and Applications. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence,* pp. 920-927. Cambridge, MA.

Deering, M., Faletti, J., and Wilensky, R. (1982). *Using the PEARL AI Package.* Computer Science Division, University of California, Berkeley, Memorandum No. UCB/ERL M82-19.

Douglass, R,, and Hegner S. (1982). An Expert Consultant for the Unix System: Bridging the Gap Between the User and Command Language Semantics. In *Proceedings of the Fourth National Conference of the Canadian Society for Computational Studies of Intelligence*. pp. 92-96. Saskatoon, Canada. May.

Duda, R. O., Hart, P. E., Nilsson, N. J., Sutherland, G. L. (1978). Semantic Network Representations in Rule-Based Inference Systems. In Waterman, D. A. and Hayes-Roth, F. (Eds.), *Pattern-Directed Inference Systems*. pp. 155-176. New York: Academic Press.

Faletti, J. (1982). PANDORA — A Program for Doing Commonsense Planning in Complex Situations. In *Proceedings of the Second Annual National Conference on Artificial Intelligence,* pp. 185-188. Pittsburg, PA, August.

Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. In *Artificial Intelligence,* 2 (3-4), pp. 189-208.

Finin, T. W. (1983). Providing Help and Advice in Task Oriented Systems. In *Proceedings of the Eight International Joint Conference on Artificial Intelligence,* 1, pp. 176-178. Karlsruhe, West Germany, August.

Finin, T. W. (1987). GUMS — A General User Modeling Shell. To appear in Kobsa, A. and Wahlster, W. (Eds.), *User Models in Dialog Systems*. Berlin: Springer.

Fischer, G., Lemke, A., Schwab, T. (1985). Knowledge-based Help Systems. In *Proceedings of the CHI'85 Conference,* pp. 161-167. San Francisco, April.

Forgy, C. L. (1982). Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. In *Artificial Intelligence,* 19, pp. 17-37.

Friedland, P. E. (1980). *Knowledge-Based Experiment Design in Molecular Genetics*. Doctoral dissertation, Computer Science Department, Stanford University.

Friedland, P. E., Iwasaki, Y. (1985). The COncept and Implementation of Skeletal Plans. In *Journal of Automated Reasoning,* I, pp. 161-208.

Grice, H. P. (1975). Logic and Conversation. In Cole, P. and Morgan, J. L. (Eds.), *Studies in Syntax,* Vol. III, pp. 41-58. New York: Seminar Press.

Hammond, K. J. (1986). CHEF: A Model of Case-based Planning. In *Proceedings of the National Conference on Artificial Intelligence,* pp. 65-69. Philadelphia, PA, August.

Hegner, S. (1988). Representation of Command Language Behavior for an Operating System Consultation Facility. To appear in *Proceedings of the Fourth IEEE Conference on Artificial Intelligence Applications*. San Diego, CA, March.

Hendler, J. A. (1985). *Integrating Marker-Passing and Problem-Solving*. Doctoral dissertation, Brown University. Also available as Technical Report CS-85-08, Computer Science Department, Brown University.

Hobbs, J. R. and Evans, D. A. (1980). Conversation as Planned Behavior. In *Cognitive Science*, 4 (4), October.

Hoeppner, W., Morik K., and Marburger H. (1984). *Talking it Over: The Natural Dialog System HAM-ANS*. Research Unit for Information Science and Artificial Intelligence, University of Hamburg, Report ANS-26.

Hovy, E. H. (1987). *Generating Natural Language Under Pragmatic Constraints*. Doctoral dissertation, Yale University. Also available as Yale University Computer Science Research Report YALEU/CSD/RR #521.

Jacobs, P. S. (1983). Generation in a Natural Language Interface. In *Proceedings of the Eight International Joint Conference on Artificial Intelligence*, 1, pp. 610-612. Karlsruhe, West Germany, August.

Jacobs, P. S. (1986). *A Knowledge-Based Approach to Language Production*. Doctoral dissertation, University of California, Berkeley. Also available as Computer Science Division, University of California, Berkeley, Report No. UCB/SCD 86/254.

Jerrams-Smith, J. (1986). *The Application of Expert Systems to the Design of Human-Computer Interfaces*. Doctoral dissertation, Birmingham University, United Kingdom.

Joshi, A., Webber, B., and Weischedel, R. M. (1984). Preventing False Inferences. In *Proceedings of Coling84*, pp. 134-138. Stanford, CA, July.

Johnson, W. L. and Soloway, E. (1984). Intention-Based Diagnosis of Programming Errors. In *Proceedings of the National Conference on Artificial Intelligence*, pp. 162-168. Austin, TX, August.

Joshi, A., Webber B. L., Weischedel, R. M. (1984). Preventing False Inferences. In *Proceedings of Coling84*, pp. 134-138. Stanford, CA, July.

Joshi, A. and Webber, B. (1984). Living up to Expectations: Computing Expert Responses. In *Proceedings of the National Conference on Artificial Intelligence,*

pp. 169-175. Austin, TX, August.

Kaplan, S. J. (1983). Cooperative Responses from a Portable Natural Language Database Query System. In M. Brady and R. C. Berwick (Eds.), *Computational Models of Discourse*. Cambridge, MA: MIT Press.

Kass, R. (1987). Student Modelling in Intelligent Tutoring Systems — Implications for User Modelling. To appear in Kobsa, A. and Wahlster, W. (Eds.), *User Models in Dialog Systems*. Berlin: Springer.

Kay, D. S. and Black, J. B. (1985). The Evolution of Knowledge Representations with Increasing Expertise in Using Systems. In *Proceedings of the Seventh Annual Conference of the Cognitive Science Society*, pp. 140-149. Irvine, CA, August.

Kemke, C. (1987). The SINIX Consultant: Requirements, Design, and Implementation of an Intelligent Help System for a UNIX Derivative. To appear in Benold, T. (Ed.), *Intelligent User Interfaces*. Amsterdam: North Holland.

Kobsa, A. (1985a). VIE-DPM: A User Model in a Natural-Language Dialogue System. In Laubsch, J. (Ed.), *GWAI-84, 8th German Workshop on Artificial Intelligence*, Berlin: Springer.

Kobsa, A. (1985b). *User Modelling in Dialogue Systems: Potentials and Hazards*. Report 85-o1, University of Vienna, Department of Medical Cybernetics, Vienna, Austria.

Kobsa, A. (1986). Generating a User Model from Wh-questions in the VIE-LANG System. In P. Hellwig and H. Lehman (Eds.), *Trends in der Linguistischen Datenverarbeitung*. Hildesheim, West Germany: Olms Verlag.

Kolodner, J. L., Simpson, R. L., and Sycara-Cyranski, K. (1985). A Process Model of Case-Based Reasoning in Problem Solving. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pp. 284-290. Los Angelos, CA, August.

Lehman, J. F., Carbonell, J. G. (1987). Learning the User's Language: A Step Towards Automated Creation of User Models. To appear in Kobsa, A. and Wahlster, W. (Eds.), *User Models in Dialog Systems*. Berlin: Springer.

Lehnert, W. G. (1978). *The Process of Question Answering*. Hillsdale, NJ: Lawrence Erlbaum.

Litman, D. J. and Allen, J. F. (1984). A plan recognition model for clarification subdialogues. In *Proceedings of the Tenth International Conference on Computational Linguistics,* 302-311. Palo Alto, CA, July.

Luria, M. (1982). Dividing up the Question Answering Process. In *Proceedings of the National Conference on Artificial Intelligence,* pp. 71-74. Pittsburgh, PA, August.

Luria, M. (1985). Commonsense Planning in a Consultant System. In *Proceedings , 1985 IEEE International Conference on Systems, Man, and Cybernetics,* pp. 602-606. Tuscon, AR, November.

Luria, M. (1987). Expressing Concern. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics,* pp. 221-227. Stanford, CA, July.

Luria, M. (forthcoming). Doctoral dissertation, University of California, Berkeley.

Macmillan, S. A. (1978). *User Models to Personalize an Intelligent Agent.* Doctoral dissertation, School of Education, Stanford University.

Marburger, H. (1986). A Strategy for Producing Cooperative nl Reactions in a Database Interface. In *Proceedings of AIMSA-86,* Wana, Bulgaria

Martin, J. H. (1985). Knowledge Acquisition through Natural Language Dialogue. In *Proceedings of the 2nd Conference on Artificial Intelligence Applications,* Miami, FL.

Martin, J. H. (1987). Understanding New Metaphors. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence,* pp. 137-139. Milano, Italy, August.

Matthews, M., Biswas, G. (1986). *USCSH: An Active Assistance Interface for Unix.* University of South Carolina technical report USC-CS TR 86-003, June.

Mayfield, J. (forthcoming). Doctoral dissertation, University of California, Berkeley.

Mays, E. (1980). Failures in Natural Language Systems: Applications to Data Base Query Systems. *Proceedings of the National Conference on Artificial Intelligence,* pp. 327-330. Stanford, CA, August.

McCoy, K. F. (1985). *Correcting Object-Related Misconceptions.* Doctoral dissertation, University of Pennsylvania, Department of Computer and Information Science, Moore School. Also available as report MS-CIS-85-57.

McCoy, K. F. (1987). Highlighting a User Model to Respond to Misconceptions. To appear in Kobsa, A. and Wahlster, W. (Eds.), *User Modelling in Dialog Systems.* Berlin: Springer.

McDermott, J., Newell, A., Moore, J. (1978). The Efficiency of Certain Production System Implementations. In Waterman, D. A. and Hayes-Roth, F. (Eds.), *Pattern-Directed Inference Systems.* pp. 155-176. New York: Academic Press.

McKevitt, P., Wilkes, Y. (1987). Transfer Semantics in an Operating System Consultant: the Formalization of Actions Involving Object Transfer In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence,* pp. 569-575. Milano, Italy, August.

McKeown, K. R. (1985). Discourse Strategies for Generating Natural-Language Text. In *Artificial Intelligence,* 27, pp. 1-41.

Meehan, J. R. (1976). *The Metanovel: Writing Stories by Computer.* Doctoral dissertation, Yale University. Also available as Yale University Computer Science Research Report #74 and through New York: Garland Publishing, 1980.

Meehan, J. R. (1981). TALE-SPIN. In Schank, R. C. and Riesbeck, C. K. (Eds.), *Inside Computer Understanding,* pp. 197-226. Hillsdale, NJ: Lawrence Erlbaum.

Morik, K. and Rollinger, C-R. (1985). The Real Estate Agent — Modeling the User by Uncertain Reasoning. In *AI Magazine,* 6 (2), pp. 44-52. Summer 1985.

Morik, K. (1987). User Models and Conversational Settings: Modeling the User's Wants. To appear in Kobsa, A. and Wahlster, W. (Eds.), *User Models in Dialog Systems.* Berlin: Springer.

Nessen, E. (1986). *SC-UM — User Modeling in the SINIX Consultant.* Memo, FR.10.2 Informatik IV, University of the Saarland, Saarbrucken, West Germany.

Newell, A. and Simon, H. A. (1963). GPS, a Program that Simulates Human Thought. In Feigenbaum, E. A. and Feldman, J. (Eds.), *Computers and Thought.* New York: McGraw Hill.

Newell, A. and Simon, H. A. (1972). *Human Problem Solving.* Englewood Clifts, NJ: Prentice-Hall.

Norvig, P. (1983). Frame Activated Inferences in a Story Understanding Program. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence,* pp. 624-626. Karlsruhe, West Germany.

Norvig, P. (1987). *A Unified Theory of Inference for Text Understanding*. Doctoral dissertation, University of California, Berkeley. Also available as Computer Science Division, University of California, Berkeley, Report No. UCB/SCD 87/339.

Paris, C. L. (1987). Tailoring Object Descriptions to a User's Level of Expertise. To appear in Kobsa, A. and Wahlster, W. (Eds.), *User Models in Dialog Systems*. Berlin: Springer.

Quilici, A., Dyer, M., Flowers, M. (1985). *Understanding and Advice Giving in AQUA*. Technical Report UCLA-AI-R-85-19, Artificial Intelligence Laboratory, UCLA.

Quilici, A., Dyer, M., Flowers, M. (1986). AQUA: An Intelligent UNIX Advisor. In *Proceedings of the 1986 European Conference on Artificial Intelligence*. pp. 33-39. Brighton, England.

Quilici, A., Dyer, M., Flowers, M. (1987). Recognizing and Responding to Plan-Oriented Misconceptions. To appear in Kobsa, A. and Wahlster, W. (Eds.), *User Models in Dialog Systems*. Berlin: Springer.

Rau, L. F. (1985). *The Understanding and Generation of Ellipses in a Natural Language System*. Computer Science Division, University of California, Berkeley, Report No. UCB/CSD 85/227.

Rau, L. F. (1987a). Information Retrieval from Never-ending Stories. In *Proceedings of the National Conference on Artificial Intelligence*, pp. 317-321. Seattle, WA, July.

Rau, L. F. (1987b). Spontaneous Retrieval in a Conceptual Information System. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pp. 155-162. Milano, Italy, August.

Reiser, B., Anderson, J. R., and Farrell, R. G. (1985). Dynamic Student Modelling in an Intelligent Tutor for Lisp Programming. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, 1, pp. 8-14. Los Angeles, CA, August.

Reiter, R. (1978). On Closed World Data Bases. In Gallaire, H. and Minker, J. (Eds.), *Logic and Data Bases*, New York: Plenum Press.

Rich, E. (1979). User Modeling via Stereotypes. In *Cognitive Science*, 3, pp. 329-354.

Rich, E. (1983). Users are individuals: individualizing user models. In the *Int. Journal of Man-Machine Studies*, 18, pp. 199-214.

Rich, E. (1983). Users are Individuals: Individualizing User Models. *Int. Journal of Man-Machine Studies,* 18, pp. 199-214.

Rich, E. (1987). Stereotypes and User Modelling. To appear in Kobsa, A. and Wahlster, W. (Eds.), *User Models in Dialog Systems.* Berlin: Springer.

Rissland, E. L. (1983). Examples in Legal Reasoning: Legal Hypotheticals. In *Proceedings of the Eight International Joint Conference on Artificial Intelligence,* 1, pp. 90-93. Karlsruhe, West Germany, August.

Rissland, E. L., Valcarce, E. M., and Ashley, K. D. (1984). Explaining and Arguing with Examples. In *Proceedings of the National Conference on Artificial Intelligence,* pp. 288-294. Austin, TX, August.

Rosch, E. (1983). Prototype Classification and Logical Classification: The Two Systems. In E. Scholnick (Ed.), *New Trends in Cognitive Representation: Challenges to Piaget's Theory.* Hillsdale, NJ: Lawrence Erlbaum.

Rosch, E. (1978). Principles of Categorization. In Rosch, E. and Lloyd, B. B. (Eds.), *Cognition and Categorization.* Hillsdale, NJ: Lawrence Erlbaum.

Rosch, E. (1977). Human Categorization. In Warren, N. (Ed.), *Studies in Cross Cultural Psychology,* I. London: Academic Press.

Rosenbloom, P. S., and Newell, A. (1982). Learning by Chunking: Summary of a Task and a Model. In *Proceedings of the National Conference on Artificial Intelligence,* pp. 255-257. Pittsburgh, PA, August.

Sacerdoti, E. D. (1974). *Planning in a Hierarchy of Abstraction Spaces. In Artificial Intelligence,* 5 (2), pp. 115-135.

Sacerdoti, E. D. (1977). *A Structure for Plans and Behavior.* Amsterdam: Elsevier North-Holland.

Searle, J. R. (1969). *Speech Acts; An Essay in the Philosophy of Language.* Cambridge, England: Cambridge University Press.

Searle, J. R. (1975). Indirect Speech Acts. In Cole, P. and Morgan, J. L. (Eds.), *Syntax and Semantics, Vol. III: Speech Acts* New York: Academic Press.

Schank, R. C. (1975). *Conceptual Information Processing.* Amsterdam: North-Holland.

Schank, R. C. and Abelson, R. P. (1977). *Scripts, Plans, Goals, and Understanding.* Hillsdale, NJ: Lawrence Erlbaum.

Shrager, J. C., Finin, T. (1982). An Expert System that Volunteers Advice. In *Proceedings of the Second Annual National Conference on Artificial Intelligence,* pp. 339-340. Pittsburg, PA, August.

Shrager, J. C. (1981). *Invoking a Beginner's Aid Processor by Recognizing JCL Goals.* Master's thesis, The Moore School, University of Pennsylvania. Also available as Report MS-CIS-81-7, Computer and Information Science.

Sleeman, D., Smith, M. J. (1981). Modelling Student's Problem Solving. *Artificial Intelligence,* 16, pp. 171-187.

Sleeman, D. and Brown, J. S. (1982). Editors of *Intelligent Tutoring Systems.* New York: Academic Press.

Smith, D. E., and Genesereth, M. R. (1983). *Finding All of the Solutions to a Problem: Serious Applications of Meta-Level Reasoning II.* Stanford University Heuristic Programming Project Memo HPP-83-21.

Stefik, M. (1980). *Planning with Constraints.* Doctoral dissertation, Computer Science Department, Stanford University. Also available as Report No. 80-784.

Stefik, M. (1981). Planning and Meta-Planning (MOLGEN: Part 2). In *Artificial Intelligence,* 16, pp. 141-170.

Stevens, A., Collins, A. Goldin, S. E. (1979). Misconceptions in Student's Understanding. In *Intl. Journal of Man-Machine Studies,* 11, pp. 145-156.

Sussman, G. J. (1975). *A Computer Model of Skill Acquisition.* New York: American Elsevier.

Tate, A. (1975). Interacting Goals and their Use. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence,* pp. 215-218. Tbilisi, Georgia, USSR, September.

Wahlster W., Marburger, H., Jameson, A., and Busemann, S. (1983). Over-Answering Yes-No-Questions: Extended Responses in a NL Interface to a Vision System. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence,* pp. 643-646. Karlsruhe, West Germany, August.

Wahlster, W. and Kobsa, A. (1987). Dialog-Based User Models. In Kobsa, A. and Wahlster, W. (Eds.), *User Models in Dialog Systems*. Berlin: Springer.

Waldinger, R. (1977). Achieving Several Goals Simultaneously. In Elcock, E. W. and Michie, D. (Eds.), *Machine Intelligence 8*, New York: Halstead/Wiley.

Warren, D. H. D. (1974). *WARPLAN: A System for Generating Plans*. Department of Computational Logic, University of Edinburgh, School of Artificial Intelligence, Memo 76.

Webber, B. L. and Mays, E. (1983). Varieties of User Misconceptions: Detection and Correction. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, 2, pp. 650-652. Karlsruhe, West Germany, August.

Wilensky, R. (1978). *Understanding Goal-Based Stories*, Doctoral dissertation, Yale University. Also available as Yale University Computer Science Research Report #140.

Wilensky, R. (1983). *Planning and Understanding: A Computational Approach to Human Reasoning*. Reading, MA: Addison-Wesley.

Wilensky, R., Arens, Y., and Chin, D. N. (1984). Talking to UNIX in English: An Overview of UC. In *Communications of the ACM*, 27 (6), June.

Wilensky, R., Mayfield, J., Albert, A., Chin, D. N., Cox, C., Luria, M., Martin, J., and Wu, D. (1986). *UC — A Progress Report*. Computer Science Division, University of California, Berkeley, Report No. UCB/CSD 87/303.

Wilensky, R. (1987). *Some Problems and Proposals for Knowledge Representation*. Computer Science Division, University of California, Berkeley, Report No. UCB/CSD 87/351.

Zadeh, L. A. (1965). Fuzzy Sets. *Information and Control* 8, pp. 338-353.

Zadeh, L. A. (1982). *A Computational Approach to Fuzzy Quantifiers in Natural Language*. Computer Science Division, University of California, Berkeley, Memorandum No. UCB/ERL M82-36(Revised).