Artificial Intelligence Review **14:** 43–88, 2000.
Stephen J. Hegner et al. (eds.), Intelligent help systems for UNIX.
© 2000 *Kluwer Academic Publishers. Printed in the Netherlands.*

The Berkeley UNIX Consultant Project *

ROBERT WILENSKY[1], DAVID N. CHIN[2], MARC LURIA[3], JAMES MARTIN[4], JAMES MAYFIELD[5] and DEKAI WU[6]
[1]*Division of Computer Science, Department of EECS, University of California, Berkeley, Berkeley, CA 94720, U.S.A. (E-mail: wilensky@cs.berkeley.edu);* [2]*Department of Information and Computer Sciences, University of Hawaii, 1680 East West Rd., Honolulu, HI 96822, U.S.A. (E-mail: Chin@Hawaii.Edu);* [3]*NDS Technologies Israel Ltd., Jerusalem, Israel (E-mail: mluria@ndc.co.il);* [4]*Department of Computer Science, University of Colorado, Boulder, CO 80309-0430, U.S.A. (E-mail: martin@cs.colorado.edu);* [5]*Computer Science and Electrical Engineering Dept., University of Maryland Baltimore County, Baltimore, MD 21228-5398, U.S.A. (E-mail: mayfield@cs.umbc.edu);* [6]*Department of Computer Science, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong, P.R. China (E-mail: dekai@cs.ust.hk)*

Abstract. UC (UNIX Consultant) is an intelligent, natural-language interface that allows naive users to learn about the UNIX operating system. UC was undertaken because the task was thought to be both a fertile domain for Artificial Intelligence research and a useful application of AI work in planning, reasoning, natural language processing, and knowledge representation. The current implementation of UC comprises the following components: A language analyzer, called ALANA, that produces a representation of the content contained in an utterance; an inference component called a *concretion mechanism* that further refines this content; a goal analyzer, PAGAN, that hypothesizes the plans and goals under which the user is operating; an agent, called UCEgo, that decides on UC's goals and proposes plans for them; a domain planner, called KIP, that computes a plan to address the user's request; an expression mechanism, UCExpress, that determines the content to be communicated to the user, and a language production mechanism, UCGen, that expresses UC's response in English. UC also contains a component called KNOME that builds a model of the user's knowledge state with respect to UNIX. Another mechanism, UCTeacher, allows a user to add knowledge of both English vocabulary and facts about UNIX to UC's knowledge base. This is done by interacting with the user in natural language. All these aspects of UC make use of knowledge represented in a knowledge representation system called KODIAK. KODIAK is a relation-oriented system that is intended to have wide representational range and a clear semantics, while maintaining a cognitive appeal. All of UC's knowledge, ranging from its most general concepts to the content of a particular utterance, is represented in KODIAK.

Keywords: agent, UNIX, consultant, natural language, intelligent interface, planning, knowledge representation, user modeling

---

[*] This paper is a considerably abridged version of "The Berkeley UNIX Consultant Project," in *Computational Linguistics* **14**(4): 35–84, Copyright ©1988 ACL and MIT Press.

1. Introduction to the UNIX Consultant (UC) Project

Some time ago, we began a project called UC (UNIX Consultant). UC was to function as an intelligent, natural-language interface that would allow naive users to learn about the UNIX[1] operating system by interacting with the consultant in ordinary English. We sometimes refer to UC as “an intelligent ‘help’ facility” to emphasize our intention to construct a consultation system, rather than a natural-language front-end to an operating system. Whereas front-ends generally take the place of other interfaces, UC was intended to help the user learn how to use an existing one.

We had two major motivations for choosing this task. These can be summarized by saying that we believed the task to be both interesting and doable. It seemed to us that much natural-language work – indeed, much of AI research – has fallen into two largely non-intersecting categories: On the one hand, there are quite interesting and ambitious projects that have been more the fertile source of exciting speculations than of useful technology. In contrast, there are projects whose scope is severely limited, either to some intrinsically bounded, real-world task or to a laboratory micro-world. These projects result in much excitement by the production of a “working system” or successful technology. But such projects have rarely produced much in the way of progress on fundamental issues that comprise the central goals of AI researchers.

Our hope was that the consultation task would require us to address fundamental problems in natural-language processing, planning and problem solving, and knowledge representation, all of which are of interest to us. We believe this to be the case because (1) the domain of an operating system is quite large and complex, (2) users’ conceptions of computer systems are often based on other domains, particularly space and containment, and (3) the structure of a consultation session requires the consultant to understand the user’s language, hypothesize the user’s intentions, reason about the user’s problem, access knowledge about the topic in question, and formulate a reasonable response. In sum, virtually all the problems of language processing and reasoning arise in some fashion.

While the task is interesting, it is nevertheless limited. Arbitrary knowledge of the world is generally not required, as it may be in other natural-language tasks, such as text processing. Even knowledge about the domain might be limited in ways that do not compromise the overall integrity of the system. In particular, the task is intrinsically “fail-soft.” Since the system is a ‘help’ facility, it need not be capable of handling every task put to it to serve a useful function. This is probably less true of systems that are intended to be interfaces. In their case, failure to correctly process a request by the user leaves the user with little recourse. However, a consultant may be quite useful even if it cannot help all the time.

Similarly, there are strategies that might be employed in a consultant task that further reduce the degree of coverage required by the system. For example, if asked a very specific question, it is not unreasonable that a consultant respond by telling the user where to look for the information. Thus, the degree of expertise of the consultation system may be circumscribed.

In other words, we felt that the operating-system domain was an appropriate replacement for the "blocks world." Building a consultant for the domain is a real task one would like to have accomplished. The domain would limit the breadth, but not the depth, of AI research required.

1.1. UC – *science or engineering?*

Our approach to AI has had a distinctly "cognitive" bent. While a lengthy exposition might be needed to define this precisely, let it suffice here to say that we are interested in modeling human beings at least to a first approximation. Thus, as far as we could, we have attempted to build a system that modeled how we believe a human consultant actually functions.

In some cases, this goal meant that we would make some problems harder for ourselves than one might if one's goals were strictly technological. For example, since many word senses are unlikely to be used when talking to a consultant, a purely engineering approach might play down the problem of ambiguity. However, it is our goal to address such problems in a general fashion.

At the same time, there were many pragmatic concessions that were made in implementing UC. Some of these were forced on us by the nature of university research. For example, a process might be divided into two components for the sake of implementation, although the particular division may not be motivated otherwise. These components might even exercise two different approaches to similar subproblems, depending on the biases of their authors. Sometimes, for the sake of efficiency, we chose to implement only part of what we believed to be a larger process. Also for efficiency's sake, and to prevent truly difficult but infrequent problems from scuttling the entire effort, we implemented some solutions that we did not believe in completely. For example, UC's control structure is overly simplistic in ways that we understand but have not corrected. We will make note of other such situations in the text below. In general, when this was the case, the solution used took the form of checking for certain frequently occurring cases in order to preclude having to solve a general problem.

Since our goals were not strictly technological, we did not feel that it was necessary or appropriate in order for our system to be considered a success to produce a product that could actually be used in a real-world setting. However, we did feel that we should show that one could develop such a system along the lines that our research suggested. This would be accomplished by developing an extendible prototype.

1.2. *Reasonable agents versus intelligent interfaces*

Our goal in building UC is to simulate a human consultant. As a result, the system has a structure that is more complex than other so-called intelligent interfaces. Indeed, we feel that looking at such a system as an *interface* is misleading. Instead, we prefer the metaphor of a *reasonable agent.* Unlike an interface, which is a conduit through which information flows, an agent is a participant in a situation. In particular, an agent has explicit goals of its own, and a reasonable agent must be able to make obvious inferences and display judgment in making decisions. Typically, a consultant constructed along the lines of a reasonable agent will make a user's goals its own in trying to help that user. However, a reasonable agent is not always compelled to do so. Human consultants will not obligingly give out information to which a user is not entitled or which they suspect will be put to ill use. Similarly, a good consultant might deflect a user's request because the consultant feels that the user does not have an adequate grasp of the domain, has a particular misconception, or is lacking some particular fact. In addition, a good consultant might do something more than simply answer a question. He might take the opportunity to show the user how to do a more general task of which the user's particular request is merely a special case. In all these situations, an action other than simply responding to a request is warranted.

A reasonable agent is ideally suited to handle such a broad class of situations. It does so by deciding what its goals should be in the given situation, and then planning for them. For example, when UC is asked how to crash the system, it forms two goals, one of helping the user to know what he or she wants, and one of protecting the integrity of the system. It then realizes that these two goals are in conflict, and eventually decides the conflict in favor of the latter goal.

Of course, it is possible to achieve by other means various parts of the functionality here attributed to the model of a reasonable agent. For example, one can simply build one component that tries to detect misconceptions, another that checks for requests having to do with crashing the system, yet another to capitalize on opportunities to educate the user, etc. However, the reasonable-agent framework provides a single, flexible control structure in which to accomplish all these task, and, in particular, deal with interactions between them. That is its engineering motivation. Our primary reason for adopting it is that it is our theory about how humans function in consulting situations.

1.3. *Overview*

The structure of this report is as follows. First, we present an outline of the structure of the current version of our consultation system. We follow this with a brief description of KODIAK. The next sections constitute the bulk of this report and are essentially a detailed description of a trace of a rather simple sentence through UC's components. In doing so, the mechanisms of those components that are primarily responsible for UC's agent-like qualities are described. Finally, we conclude with some discussion of the deficiencies of our current design.

1.3.1. *Outline of* UC*'s structure*

UC is comprised of a number of components, which are invoked in a more or less serial fashion.

(1) Language Analysis (ALANA)

Language analysis is that component of the understanding process that computes a representation of the content of an utterance. ALANA, written by Charles Cox, produces a KODIAK representation of the content of an utterance. This representation generally contains only what can be determined from the words and linguistic structures present in the utterance.

In our theoretical framework, we call such an analysis of an utterance its *primal content*. The concept of primal content is related to what is usually described as the *literal meaning* or *sentence meaning* of an utterance. However, unlike literal meaning, the primal content of an utterance involves certain idiomatic interpretations (i.e., it is not necessarily composed from words and general grammatical constructions). Also, the primal content of an utterance may be rather abstract, perhaps so much so that it may not be a suitable candidate for a meaning. For example, the literal meaning of "The cat is on the mat" is generally taken to be a conventional situation in which a cat is resting upon a mat. However, the primal content of this sentence would be more abstract, where the contribution of "on" is identical to that in the primal content of "The light fixture is on the ceiling" or "The notice is on the bulletin board." Presumably, this conveys some sort of support relation. Note that such an abstract content appears never to be in itself the meaning of such an utterance (cf. Searle 1979).

In contrast to primal content is the *actual content* of an utterance. The actual content is context-dependent, generally requires some amount of inference based on world knowledge, and is a suitable candidate for the meaning of an utterance. For example, the actual content of “The cat is on the mat,” without a further context specified, is what the literal meaning of this sentence is generally taken to be. Computing this content from the primal content requires pragmatic knowledge about the kind of support relation a cat and a mat are likely to be in, and requires making an inference that cannot be justified by the meanings of the terms and the grammatical constructions present in the utterance. The primal/actual content distinction is elaborated on in Wilensky (1987).

(2) Inference (Concretion Mechanism)

The particular kind of inference needed to go from a primal content to an actual content sometimes involves a process known as *concretion* (Wilensky 1983). Concretion is the process of inferring a more specific interpretation of an utterance than is justified by language alone. Concretion may involve finding a more specific default interpretation or some other interpretation based on the context. For example, in the example “the cat is on the mat” above, the actual content computed is the default support relation between a cat and a mat. In some compelling context, a quite different actual content may be computed from the same primal content.

(There are other possible relations between primal and actual content besides the latter being a more specific interpretation of the former. For example, a conventionalized metaphor might have a primal content that more closely resembles its literal interpretation but an actual content resembling its metaphoric interpretation. Thus, one analysis of a sentence like “John gave Mary a kiss” will have as its primal content an instance of giving, but as its actual content an instance of kissing. We will not pursue further the details of the primal/actual-content distinction here. This is largely because, in UC, the need for concretion is widespread, and our handling of other kinds of primal/actual content computations is more haphazard.)

In UC, concretion is needed primarily because we need to organize knowledge about more specific interpretations of utterances than can be arrived at through linguistic knowledge alone. For example, if UC is asked the question “How can I delete a file?”, ALANA can represent that this is a question about how to delete a file. But it would not have any reason to assume that deleting a file is a specific kind of deleting. Determining that this is so is likely to be important for several reasons. For example, knowledge about how to delete a file will be found associated with the concept of “file deletion,” say, but not with the concept of deletion in general. Thus UC must infer that “deleting a file” refers to the specific kind of deletion having to do with computer storage in order to perform subsequent tasks like finding plans for accomplishing the user’s request.

In UC, concretion is the function of a special mechanism designed specifically for that purpose by Dekai Wu. The output of the concretion mechanism is another KODIAK representation, generally one containing more specific concepts than that produced by ALANA. Having a specific concretion mechanism is a pragmatic concession. We feel it is unlikely that such a specific mechanism is theoretically warranted. A more justifiable position is that a general inference mechanism should be exploited here, concretion being only one of the kinds of inference such a mechanism accomplishes. A unified text-inference mechanism that accomplishes concretion as well as other forms of inference has been built (Norvig 1987). It is our belief that some mechanism akin to Norvig's should be used in UC in place of a specialized concretion engine, but no attempt has yet been made to do so.

(3) Goal Analysis (PAGAN)

Having computed an actual content for an utterance, UC then tries to hypothesize the plans and goals under which the user is operating. This level of analysis is performed by PAGAN, written by James Mayfield. PAGAN performs a sort of "speech act" analysis of the utterance. The result of this analysis is a KODIAK representation of the network of plans and goals the user is using with respect to UC.

Goal analysis is important in many ways for UC. As is generally well-known, an analysis of this sort is necessary to interpret indirect speech acts, such as "Do you know how to delete a file?", or "Could you tell me how to delete a file?". Furthermore, goal analysis helps to provide better answers to questions such as "Does ls -r recursively list subdirectories?". An accurate response to the literal question might simply be "No." But a better response is "No, it reverses the order of the sort of the directory listing; ls -R recursively lists subdirectories." To produce such a response, one needs to realize that the goal underlying the asking of this question is either to find out what ls -r does, or to find out how to recursively list subdirectories. It is the job of the goal analyzer to recognize that such goals are likely to be behind such a question. More details about PAGAN can be found in Mayfield (1989, 1992, 2000).

(4) Agent (UCEgo)

Having hypothesized what the user wants of it, we would expect a system like UC to do what the user requested. But, as mentioned above, this is not always appropriate. UC should not aid and abet a user trying to perform malicious mischief; it might need to correct an errant user or it might decide to supply unasked-for information to one diagnosed as not knowing an important fact.

In order to deal with such situations UC is constructed as an agent. This agent reacts to users' requests by forming goals and acting on them. The central mechanism of UC is called UCEgo, and has been developed by David Chin.

In a typical transaction, UCEgo will simply adopt the goal of having the user know what the user wants to know. However, as the example above illustrates, UCEgo may adopt other goals as well, such as protecting the integrity of the system. It may also have to detect conflicts between these goals. Sometimes, UCEgo, attempting to be educational, may adopt a somewhat different goal from the user's. Thus, if the user asks UC to actually perform some request, such as telling the user who is on the system, UC will decide to tell the user *how to perform* such a function, rather than do what the user requested.

UCEgo implements much of the agent-like character of UC. While interfaces are generally thought of as passive conduits through which information flows, UC is an agent that listens to the user and is generally helpful. But it has its own agenda, and the requests of the user are merely a source of input to it. More details about UCEgo can be found in Chin (1987, 1991, 1998, 2000a, 2000b).

(5) User Modeling (KNOME)

Several of UC's components may need information about the user to make an effective choice. For example, an expert user certainly knows how to delete a file. Thus, such a user uttering "Do you know how to delete a file?" is unlikely to be asking for this information – more likely this user is testing the consultant's knowledge.

Assessing the knowledge state of the user is the function of a user-modeling program called KNOME, developed by David Chin. It is exploited by several components, including the Expression Mechanism described below. More details about KNOME can be found in Chin (1986, 1987, 1989).

(6) Domain Planner (KIP)

Typically, UCEgo tries to help the user. This usually requires determining a fact that the user would like to know. This task is accomplished by KIP. KIP is a "domain planner" developed by Marc Luria. While UCEgo infers its own goals, and plans to act on them, KIP is given a task by UCEgo of determining how to accomplish what the user wants to accomplish. KIP tries to determine how to accomplish this task, using knowledge about UNIX and knowledge about the user's likely goals. KIP returns a plan, represented in KODIAK. For example, UCEgo may give KIP the task of determining how to move a file to another machine, if this is something the user wants to know. Here, KIP would come up with the plan of copying the file to the target machine and then deleting the original.

Since UCEgo is also a planner, UC in effect has two planners within it. Again, this is probably not theoretically justifiable, although the two planners have ended up focusing on rather different aspects of planning. It remains to be seen whether a single mechanism might accommodate both functions. More details about KIP can be found in Luria (1985, 1987, 1988).

(7) Expression Mechanism (UCExpress)

Having gotten KIP to compute a plan for the user's request, UCEgo now tries to communicate this plan to the user. To do so, it must determine which aspects of the plan are worthy of communication and how best to communicate them. For example, if it is likely that the user knows how to use commands in general, it might be sufficient just to specify the name of the command. In contrast, it might be helpful to illustrate a general command with a specific example.

UCExpress is an "expression mechanism" written by David Chin. It edits out those parts of the conceptual answer returned by KIP that, for various reasons, appear unnecessary to communicate. UCExpress may also choose to illustrate an answer in several formats. For example, it might illustrate a general answer by generating a specific example, or it might explain one command in terms of another, simpler, command.

The result of UCExpress is an annotated KODIAK network, where the annotation specifies which part of the network is to be generated. More details about UCExpress can be found in Chin (1987, 1988, 2000a, 2000b).

(8) Language Production (UCGen)

Once UC has decided what to communicate, it has to put it into words. This is done by a generation program called UCGen. UCGen is a simple generator, programmed by Anthony Albert and Marc Luria. It takes the marked KODIAK network produced by UCExpress and, using knowledge of English, produces sentences to complete the transaction with the user.

(9) Learning Mechanism (UCTeacher)

Since it is intended that UC be an extensible system, a mechanism has been developed to add new knowledge to the system by talking to it in natural language. This mechanism, called UCTeacher, is the work of James Martin. UCTeacher has capabilities to extend both UC's knowledge base of UNIX facts as well as its knowledge of English vocabulary. More details about UCTeacher can be found in Martin (1985, 1986a, 1986b, 1987, 1988, 2000).

Note that several UC components deal with goals and plans, but in rather different ways. To minimize confusion, we emphasize the different tasks that these programs perform: PAGAN hypothesizes goals of the user with respect to UC, UCEgo reacts to the user by forming goals of its own and then trying to accomplish them, and KIP is given the goal of solving some underlying UNIX problem. Thus, PAGAN is a goal analyzer, i.e., a program that attempts to determine someone else's goals, while UCEgo and KIP are planners, i.e., programs that determine and produce plans for their own goals.

UCEgo and KIP exist as two separate programs primarily for pragmatic reasons. It may be possible that a single planner could accomplish the tasks of both UCEgo and KIP. Such an integration will have to await a revision of our system.

1.4. UC *examples*

The first set of examples illustrates the scope of the current UC prototype. They are intended to show the kind of requests the system can handle, as well as its scope. The second set demonstrates some more complex capabilities.

Each sentence below preceded by the prompt '#' is supplied by the user. The rest of the text is a computer-generated response. Note that the system handles a variety of request forms, some direct and some indirect. Also, the responses to "general" questions will cause an example to be generated. Finally, the reader may notice that the vocabulary used to respond to a request is often different from that used in the request itself. For example, a question about file protection is answered in terms of file permission; a question about changing a directory is answered in terms of connecting to a directory. This is because the representations used to encode the system's knowledge are kept in a canonical form, so that different expressions of the same meaning are mapped to the same internal representation. Upon generating a response, the system uses only this internal representation, which, coupled with its generating biases, causes an expression of an answer that is largely independent of the vocabulary that appeared in the request.

1.4.1. *Simple examples*

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

Hi.
How can I help you?
# Can you tell me how to move a file?
Use mv. For example, to move the file named foo to the file named foo1,
type 'mv foo foo1'.
```

```
# Do you know how to delete a file named dekai?
Type ‘rm dekai’.
# Do you know how to delete a file?
Use rm. For example, to delete the file named foo, type ‘rm foo’.
# How can I delete the file charlie?
Type ‘rm charlie’.
# How can I delete the file named tony?
Type ‘rm tony’.
# How can I get rid of the file named boob?
Type ‘rm boob’.
# I want to delete a file named peter.
Type ‘rm peter’.
# I want to delete a file.
Use rm. For example, to delete the file named foo, type ‘rm foo’.
# I want to delete the file dave.
Type ‘rm dave’.
# How can I add read permission to a file?
Use chmod. For example, to add individual read permission to the file
named foo, type ‘chmod u+r foo’.
# How can I add the owner execute permission to a file?
Use chmod. For example, to add individual execute permission to the file
named foo, type ‘chmod u+x foo’.
# How can I change the group execute permission of a file?
Use chmod. For example, to remove group execute permission from the file
named foo, type ‘chmod g-x foo’.
# How can I change the user read permission of a file?
Use chmod. For example, to remove individual read permission from the file
named foo, type ‘chmod u-r foo’.
# How can I remove the group write permission from the file named bimbo?
Type ‘chmod g-w bimbo’.
# How do I change the protection of a file?
Use chmod. For example, to remove individual read permission from the file
named foo, type ‘chmod u-r foo’.
# How can I compact a file?
Use compact. For example, to compact the file named foo, type ‘compact
foo’.
# How can I create a directory?
Use mkdir. For example, to create the directory named foo, type ‘mkdir
foo’.
# How can I find a file?
Use whereis. For example, to find the file named foo, type ‘whereis foo’.
# How can I find out how much disk space I am using?
Type ‘du’.
# How can I find out who is using the system?
Type ‘finger’.
# How can I find out wilensky’s phone number?
Type ‘finger wilensky’.
# How can I list a directory?
Use ls. For example, to list the directory named foo, type ‘ls foo’.
```

```
# How can I logout?
Type ‘logout’.
# Tell me how to leave.
Type ‘logout’.
# How can I move a file to the file named blah?
Use mv. For example, to move the file named foo to the file named blah,
type ‘mv foo blah’.
# How can I move the file junk?
Use mv. For example, to move the file named junk to the file named foo,
type ‘mv junk foo’.
# How do I move a file?
Use mv. For example, to move the file named foo to the file named foo1,
type ‘mv foo foo1’.
# How do I change my directory?
Use cd. For example, to connect to the directory named foo, type ‘cd foo’.
# How do I copy a file?
Use cp. For example, to copy the file named foo to the file named foo1,
type ‘cp foo foo1’.
# How do I copy the file foo to the file blah?
Type ‘cp foo blah’.
# How do I create a directory?
Use mkdir. For example, to create the directory named foo, type ‘mkdir
foo’.
# How do I print a file named info?
Type ‘lpr info’.
# I want to print a file.
Use lpr. For example, to print the file named foo, type ‘lpr foo’.
# I want to change my working directory.
Use cd. For example, to connect to the directory named foo, type ‘cd foo’.
# I want to copy the file named rick.
Use cp. For example, to copy the file named rick to the file named foo,
type cp ‘rick foo’.
# I want to delete a directory.
Use rmdir. For example, to delete the directory named foo, type ‘rmdir
foo’.
# I want to remove a directory.
Use rmdir. For example, to delete the directory named foo, type ‘rmdir
foo’.
# What’s the best way to delete a directory?
Use rmdir. For example, to delete the directory named foo, type ‘rmdir
foo’.
# I want to display a file.
Use more. For example, to display the file named foo, type ‘more foo’.
# I want to edit a file.
Use vi. For example, to edit the file named foo, type ‘vi foo’.
# I want to find out Joe’s address.
Type ‘finger joe’.
# I want to remove a directory named junk.
Type ‘rmdir junk’.
```

```
# What’s the command to compact a file?
Use compact. For example, to compact the file named foo, type ‘compact
foo’.
# What is a directory?
A directory is a file that is used to contain files.
# What is a file?
A directory is a container that is used to contain text, code or files.
# What is a rm?
Rm is a command that is used to delete files.
# What is a search path?
A search path is a list of directories that is used by the csh to search
for programs to execute.
```

1.4.2. *More complex examples*

```
Welcome to UC (Unix Consultant) version 3.23
To a UC ‘#’ prompt, please type in your questions about the Unix file
system in English.
To leave, just type a ‘ˆD’ 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’.
```

In this example, UC hypothesized that the user asked the question in order to get an answer to some underlying question. In particular, PAGAN hypothesized possible goals of wanting to know how to rename a file and wanting to know what ‘rn’ does. Since the answer to the direct question is negative, UCEgo realizes that the underlying goals are not addressed by this answer, and, therefore, attempts to answer them. This results in the extended answer.

```
# How can I crash the system?
I’m sorry, I cannot tell you.
```

In this example, UC detected a conflict between its goal of helping the user and that of preserving the integrity of the system. It resolved this conflict in favor of the latter goals. However, since it could not accommodate the user’s request, it attempts to be polite by apologizing to the user.

```
# What does ls -v do?
ls does not have a -v option.
```

Here the user asks about a non-existent command option. Actually, the UNIX ‘ls’ command ignores options that it does not recognize, so the consultant could have answered truthfully by saying that ‘ls -v’ causes one’s directory to be listed. It did not, because PAGAN interpreted the request as a question about the function associated with the command. UCEgo reasons that, since (1) it doesn’t know about such an option, and (2) it knows all the options to all simple commands, and (3) ‘ls’ is a simple command, the user must have a misconception. It then generates the goals of correcting the misconception, which results in the answer shown. Note that UC uses an open world hypothesis together with metaknowledge to determine whether it knows some facts about UNIX.

In the next example, two different sessions with two different users are represented.

```
Welcome to UC (Unix Consultant) version 3.23
To a UC ‘#’ prompt, please type in your questions about the Unix file
system in English.
To leave, just type a ‘ˆD’ 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 rwho do?
Rwho is used to list all users on the network, the users’ tty, the users’
login time, and the users’ idle time.
```

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

Hi.
How can I help you?
# How can I find out the inode of a file?
Use ls -i.
# What does rwho do?
Rwho is like who, except rwho lists all users on the network.
```

Because he asked such a simple question, the first user was diagnosed by KNOME as being a novice user. This resulted in the answer to his first question being illustrated by an example, and the extended format of his answer to second. In contrast, the second user asked about a relatively technical aspect of UNIX. Thus, this user is likely to be a relatively sophisticated user. As a result, the answer to that user’s first question was not illustrated with an example. For the same reason, UC presumes that the second user knows about the ‘who’ command, and takes advantage of this knowledge in generating the response.

Note this example also illustrates UC’s ability to generate different types of responses. This example was able to use a *simile* format, because the command to be generated was thought to be sufficiently close to one UC believes the user to already know. Formatting is discussed in Section 3.6.

The next example illustrates some aspects of UC’s ability to generate plans.

```
# How do I print Jim's file on the lineprinter?
Use lpr. This plan will not work if you don't have read permission on
Jim's file.
```

In this example, KIP, UC’s domain planner, suspects that the plan it is suggesting might fail. This is because the plan involves printing someone else’s file, and KIP knows that the lack of appropriate permission in such cases is a likely source of plan failure. However, since it does not know a more certain plan, it delivers the suspect one, but warns the user of the possible problem.

A fact about a possible problem with a plan is called a *concern*. Concerns are explicitly represented in UC’s knowledge base. At this stage, UC’s knowledge of concerns is fragmentary. Thus, questions above that have the same structure as this one may not cause any warnings to be generating simply because the requisite knowledge is lacking. Concerns are discussed further in Luria (1985).

```
# Who is on the system?
I'm sorry, I don't know. Use users to find out who is on the system.
```

Here UC fails to answer the user’s question, because it does not have any way of running UNIX commands itself. Its failure causes it to apologize. UC then forms the goal of the user knowing how to determine the requested information. It addresses this goal, which leads to the second part of the response.

2. KODIAK

The knowledge used by UC is represented in KODIAK, as are the various stages of the processing of an utterance. KODIAK (Keystone to Overall Design for Integration and Application of Knowledge) is an implementation of CRT (Cognitive Representation Theory), an approach to knowledge representation that bears similarities to numerous other systems, but especially

Table 1. KODIAK legend.

Representation	Meaning
	NODES
$\boxed{\text{CONCEPT}}$	An absolute
○	A relation
$\boxed{\text{CONCEPT}n}$, where *n* is an integer	$\boxed{\text{CONCEPT}n}$ is an instance of $\boxed{\text{CONCEPT}}$.
	LINKS
The following is priority ordered (i.e. use first match for meaning):	
$\boxed{\text{CATEGORY}}-\alpha \rightarrow$ ○	○ is an aspectual, i.e., a relation in which $\boxed{\text{CATEGORY}}$ participates.
$\boxed{\text{CATEGORYA}}-\text{D}\rightarrow\boxed{\text{CATEGORYB}}$	$\boxed{\text{CATEGORYB}}$ dominates $\boxed{\text{CATEGORYA}}$.
$\boxed{\text{INSTANCE}}-\text{I}\rightarrow\boxed{\text{CATEGORY}}$	$\boxed{\text{INSTANCE}}$ is an instance of $\boxed{\text{CATEGORY}}$.
○$-\text{C}\rightarrow\boxed{\text{CATEGORY}}$	The argument to ○ is constrained to be of type $\boxed{\text{CATEGORY}}$.
○$-\text{V}\rightarrow\boxed{\text{OBJECT}}$	The value of the argument to ○ is $\boxed{\text{OBJECT}}$.
○$-{=}\rightarrow$○	The first aspectual is constrained to have the same value as the second.
$\boxed{\text{CATEGORY}}-rel\rightarrow$○	○ is an aspectual of $\boxed{\text{CATEGORY}}$, and ○ specializes *rel*, an aspectual of some concept dominating $\boxed{\text{CATEGORY}}$.
$\boxed{\text{CATEGORYA}}-rel\rightarrow\boxed{\text{CATEGORYB}}$	Each member of $\boxed{\text{CATEGORYA}}$ participates in *rel* with some member of $\boxed{\text{CATEGORYB}}$.

those of Schank (1975), Schubert (1976), Shapiro (1979), and Brachman and Schmolze (1985). KODIAK differs from these systems in what we believe are significant ways. However, here we shall try to trivialize rather than accentuate these differences so that the reader can relate our representations to more familiar ones. The reader should consult Wilensky (1986) for a more detailed account and justification of this representation system. Table 1 summarizes the notation used in KODIAK diagrams.

2.1. UNIX *knowledge in* UC

The KODIAK knowledge representations used in UC include several rather general notions, such as state change, goal, and action, plus many specific facts about UNIX. The complete collection is too lengthy to include here. (UC is currently constructed from approximately 200 KODIAK diagrams, consisting of about 1000 absolutes and 2,000 relations. While a scope of a

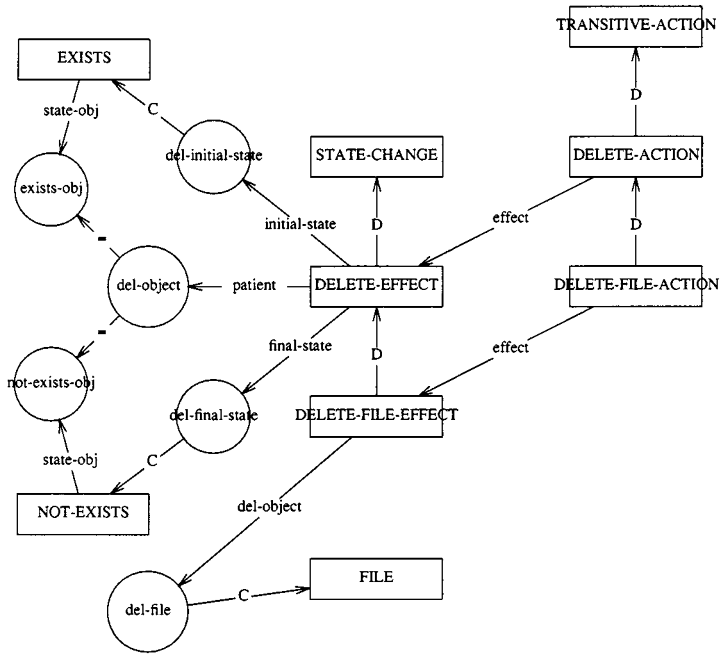

Figure 1. Knowledge about deletion represented in KODIAK.

diagram is to some degree arbitrary, diagrams roughly correspond to definitions of meaningful entities, like the definition a particular command.) Some more important concepts used in modeling the domain will be explained in the individual sections of this report.

To facilitate understanding the KODIAK diagrams that follow, consider the representation that the UC’s knowledge base contains about the UNIX ‘rm’ command. This is used to delete a file named by its argument. Figure 1 shows how knowledge about deletion is represented in UC. The central node in this diagram is DELETE-EFFECT. DELETE-EFFECT is shown as being a kind of STATE-CHANGE that causes a something to go from existence to non-existence. (The notions of existence and negation do play a special role in KODIAK, but also exist as ordinary states, as is the case here. From the point of view of this example, these are just like any other states.)

DELETE-EFFECT specifies the “minimal” deletion event. For example, it says nothing about the cause of such an event, or who the actor of it may be. In UC in particular and in CRT in general, such state changes are the bases from which we build more elaborate concepts. For example, the action of deleting something is represented as an action that causes something to be deleted.

This is encoded by showing DELETE-EFFECT to be the effect of DELETE-ACTION.

Using names like DELETE-EFFECT may be somewhat misleading. In particular, DELETE-EFFECT is not required to be the effect of anything – while DELETE-ACTION is defined as having DELETE-EFFECT as its result, this statement imposes a requirement on DELETE-ACTION, not on DELETE-EFFECT. We call such concepts EFFECTs rather than EVENTs, say, to emphasize that we mean to include only the most essential elements of the concept, i.e., just a specialized state change.

DELETE-FILE-EFFECT is a specialized version of DELETE-EFFECT in which the object deleted is constrained to be a file. DELETE-ACTION is correspondingly specialized to DELETE-FILE-ACTION. It is also shown as being a kind of TRANSITIVE-ACTION. This is a very general category denoting any action that acts upon an object.

This sort of structure, in which there exists parallel effect and action hierarchies, with the effect hierarchy carrying most of the semantic weight, is typical of the representation that appear in UC.

Figure 2 connects this general knowledge about deletion with knowledge about UNIX. Here we state how to achieve a DELETE-FILE-EFFECT. This accomplished by the node labeled PLANFOR2, which points to EXECUTE-UNIX-RM-COMMAND and to DELETE-FILE-EFFECT. A PLANFOR indicates that something is conceptualized as a plan for a particular goal (PLANFORS are discussed below). In other words, this notation represents the particular fact that the ‘rm’ command (i.e., the command whose name is “rm”) is used to achieve the effect of deleting a file. Again, this structure is typical of that seen in UC – most of the information about a command is represented as information about the use of that command; the intended function of the use of a command is represented by a PLANFOR between a node representing the use of the command and some effect.

The rest of the diagram specifies the format of the command. In particular, the **Equate** link specifies that, to delete a particular file, its name must be the same as that of the argument supplied to ‘rm’.

3. A Tour Through UC

The following sections describe the components of UC in more detail. To aid in understanding how these components contribute to the processing of an individual utterance, we show how each section processes the example sentence “Do you know how to print a file on the imagen?”[2] In most cases, a module is capable of doing a great deal more than is required for this example, and such capabilities are attested to. However, the example is

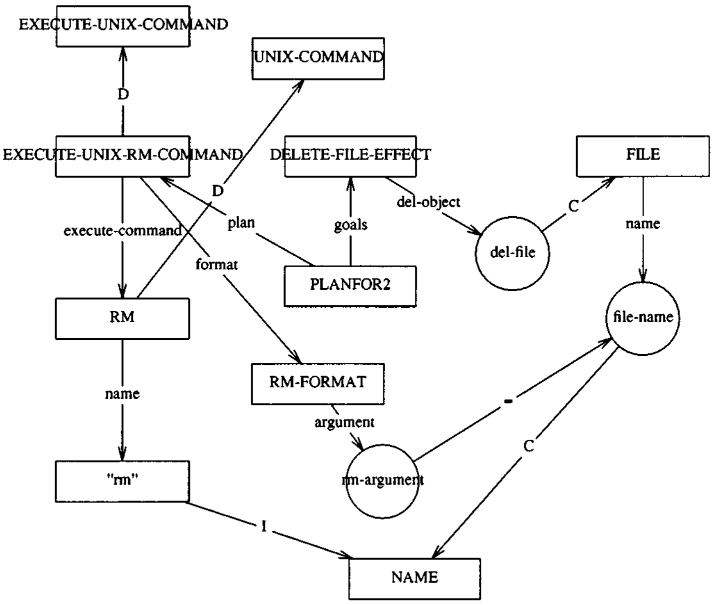

Figure 2. Knowledge about deleting files in UNIX represented in KODIAK.

useful for illustrating the kind of processing that is performed for a typical request.

In order to produce a paper of reasonable length, we reduced considerably the description of some of UC's modules. We have focused on those processes that contribute to UC's agent-like nature, while some more conventional modules, such as the conceptual analyzer, are mentioned only in passing. References are given to descriptions of these neglected components, which have all appeared elsewhere in print, should the reader find the account herein dissatisfying.

3.1. *The analyzer*

A conceptual analyzer maps a string of words into a meaning representation. ALANA (Augmentable LANguage Analyzer), the conceptual analyzer for UC, takes as input a sentence typed by a user, and builds a conceptual representation using the KODIAK knowledge-representation language. ALANA constructs the "primal content" of the input utterance. The primal content is the interpretation that can be computed from grammatical and lexical knowledge; it is generally rather abstract. ALANA's results are further interpreted and refined by other parts of the system, such as the concretion mechanism, to produce an "actual content," and the goal analyzer, to produce a representation of the intentions underlying the utterance.

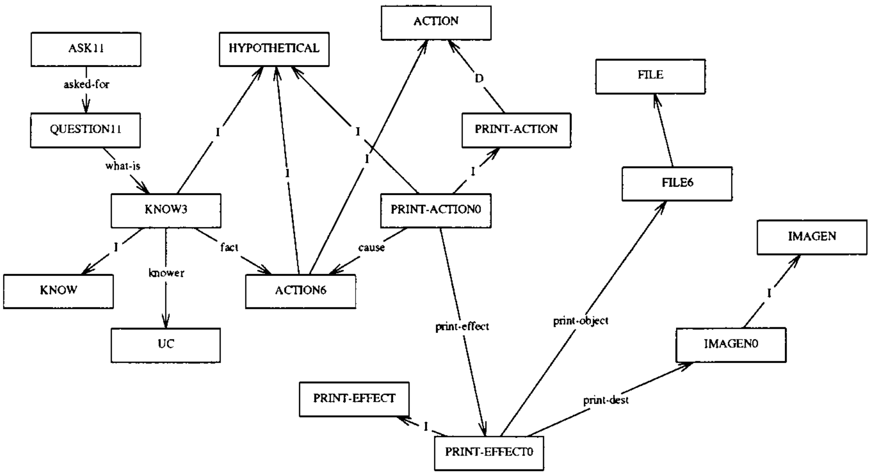

Figure 3. ALANA's output for "Do you know how to print a file on the Imagen?"

ALANA is a descendent of PHRAN (Wilensky and Arens 1980), the front-end natural-language component for the original UC (Wilensky et al 1984). Like PHRAN, ALANA reads the user's input and forms a concept that the other UC components can use for their tasks. Also like PHRAN, ALANA uses as its primitive knowledge unit the *pattern-concept pair*, which relates a natural-language structure to a conceptual structure. UC has a total of 476 patterns and knows 284 words.

ALANA differs from PHRAN in its generality. ALANA generalizes on the idea of pattern-concept-pair analysis, while making it easier than it was with PHRAN for a knowledge-adder to add new patterns to the system. Since a more detailed description of ALANA can be found in (Cox 1986), we will not elaborate on it here. Instead, we merely show in Figure 3 the output produced by ALANA upon reading the sentence "Do you know how to print a file on the imagen?".

This diagram may be interpreted as follows: The entire request is summarized as ASK11, i.e., some asking event. What is asked for is verification of some item, QUESTION11, whose content is KNOW3, i.e., an instance of knowing. The knower of the item is UC, and the fact is ACTION6. ACTION6 is interpreted as something that is the cause of a printing action PRINT-ACTION0, which is itself an action whose effect (PRINT-EFFECT0) is to cause a file (FILE6) to be printed on an imagen printer (IMAGEN0).

Some of the nodes in this diagram point to a node labeled HYPOTHETICAL. This is a tentative convention used to indicate that the knowing and printing event, etc., are not real events, but merely hypothetical ones.

3.2. *The concretion mechanism*

As mentioned previously, our theoretical posture is that concretion is but one of a number of inference processes that can be accomplished by a single mechanism. However, in UC, for reasons of efficiency, and for pragmatic advantages, a separate concretion mechanism was implemented (by Dekai Wu). This mechanism currently does rather straightforward classification.

The mechanism concretes by using information about inheritance and value constraints, as well as by considering relation information between concepts. A concept represented as an instance of a category is passed to the concretion mechanism. Its eligibility for membership in a more specific subcategory is determined by its ability to meet the constraints imposed on the subcategory by its associated relations and aspectual constraints. If all applicable conditions are met, the concept becomes an instance of the subcategory. At the same time, the relations in which the concept participates may be concreted to reflect the more specific relations of the new category of which it is inferred to be a member.

Parts of the representation of printing are shown in Figure 4. Besides the printing of the contents of a computer file, PRINT-EFFECT is in principle applicable to other types of printing, such as printing a newspaper or a book. The concretion mechanism checks each of the more specific concepts dominated by PRINT-EFFECT, searching for one whose constraints can be satisfied by the input. It finds PRINT-FILE-EFFECT, who only additional constraint is that its print-object must be a file. Since PRINT-EFFECT0 is in print-object relation with the object FILE6, which is indeed an instance of FILE, the process can descend to this node. The concretion process will continue until it can concrete no further.

Of course, it is perfectly plausible just to preclude from UC on engineering grounds interpretations of words that do not occur in the UNIX domain. As we suggested earlier, it is our preference not to do so, since we wish to address, rather than finesse, fundamental language issues. However, doing so would not really eliminate the need for concretion. Even if we do not include concepts of non-computer printing in our knowledge base, we would still have many different kinds of printing, e.g., printing ASCII files versus binary files or printing on the lineprinter versus the laser printer. A query about each of these kinds of printing requires a different response, although the term “printing” applies to all of these. A system like UC needs to concrete the concept of printing in general to the particular kinds of printing that it knows about, in order to find the knowledge needed to answer the question. Thus, eliminating interpretations that lie outside the domain simplifies the problem somewhat, but it does not change its essential nature.

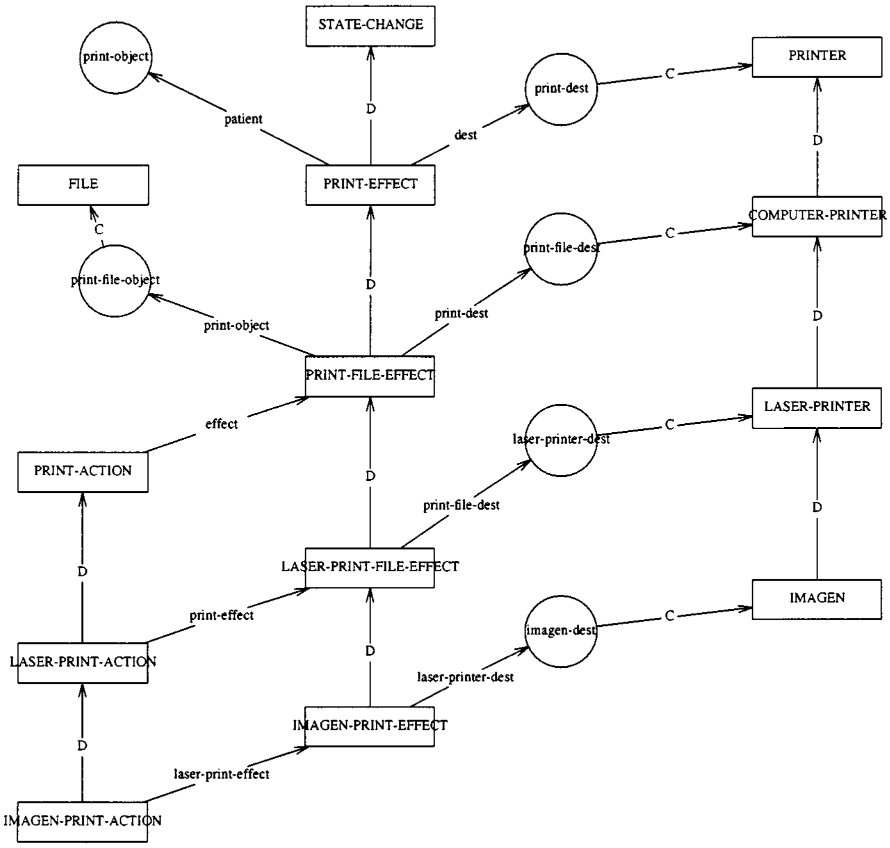

Figure 4. Some knowledge about printing.

In general, when concretion occurs, some node is reclassified as being an instance of a more specific category, and, in addition, the relations predicated about that node are also reclassified. For example, here we concrete PRINT-EFFECT0 to an instance of PRINT-FILE-EFFECT. At the same time, we should concrete the relation print-object predicated about it to a use of the more specific relation print-file-object. Similarly, print-dest is concreted to print-file-dest.

Continuing in this fashion, the mechanism can move from PRINT-EFFECT to LASER-PRINT-EFFECT, and finally to IMAGEN-PRINT-EFFECT, since the print-dest of the input is IMAGEN0, which is an instance of IMAGEN. At the same time, the relation print-dest is concreted to imagen-dest. In parallel with this concretion, the node PRINT-ACTION0 gets concreted to an instance of IMAGEN-PRINT-ACTION. The final result is shown in Figure 5.

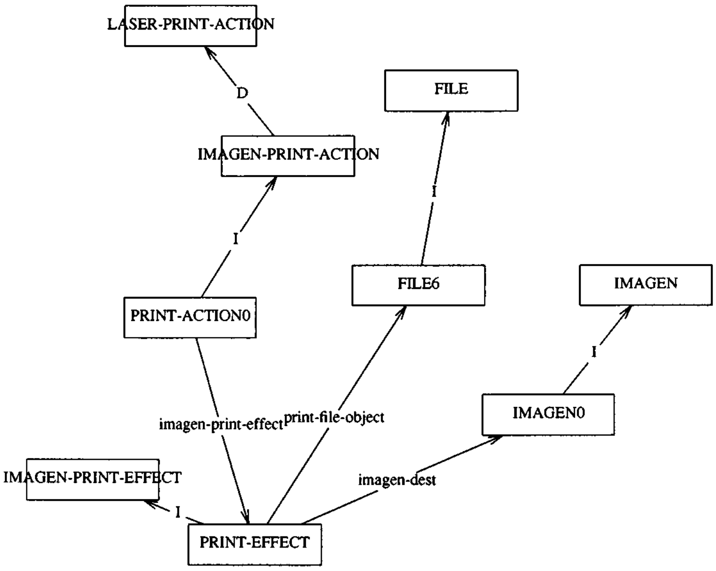

Figure 5. Result of concreting "print file on the imagen."

3.3. *The goal analyzer*

Once an utterance has been converted to a KODIAK representation by ALANA, and has been further refined by the concretion mechanism, this internal representation is passed to PAGAN (Plan And Goal ANalyzer). PAGAN's task is to determine what goals the speaker is addressing in making the utterance. For example, when given a representation of the utterance "Do you know how to print a file on the imagen?" asked by a naive user, PAGAN should infer that the user was using the utterance to address the goal of knowing how to print a file on the imagen. Note that PAGAN is not responsible for detecting goals that are held by the speaker but that are not conveyed by the speaker's utterances. This problem is addressed by the ego mechanism and by the planner.

To successfully do goal analysis, at least two questions must be answered. The first concerns the utterance in isolation:

> What kind of act does this utterance constitute?

This question has traditionally fallen under the rubric of "speech-act theory" (Austin 1962; Searle 1969). For example, "Do you know how to print a file on the Imagen?" potentially has both a direct and indirect interpretation which PAGAN must choose between.

The second question a goal-analysis mechanism must answer examines the role of the utterance in conversation:

```
1.  This utterance wasn't expected.
2.  This utterance wasn't an adopted plan.
3.  Matching ASK0 pattern against ASK11.
4.  Could not match  KNOW3 pattern to ACTION because of category KNOW.
5.  Match failed -- try the next one.
6.  Matching ASK34 pattern against ASK11.
7.  Match found.
8.  Matching ASK39 pattern against ASK11.
9.  Match found.
10. Attempting to resolve ambiguity in the interpretation of ASK11.
11. The alternatives are: KNOW34 KNOW39.
12. Trying to determine whether KNOW34 was expected.
13. KNOW34 was not expected.
14. Trying to determine whether KNOW39 was expected.
15. KNOW39 was not expected.
16. The goal KNOW39 is implausible,
    since the speaker probably believes that it already holds.
17. ASK11 is explained by the goal KNOW34.
18. Creating new HAS-GOAL node: HAS-GOAL-ga0.
19. Returning goal KNOW-ga0.
```

Figure 6. Trace of PAGAN's processing of "Do you know how to print a file on the imagen?"

How does this utterance relate to other utterances?

By virtue of being an action, an utterance always occurs within a context. This context includes such diverse factors as the identities of the speaker and of the audience, the social relationship between them, the physical locale, the task the conversation is supplementing if any, and so on. One feature of this context that is salient to goal analysis is the presence of conventional, multi-utterance sequences. Consider the exchange:

1. Do you have write permission on the parent directory?
2. Yes.

The ability to understand the full meaning of (2) is contingent on the realization that it relates directly and conventionally to (1). Thus, PAGAN will require knowledge of such sequences to correctly determine the goal underlying utterances such as (2).

The input to PAGAN is the structure built by the analyzer from this utterance and refined by the concretion mechanism. A trace of PAGAN as it processes this structure is shown in Figure 6.

The first step performed by PAGAN is to determine whether the utterance is the continuation of a conversational plan already in effect. For this to be the case, there would need to be some previous dialogue to provide the necessary context. This dialogue would take one of two forms. It might be a plan that UC believed the user to be pursuing *before* the current utterance was encountered. Alternatively, it could be a plan introduced by UC that the user has adopted, that UC believes the user to be pursuing only after witnessing the current

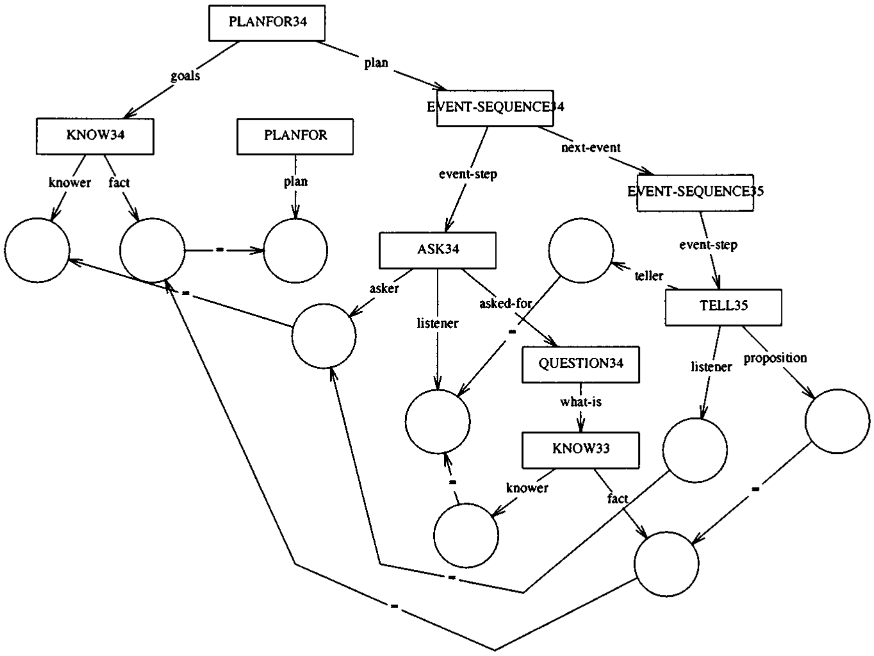

Figure 7. A plan for knowing is to ask if the hearer knows.

utterance. Since there is no previous context in the example we are tracing, neither of these possibilities is found to hold (1–2).

Next, PAGAN tries to match the utterance against the first steps of plans in its planfor knowledge base. The first possibility is compared with the input structure (3), but one pair of corresponding nodes is found not to match (4–5). The second possibility, one that does match the utterance, is then compared with the input structure (6–7). This planfor corresponds to the indirect interpretation of the utterance. This is the planfor that is shown in Figure 7. A third possibility, corresponding to the direct interpretation of the utterance, also matches the input structure (8–9). An attempt to resolve this ambiguity is now made (10–11). Since neither goal matches an expected goal (12–15), the planfors are examined for plausibility. The direct interpretation is discarded, because the user model indicates that it is likely that the user knows that UC knows how to print a file on the imagen (16). (More details about the user model can be found in Chin (1986, 1989).) Thus, the planfor representing the indirect interpretation is selected (17).

Once the utterance has been matched to the first step of this planfor, an instance of a PLANFOR is created with the goals determined from the input. In addition, an instance of the HAS-GOAL state is built (18). The planner of this state is the user, and the goal is the goal of the PLANFOR. This HAS-

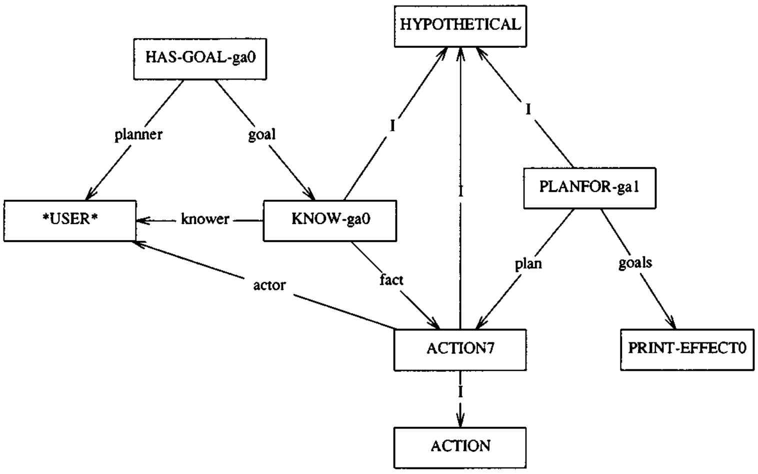

Figure 8. PAGAN output.

GOAL represents the goal that UC believes the user had in mind in making the utterance, and is returned by PAGAN as its result (19). It is shown in Figure 8. In this figure, note that PAGAN has created a node labeled ACTION7, whose actor is the user. This represents the inference made by the goal analyzer that, if a user wants to know an action to achieve some goal, then the user intends to be the actor of that action.

3.4. *The ego mechanism*

Thus far in the processing, UC has parsed and understood the question, and the goal analyzer has asserted that the user has the goal of knowing a plan for printing a file on the imagen. At this point, UCEgo's processing begins.

Figure 9 shows a trace of the goal-detection phase of UCEgo. In the trace, UC-HAS-GOAL19 represents UC's goal of helping the user (HELP2). HAS-GOAL-ga0, which is also shown in Figure 8, is the user's goal of knowing (KNOW-ga0) how to print a file on the imagen. PLANFOR29 represents the fact that a plan for helping the user (HELP2) is for UC to satisfy KNOW-ga0, which is the user knowing how to print a file on the imagen. UC-HAS-INTENTION6 represents UC's intention to satisfy KNOW-ga0. UC-HAS-GOAL20 represents UC's goal of the user knowing how to print a file on the imagen.

The user's goal (HAS-GOAL-ga0 in the trace) combines with UC's goal of helping the user (UC-HAS-GOAL19, present in UC's initial state) to activate the fact (PLANFOR29) that a plan for helping the user is for UC to satisfy the goal of the user knowing a plan for printing a file on the imagen. Next,

```
UCego detects the following concepts:
(UC-HAS-GOAL19 (goal = (HELP2 (helpee = *USER*) (helper = UC))))
(HAS-GOAL-ga0 (planner = *USER*)
              (goal = (KNOW-ga0 (knower = *USER*)
                                (fact = (ACTION7 (actor = *USER*))))))
and asserts the following concept into the database:
(PLANFOR29 (goals = (HELP2 (helpee = *USER*) (helper = UC)))
           (plan = (SATISFY2 (need = (KNOW-ga0 \&))
                             (actor = UC))))
UCego detects the following concepts:
(UC-HAS-GOAL19 (goal = (HELP2 (helpee = *USER*) (helper = UC))))
(PLANFOR29 (goals = (HELP2 (helpee = *USER*) (helper = UC)))
           (plan = (SATISFY2 (need = (KNOW-ga0 \&))
                             (actor = UC))))
and asserts the following concept into the database:
(UC-HAS-INTENTION6 (intention = (SATISFY2 (need = (KNOW-ga0 \&))
                                          (actor = UC))))
UCego detects the following concepts:
(UC-HAS-INTENTION6 (intention = (SATISFY2 (need = (KNOW-ga0 \&))
                                           (actor = UC))))
and asserts the following concept into the database:
(UC-HAS-GOAL20 (goal = (KNOW-ga0 (knower = *USER*)
                                 (fact = (ACTION7 (actor = *USER*))))))
```

Figure 9. Trace of UCEgo's goal-detection process.

this planfor combines with UC's goal of helping the user (UC-HAS-GOAL19) to make UCEgo adopt the intention (UC-HAS-INTENTION6) of satisfying the goal of "the user knowing a plan for printing a file on the imagen." This intention arises as a result of UCEgo's if-detected demon for plan selection. Finally, UCEgo adopts the user's goal as its own. This subgoal (UC-HAS-GOAL20) is the result of UCEgo's goal detection process.

After UCEgo has detected the goal of "the user knowing a plan for printing a file on the imagen," the plan-selection phase of UCEgo attempts to select a plan to satisfy this goal. Figure 10 shows a trace of this part of the processing. In the trace, UC-HAS-GOAL20 is UC's goal of knowing (KNOW-ga0) ACTION7, which represents the plan part of the planfor (PLANFOR-ga1) for printing a file on the imagen (PRINT-EFFECT0). UNIX-planner1 represents a call to KIP. PLANFOR70 says that a plan for achieving the goal of PRINT-EFFECT0 is to use EXECUTE-UNIX-IPR-COMMAND0, which entails using the command 'lpr -Pip' on the name of the file to be printed. ANSWER-FOR1 says that an answer to the query of "how to print a file on the imagen" (ACTION7) is PLANFOR70. PLANFOR30 says that a plan for achieving the goal of the user knowing how to print a file on the imagen (KNOW-ga0) is for UC to tell (TELL4) the user PLANFOR70. UC-HAS-INTENTION7 represents UC's intention of telling the user (TELL4). UCexpress1 represents a call to UCExpress to execute TELL4.

```
UCego detects the following concepts:
(PLANFOR-ga1 (goals = PRINT-EFFECT0) (plan = (ACTION7 (actor = *USER*))))
(UC-HAS-GOAL20 (goal = (KNOW-ga0 (knower = *USER*) (fact = (ACTION7 (actor = *USER*))))))
and asserts the following concept into the database:
(UNIX-planner1 (user-goals = PRINT-EFFECT0))
The planner is passed:
PRINT-EFFECT0
The planner produces:
(PLANFOR70 (goals = PRINT-EFFECT0)
           (plan = (EXECUTE-UNIX-IPR-COMMAND0
                    (ipr-file = FILE6)
                    (ipr-execute-command = "lpr -Pip")
                    (ipr-format = (IPR-FORMAT0 (ipr-format-arg = NAME6))))))
UCego detects the following concepts:
(ANSWER-FOR1
   (answer = (PLANFOR70 (goals = PRINT-EFFECT0)
                        (plan = (EXECUTE-UNIX-IPR-COMMAND0
                                 (ipr-file = FILE6)
                                 (ipr-execute-command = "lpr -Pip")
                                 (ipr-format = (IPR-FORMAT0 (ipr-format-arg = NAME6))))))
   (query = (ACTION7 (actor = *USER*))))
(UC-HAS-GOAL20 (goal = (KNOW-ga0 \&)))
and asserts the following concept into the database:
(PLANFOR30 (goals = (KNOW-ga0 \&))
           (plan = (TELL4 (listener = *USER*)
                          (speaker = UC)
                          (proposition = (PLANFOR70 \&)))))
UCego detects the following concepts:
(UC-HAS-GOAL20 (goal = (KNOW-ga0 \&)))
(PLANFOR30 (goals = (know-ga0 \&)) (plan = (TELL4 \&)))
and asserts the following concept into the database:
(UC-HAS-INTENTION7 (intention = (TELL4 \&)))
UCego detects the following concepts:
(UC-HAS-INTENTION7 (intention = (TELL4 \&)))
and asserts the following concept into the database:
(UCexpress1 (gen-prop =
             (TELL4 (listener = *USER*)
                    (speaker = UC)
                    (proposition =
                     (PLANFOR70 (goals = PRINT-EFFECT0)
                                (plan = (EXECUTE-UNIX-IPR-COMMAND0
                                         (ipr-file = FILE6)
                                         (ipr-execute-command = "lpr -Pip")
                                         (ipr-format = (IPR-FORMAT0 (ipr-format-arg =
                                                             NAME6))))))))))
```

Figure 10. A trace of UCEgo's plan selection process.

The first step of the plan is to call KIP, the planner component of UC. Figure 11 shows the if-detected demon that calls KIP. KIP is called whenever UC has the goal of knowing a plan for something In the trace, UC-HAS-GOAL20 and PLANFOR-ga1 combine to cause a call to KIP (UNIX-planner1 in the trace). KIP, as is describing is the next section, comes back with an answer (PLANFOR70), which is an answer (ANSWER-FOR1) to the user's question. Then UCEgo detects the plan of telling the user the answer (PLANFOR30 in the trace). This plan, with UC's goal (UC-HAS-GOAL20) of the user knowing

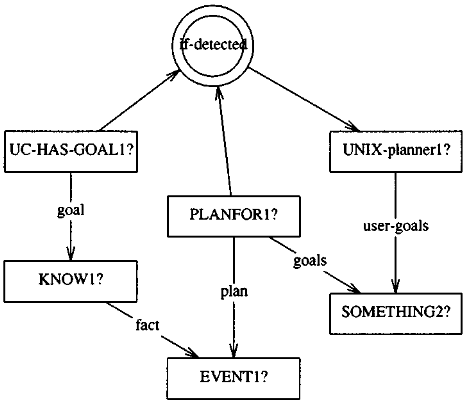

Figure 11. If-detected demon for calling the UC planner, KIP.

the answer leads to UC's intention to tell the user the answer (UC-HAS-INTENTION7). Finally, the intention translates into a call to UC's expression mechanism, UCExpress (UCexpress1 in the trace), which eventually calls UCGen to produce the answer. KIP, UCExpress, and UCGen are described in the following sections.

3.5. *The planner*

This section describes KIP (Luria 1985), a knowledge-based, common-sense planner (Wilensky 1983). KIP includes:

- a knowledge base of facts about the domain;
- a planning component that uses this knowledge to:
  - find potential plans for problem situations;
  - notice potential problems with these plans;
  - use metaplanning knowledge (knowledge about plans) to determine which plans to suggest.

KIP uses the same knowledge base as the rest of UC. In principle, it could be used to do the planning required by UCEgo. As mentioned previously, this was not attempted mostly for pragmatic reasons. In addition, the planning done by UCEgo is much more straightforward and does not require recourse to the same magnitude of knowledge as does KIP. Thus, it seems reasonable to use a much simpler planner for communicative functions.

The basic structure of KIP is similar to that of UCEgo. However, KIP is a more elaborate planner. It must be able to plan for unanticipated goals of the user, and must be concerned with adverse consequences of the plans it proposes. In general, KIP has to iterate through the planning process a number of times to arrive at an adequate plan.

The following are the steps of the iterative process that KIP uses:

1. Goal detection – decide what goals to work on:
   - Start with the goals input from UCEgo.
   - Detect new goals that might arise from use of proposed plans.
2. Plan selection – select a possible plan:
   - Try to find a stored plan that is related to the user's goals.
   - Propose a new plan if necessary based on knowledge in the system.
3. Projection – test whether plan would be executed successfully:
   - Check for conditions that are likely not to be met.
   - Notice bad side effects.
   - Detect goals needed to address discovered problems
4. Plan evaluation – decide whether plan is acceptable:
   - Reject plan if impossible.
   - Accept if unproblematic.
   - Create warning if possible but problematic.

The iterative structure described here is implemented by a series of metaplans (Wilensky 1983). The underlying metaplan is to find a particular plan that the user can use; these steps are parts of that process.

The example being considered throughout this paper does not require much work by KIP. This is because UC's knowledge base contains a plan for the goal of printing a file on the imagen. Also, there are no concerns associated with this particular plan. (Had the input referred to printing the file of another user, or printing on another printer, KIP would have retrieved concerns about not being about to access another persons' files, or about the peculiarities of the type of printer, respectively.) Thus, the planner simply ends up retrieving the stored plan, making an instance of it, and passing it on. The input to the planner from UCEgo is PRINT-EFFECT0, which we show in Figure 12 along with its most pertinent connections. Figure 13 shows an annotated trace of KIP's processing for this example. KIP is passed a single goal, PRINT-EFFECT0. After checking for goal conflict concerns, the planner checks to see if it has achieved the exact same goal before. When it fails, as is the case here, it looks up the hierarchy. In this example, it looks first at the category IPRINT-EFFECT, of which PRINT-EFFECT0 is represented as being an instance. KIP finds in the knowledge base PLANFOR7 (not shown in the trace) that connects IPRINT-EFFECT with EXECUTE-UNIX-IPR-COMMAND. This knowledge is shown in Figure 14.

There are no concerns retrieved with this particular plan, so KIP will simply produce an instance of it and return it to UCEgo.

Note that KIP's retrieval of a plan for this goal is simplified by the concretion mechanism's having classified the input as an instance of "imagen printing." Originally, ALANA represented the meaning of the input as an

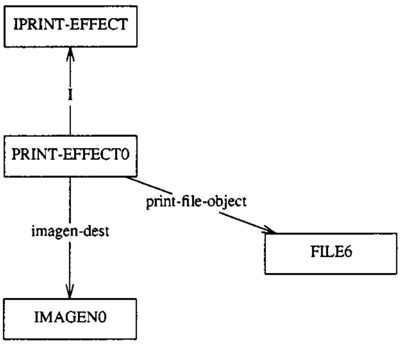

Figure 12. Input to KIP from “Do you know how to print a file on the imagen?”: PRINT-EFFECT0 and important links.

```
Planner is passed:
(PRINT-EFFECT0 (print-file-object = FILE6)
               (imagen-dest = IMAGEN0))
Looking for stored plan for PRINT-EFFECT0
Looking for specific stored-plan for this individual goal
Looking for plan for PRINT-EFFECT0
No stored plan for PRINT-EFFECT0 was found
Try all the parents to see if they have a plan
Looking for plan for IPRINT-EFFECT
EXECUTE-UNIX-IPR-COMMAND is a plan for IPRINT-EFFECT
No condition concerns
No goal conflict concerns
Making a new instance of EXECUTE-UNIX-IPR-COMMAND
Making a new instance of IPR-FORMAT
Making a new instance of NAME
Making NAME6 an instance of HYPOTHETICAL
Filling in aspectual ipr-format-arg with value NAME6
FILE6
Filling in aspectual name with value NAME6

Making a new instance of EXECUTE-UNIX-IPR-COMMAND
Filling in aspectual ipr-file with value FILE6
Filling in aspectual ipr-execute-command with value "lpr\ -Pip"
Filling in aspectual ipr-format with value IPR-FORMAT0

Making a new instance of PLANFOR7
Filling in the goals with PRINT-EFFECT0
Filling in the plan with EXECUTE-UNIX-IPR-COMMAND0

Planner returns:
PLANFOR70
Planner produces:
(PLANFOR70 (goals = PRINT-EFFECT0)
           (plan = (EXECUTE-UNIX-IPR-COMMAND0
                    (ipr-file = FILE6)
                    (ipr-execute-command = "lpr\ -Pip")
                    (ipr-format = (IPR-FORMAT0 (ipr-format-arg = NAME6))))))
```

Figure 13. Trace of KIP’s processing.

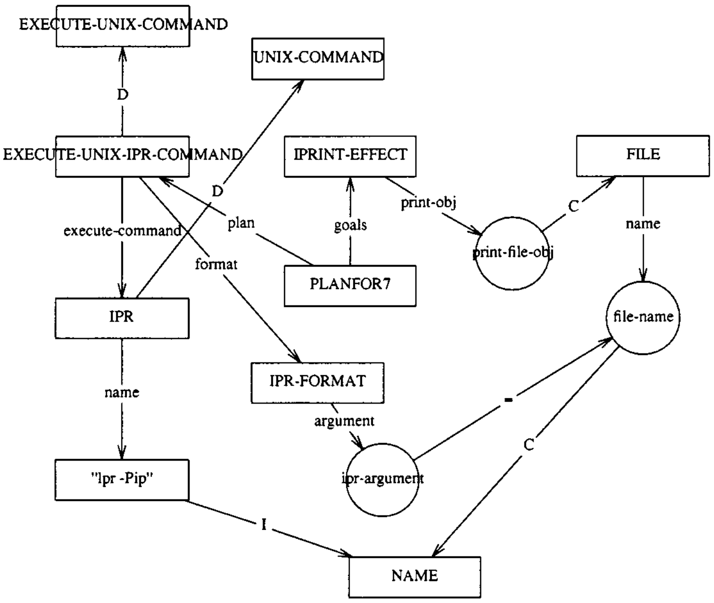

Figure 14. Representation of plan for printing a file on the imagen.

instance of printing. However, there are no plans associated with printing per se, but only with printing's more specific subcategories. Thus, the planner would have to search for an appropriate plan had the concretion mechanism not done its job. In actuality, the planner starts at the most specific category that the input could be classified as, and works up the hierarchy. Therefore, the planner is guaranteed to find the most specific plan applicable to the situation. Note that KIP has made NAME6 a hypothetical object, because it is the name of FILE6, which is itself hypothetical. After KIP has made a filled-in copy of the plan, no pertinent concerns are found, and there is nothing more to do, so the plan found is returned. The planner's output is shown in Figure 15.

3.6. *The expression mechanism*

After UCEgo, in conjunction with KIP, has determined the proper answer, it calls UCExpress to express it to the user. The input to UCExpress is shown in Figure 16. This input conceptual network is UCEgo's plan to tell the user the plan that KIP produced (Figure 15). If the plan were to be directly generated into English, the result might be something like:

> A plan for printing a file on the imagen printer is to use the lpr command with the imagen printer option. The format of the command is "lpr -Pip" and followed by the name of the file to be printed on the imagen printer.

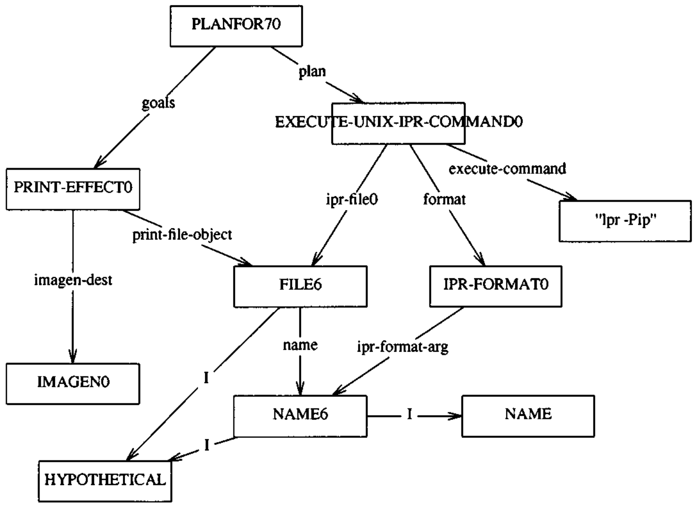

Figure 15. Output from KIP for example "Do you know how to print a file on the imagen?"

Instead, UCExpress prunes the output, and, since the user is suspected of being a novice, generates an example to illustrate the plan. The following trace illustrates the processing of UCExpress:

The first phase of UCExpress is pruning, during which those concepts that the user already knows are marked so that the generator will not generate them. In this case, PRINT-EFFECT0, the goal of PLANFOR70, is pruned, as it is in the current context.

In traversing the input conceptual network, UCExpress runs into the command-format IPR-FORMAT0. Since this node is presumably not known to the user, UCExpress would normally not prune it. However, UCExpress specifically checks command-formats to see if they are as specific as possible, and since FILE6 does not have a name, UCExpress schedules the creation of an example to explain the format, and prunes the original node. The rest of the network is passed without any pruning.

In creating an example, UCExpress must specify all the parameters in the command format. Thus, the name of the file, which was not specified by the user, is made explicit in the example. Here, the name "foo" was chosen arbitrarily from a list. The complete example is then turned into the proposition part of a TELL (TELL5 in the trace).

Figure 18 shows the conceptual network after pruning and the addition of an example. Note that the exemplified of an EXAMPLE is automatically pruned, as it is always expressed by the expressing the generic response. The pruned and augmented conceptual network is next passed to the generator, described in the next section, which produces the following English output:

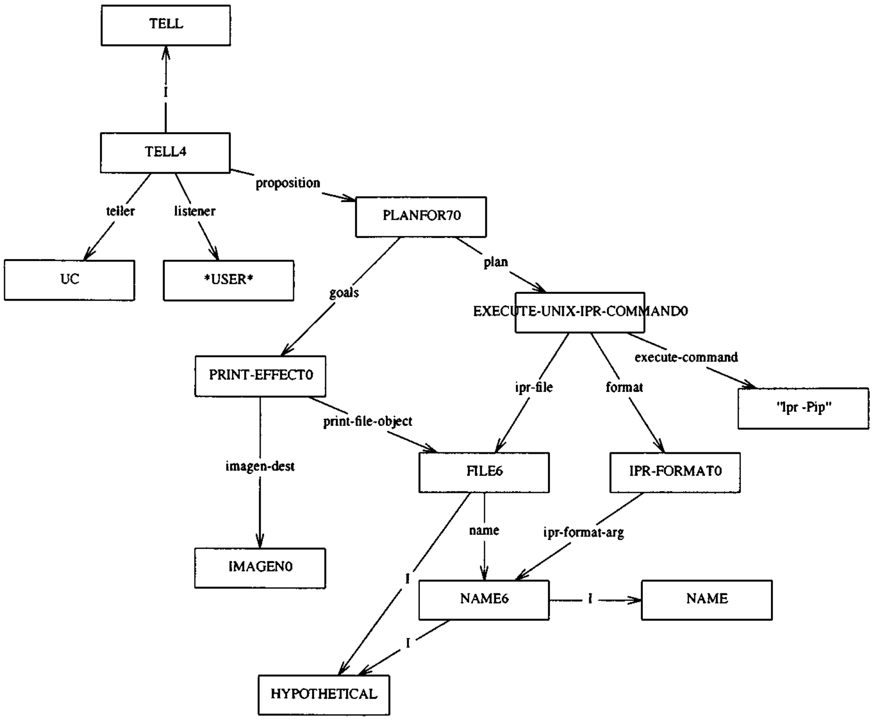

Figure 16. The input to UCExpress for example "Do you know how to print a file on the imagen?"

```
Express:  now expressing the PLANFOR,
(PLANFOR70 (goals = PRINT-EFFECT0)
           (plan = (EXECUTE-UNIX-IPR-COMMAND0
                    (ipr-file = FILE6)
                    (ipr-execute-command = "lpr -Pip")
                    (ipr-format = (IPR-FORMAT0 (ipr-format-arg = NAME6))))))
Express:  not expressing PRINT-EFFECT0, since it is already in the context.
Express:  creating an example for the incomplete plan, IPR-FORMAT0
Express:  choosing a name, foo, for an example file.
Express:  created the example(s):
((TELL5 (teller = UC)
        (listener = *USER*)
        (proposition =
         (EXAMPLE0 (exampler = (PLANFOR31 (goals = PRINT-EFFECT4)
                                          (plan = (TYPE-ACTION0
                                                   (actor = *USER*)
                                                   (type-string =
                                                    (CONCAT-STR0 (string1 = "lpr -Pip")
                                                                 (string2 = "foo")))))))
                   (exemplified = PLANFOR70)))))
```

Figure 17. Trace of UCExpress.

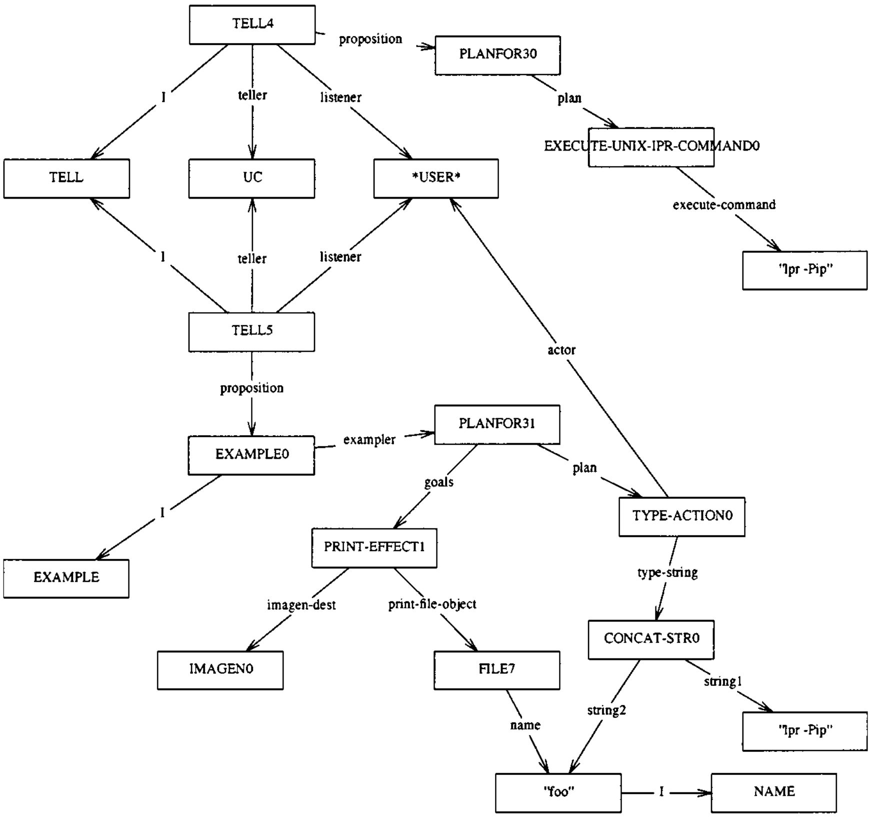

Figure 18. The output of UCExpress for the example "Do you know how to print a file on the imagen?"

> Use lpr -Pip. For example, to print the file named foo on the imagen printer, type 'lpr -Pip foo'.

If the user had been judged to be at least a beginner in experience, then the command-format would also have been pruned. This is because KNOME believes that users at the beginner level and up can be assumed to know that part of the command format. In such a case, the entire output would consist of the pruned version of TELL4, and the generated entire response would be:

> Use lpr -Pip.

3.7. *The generator*

After UCExpress formats an answer, the generator, UCGen, converts the conceptual response into text. The current version of UCGen has been customized to work with the types of responses that the system typically produces. It has been built to take advantage of the limited structure of these responses.

To convert a KODIAK representation of a concept into text, UCGen must associate some linguistic information with the concept. This is done by attaching to a concept a pattern that represents some linguistic form. For example, a concept often expressed is PLANFOR. This concept relates a plan for achieving a goal with the goal itself. A pattern for PLANFOR is:

To (gen goals) comma (gen plan).

This pattern might be used to generate the sentence:

To delete a file, use rm.

This is somewhat akin to the pattern-concept pair construct in PHRED (Jacobs 1984), or to KING's REF links (Jacobs 1985), although the KODIAK representation accommodates different methods for fetching patterns.

Patterns mix words and punctuation with function calls. In the above example, 'gen' is a function that will be called with argument 'goals' and later with argument 'plan'. In general, the arguments to functions that are found in generator patterns are the aspectuals associated with the concept to which the pattern is attached. In this example, the aspectuals of PLANFOR, 'goals' and 'plan,' are arguments to gen.

The pattern given above for PLANFOR is the most general one for that concept. That is, it is the pattern used when both the goals and the plan are to be expressed. As described in the previous section on UCExpress, it is not always necessary to express both of these parts. For example, two answers to 'How do I delete a file?' are:

1. To delete a file, use rm.
2. Use rm.

The expression mechanism puts a flag on each aspectual that it does not want expressed. Consequently, associated with each concept may be zero or more patterns, one for each combination of aspectuals that are to be expressed. PLANFOR is associated with the general pattern shown above, as is the pattern '(gen plan)', which is applicable to the case where only the plan is to be expressed.

When a concept to be output is given to the generator, those KODIAK concepts that either dominate or are categories for the concept are searched for one that has an attached generator pattern. If no pattern is found, and the concept is an aspectual, then the value for the aspectual is sent to the generator. The first pattern found is applied to the concept to be expressed to produce an English sentence. Words in the pattern are output as is. Punctuation and function calls must go through further processing. For example, in the pattern ‘To (gen goals) comma (gen plan)’, the word ‘To’ is output directly, whereas the (gen ...) function calls must be evaluated, and the ‘comma’ will be converted to a ‘,’.

This generator is easy to understand and extend, and is well integrated with the rest of UC; it shares the KODIAK representation and concepts used by the rest of the system. Some weaknesses are that the overall structure is top down; i.e., only those concepts that are expected to exist are expressed. In general, a generator should be able to handle arbitrary permutations of conceptual relationships. Also, this generator uses little linguistic knowledge. With more complicated utterances, the simple pattern strategies employed so far would become inadequate.

This section describes how the output is delivered by UC in response to the question, ‘Do you know how to print a file on the imagen?’ A diagram of some of the relevant knowledge structures is given in Figure 19. A trace produced while generating this output is given in Figure 20.

The expression mechanism of UCEgo first passes TELL4 to the generator. Only the proposition part of the TELL will be expressed, so its value, PLANFOR70, is passed to the generator’s main routine, ‘gen.’ PLANFOR70 is dominated by PLANFOR, so the pattern for PLANFOR is retrieved. Since the goals aspectual of PLANFOR70 is marked to be omitted from the response by the expression mechanism, only the plan will be expressed. The pattern found is ‘(gen plan)’. The value of the plan aspectual, EXECUTE-UNIX-IPR-COMMAND0, is sent to ‘gen’. The pattern for this concept is found under EXECUTE-FILE-COMMAND and is ‘use (gen execute-command)’. The value of execute-command aspectual of EXECUTE-UNIX-IPR-COMMAND0 is ‘lpr -Pip.’ The first response is therefore:

> Use lpr -Pip.

Next, the generator is passed TELL5. Once again, only the proposition is to be expressed, so EXAMPLE0 is to be generated. The pattern, found under EXAMPLE, is ‘for example comma (gen exampler)’. This sets up a recursive call to gen with the value of the exampler relation as argument. This value is PLANFOR31.

Once again, a PLANFOR is to be generated. This time, however, both the plan and goals will be expressed. The pattern is ‘to (gen goals) comma (gen plan)’. The value of the goals aspectual is PRINT-EFFECT1. The pattern is found under LAS-PR-EFFECT, and is ‘print (las-pr-file-obj) on the (las-pr-dest-obj)’.

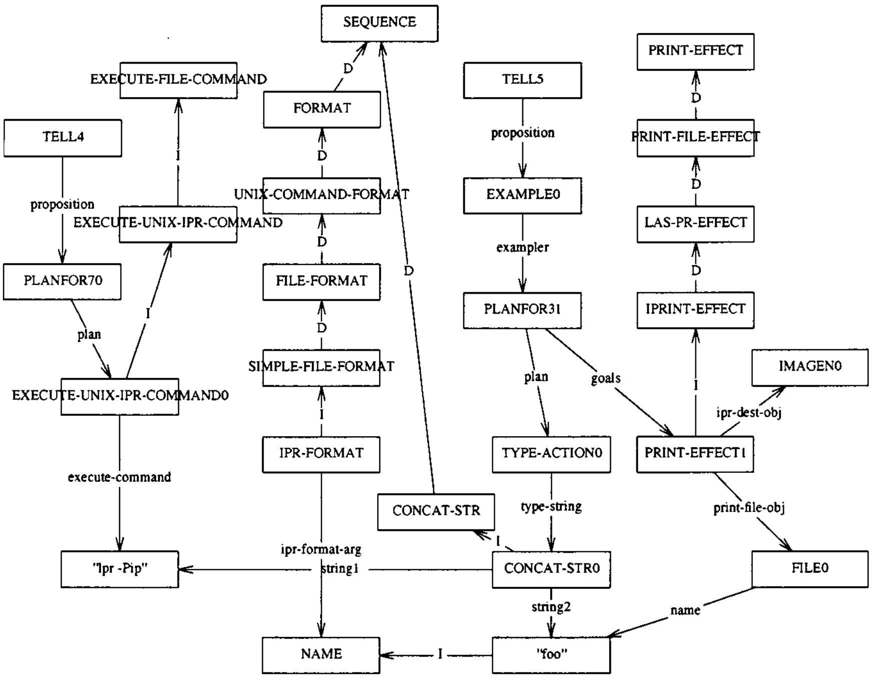

Figure 19. Knowledge pertaining to generating response to “Do you know how to print a file on the imagen?”

The las-pr-file-obj specification causes the generator to find on object in this relation to PRINT-EFFECT1, in this case, FILE0. One of the patterns for FILE is ‘file named (gen name)’. This pattern is applicable if the object in question has a name. Here, FILE0 has the name ‘foo’. When UCGen is generating the name of an object, it uses a heuristic to decide which, if any article to precede the object description with. In general, UCGen will use definite articles if it can.

The other part of the output from the goals aspectual is from the second half of the pattern: ‘on the (las-pr-dest-obj)’. Here the value found is IMAGEN0. The pattern for IMAGEN0 is just ‘imagen’. Again, the generator with supplement this description with a definite article.

The value of the ‘plan’ aspectual for PLANFOR31 is TYPE-ACTION0. The pattern for this concept is from TYPE-ACTION and is ‘type lquote (gen type-string) rquote’. The value for the type-string aspectual of TYPE-ACTION0 is CONCAT-STR0. The pattern is from SEQUENCE and is ‘(gen step) (gen next)’. Here, the step is ‘lpr -Pip’ (i.e., the string1 of CONCAT-STR0), and the next is the name of the file, ‘foo’ (the string2 of CONCAT-STR0). Thus, the output for this call to the generator is:

```
Pattern for PLANFOR70 is: ((gen plan))
Value for plan is: UNIX-IPR-COMMAND0
Pattern for UNIX-IPR-COMMAND0 is: (use (execute-command))
Phrase derived from (execute-command) is: (lpr -Pip)
Phrase derived from (gen plan) is: (use lpr -Pip)

Use lpr -Pip.

Pattern for EXAMPLE0 is: (for example comma (gen exampler))
Value for exampler is: PLANFOR31
Pattern for PLANFOR31 is: (to (gen goals) comma (gen plan))
Value for goals is: PRINT-EFFECT1
Pattern for PRINT-EFFECT1 is:
(print (las-pr-file-obj) on the (las-pr-dest-obj))
Value for las-pr-file-obj is: FILE0
Pattern for FILE0 is: (file named (gen name))
Value for name is: "foo"
Phrase derived from (file named (gen name)) is: (file named foo)
Phrase derived from (las-pr-file-obj) is: (the file named foo)
Value for las-pr-dest-obj is: IMAGEN0
Pattern for IMAGEN0 is: (imagen)
Phrase derived from (las-pr-dest-obj) is: (the imagen)
Phrase derived from (gen goals) is:
(print the file named foo on the imagen)
Value for plan is: TYPE-ACTION0
Pattern for TYPE-ACTION0 is: (type lquote (gen type-string) rquote)
Value for type-string is: CONCAT-STR0
Pattern for CONCAT-STR0 is: ((gen step) (gen next))
Value for step is: lpr -Pip
Pattern for lpr -Pip is: (lpr -Pip)
Phrase derived from (gen step) is: (lpr -Pip)
Value for next is: foo
Pattern for foo is: (foo)
Phrase derived from (gen next) is: (foo)
Phrase derived from (gen type-string) is: (lpr -Pip foo)
Phrase derived from (gen plan) is: (type lquote lpr -Pip foo rquote)
Phrase derived from (gen exampler) is:
(to print the file named foo on the imagen comma type lquote lpr -Pip foo rquote)

For example, to print the file named foo on the imagen,
type 'lpr -Pip foo'.
```

Figure 20. UCGen trace for the example question.

For example, to print the file named foo on the imagen, type ‘lpr -Pip foo’.

4. Problems

As the preceding sections describe, there are many technical problems yet to be resolved for each component of UC. However, several problems appear to be more pervasive.

One general problem is the integration of the components of the system. Control flows unidirectionally through UC. However, there are several cases in which this control structure is unsatisfactory. One such problem is the relation of language analysis and inference. We believe it is cognitively correct that these components function concurrently to produce an interpretation of an utterance, whereas in UC they function serially.

For example, consider again the process of understanding the sentence we have been using in our extended example: "Do you know how to print a file on the imagen?". This utterance is syntactically ambiguous in the attachment of the prepositional phrase "on the imagen." Syntactically, this may modify "you" or "a file" as well as "print." UC does not deal with this ambiguity, because one of ALANA's patterns for "print" specifically looks for "on" followed by a device. However, a more elaborate analyzer would probably not include specific information that relates this preposition to the verb, but rather would try to relate them on more general principles. In such a system, the ambiguity would be a more difficult problem.

Our current approach is to build such a system and use a marker-passing algorithm (Norvig 1987) to help suggest which syntactic combination to try. For example, our knowledge about printing is such that a path between printing and a device designed for printing should be easy to find. In contrast, there would be a less obvious connection between imagen and file, or imagen and the referent of "you." This "conceptual closeness" would suggest trying to relate printing and the imagen with a grammatical pattern, so the correct interpretation would be arrived at without other interpretations being tested.

Properly done, such a marker-passing scheme would effect concretion as well. For example, to arrive at the connection between printing and the imagen, it is probable that one needs to access the node for "computer-printing." Thus, it seems that concretion should not be a separate inference process, but one of several kinds of inference that are performed by a marker-passing mechanism. We are currently attempting to reform the analyzer and the inference mechanism in the direction described.

It seems that the sort of unidirectional architecture we have employed has drawbacks elsewhere in the system. There are situations in which it seems that one component should be allowed to fail, and the failure be propagated back to another component. For example, consider processing the following query:

How can I edit Joe's file?

Initially, the goal analyzer may interpret this request literally. Then the planner may fail, because the file may be protected from just such an action. It seems reasonable, however, for a consultant to suggest copying the file and editing the copy. For this to happen, control must be returned to the goal analyzer, which needs to hypothesize yet another goal underlying the goal it may have suggested initially. We are attempting to design a control structure that accommodates this flow of control.

The concretion mechanism and the goal analyzer also appear to interact in important ways. For example, consider the following example:

What does ls -v do?

Above we showed the UC can respond appropriately to this question by uttering “There is no -v option to the ls command.” However, the question is problematic because another response to it might be “It lists the contents of the current directory.” This response is possible because, although there is no ‘-v’ option to the ‘ls’ command, it is a characteristic of this command that it ignores options it does not recognize.[3]

To produce the desired response, the system must recognize that the intent of the question is something like “Tell me the conventional function of the command ls -v,” and not “Tell me what actually happens when we type ls -v.” One way to phrase this is that “conventional function” and “effects occurring from” are two kinds of “doing.” There are certainly other kinds as well. For example, the same form may refer to the steps of a process.

Therefore, it would appear to be the job of the concretion mechanism to select the appropriate interpretation. However, it seems that the concretion mechanism cannot choose this interpretation without some knowledge of typical user goals. For example, if a user is debugging a program, it would probably be appropriate to interpret the question as referring to the steps incurred in the process rather than to the process’s purpose. But reasoning about the user’s goals is the job of the goal analyzer, which normally is not invoked until the concretion mechanism has completed its task.

The problem is avoided in the current implementation by not allowing for the other, less obvious interpretations at all. However, the example illustrates the need to have more communication between the concretion mechanism and the goal analyzer. Put more strongly, the example suggests that these distinctions between language analyzer, concretion mechanism, and goal analyzer are somewhat artificial. At this stage of our work, it is difficult to determine whether we simply want modules that interact more or a more radical control structure that integrates all these functions.

There are several other more specific deficiencies of which we are aware. As we discussed previously, patterns were built into ALANA on an “as needed” basis. We are attempting to produce a more accurate language specification as we develop the inference component. Also, a mechanism for doing ellipsis, which ran in a previous version of UC, has yet to be integrated into this one.

Undoubtedly, there are many deficiencies that we have not yet discovered. For example, we recently discovered that asking the same question twice resulted in no answer at all being generated for the second request. The problem turned out to be that the user model, after a question is answered, updates its model of the user to show that the user now knows this information. The second time around, this knowledge allowed the expression mechanism to prune away the entire answer, as it inferred the user already knew it. Our approach to fixing this problem is to add another demon that will detect asking for the same thing twice. Then plans for responding appropriately with this situation could be brought to bear.

One important deficiency of our current system is that it still doesn't participate in real conversations. It is our intention that UC function as a consultant and not as a front end to a data base of facts about UNIX. But our current system performs little more than this. Much of the machinery is in place, in UCEgo and PAGAN in particular, to accommodate some conversational situations. We expect much of our further development to be in this direction.

Finally, although we have found that our current representation is advantageous, there are many representational issues that remain unresolved. In particular, it is difficult to express certain aspects of quantification in KODIAK. In UC, one often wants to represent facts like "all files have names" or "most directories are not empty." We are currently working on extending KODIAK to be able to represent such notions in a cognitively plausible way.

Acknowledgements

This research was sponsored in part by the Defense Advanced Research Projects Agency (DoD), ARPA order No. 4871, monitored by Space and Naval Warfare Systems Command Command under contract N00039-84-C-0089, by the Office of Naval Research under contract N00014-80-C-0732, by the National Science Foundation under grant MCS79-06543, and by the Office of Naval Research under contract N00014-97-1-0578.

Notes

[1] UNIX is a trademark of The Open Group.
[2] A kind of laser printer used at our site.
[3] This "feature" has been changed to produce an error message on some versions of UNIX.

References

Allen, J. F. & Perrault, C. R. (1980). Analyzing Intention in Utterances. *Artificial Intelligence* **15**: 143–178.

Allen, J. F., Frisch, A. M. & Litman, D. J. (1982). ARGOT: The Rochester Dialogue System. In Proceedings of *The Second National Conference on Artificial Intelligence*, 66–70. Pittsburgh, PA: AAAI Press.

Austin, J. L. (1962). *How To Do Things With Words.* Harvard University Press: Cambridge, MA.

Brachman, R. & Schmolze, J. (1985). An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science* **9**: 171–216.

Brown, J. S. & Burton, R. R. (1976). A Tutoring and Student Modelling Paradigm for Gaming Environments. In *Symposium on Computer Science and Education*, 236–246. Anaheim, CA.

Carberry, S. (1983). Tracking User Goals in an Information-Seeking Environment. In Proceedings of *Third National Conference on Artificial Intelligence*, 59–63. Washington, DC: AAAI Press.

Chin, D. N. (1986). User Modeling in UC, the UNIX Consultant. In Proceedings of *CHI '86. Human Factors in Computing Systems*, 24–28. Boston, MA: ACM Press.

Chin, D. N. (1988). Exploiting User Expertise in Answer Expression. In Proceedings of *The Seventh National Conference on Artificial Intelligence*, 756–760. Saint Paul, MN: AAAI Press.

Chin, D. N. (1989). KNOME: Modeling What the User Knows in UC. In Kobsa, A. & Wahlster, W. (eds.) *User Models in Dialog Systems*, 74–107. Springer-Verlag: Berlin.

Chin, D. N. (1987). *Intelligent Agents as a Basis for Natural Language Interfaces*. Ph.D. diss., Computer Science Division, University of California, Berkeley, CA. Also available as UCB/CSD 88/396, Computer Science Division, University of California, Berkeley, CA.

Chin, D. N. (1991). Intelligent Interfaces as Agents. In Sullivan, J. W. & Tyler, S. W. (eds.) *Intelligent User Interfaces*, 177–206. ACM Press: Reading, MA.

Chin, D. N. (1998). Intelligent Interfaces as Agents. In Maybury, M. T. & Wahlster, W. (eds.) *Readings in Intelligent User Interfaces*. R. Morgan-Kaufmann: Los Altos, CA.

Chin, D. N. (2000a). Strategies for Expressing Concise, Helpful Answers. In Hegner, S., McKevitt, P., Norvig, P. & Wilensky, R. (eds.) *Intelligent Help Systems for UNIX*. Kluwer Academic Publishers: Dordrecht, The Netherlands (this volume).

Chin, D. N. (2000b). Planning Intelligent Responses in a Natural Language System. In Hegner, S., McKevitt, P., Norvig, P. & Wilensky, R. (eds.) *Intelligent Help Systems for UNIX*. Kluwer Academic Publishers: Dordrecht, The Netherlands (this volume).

Cohen, P. R. & Levesque, H. J. (1987). *Persistence, Intention, and Commitment*. SRI International Technical Report 415.

Cox, C. A. (1986). *ALANA: Augmentable LANguage Analyzer*. UCB/CSD 86/283, Computer Science Division: University of California, Berkeley, CA.

Davis, R. (1976). *Applications of Meta Level Knowledge to the Construction, Maintenance and Use of Large Knowledge Bases*. STAN-CS-76-552, Computer Science Dept.: Stanford University, Stanford, CA.

Deering, M., Faletti, J. & Wilensky, R. (1982). *Using the PEARL AI Package*. UCB/ERL M82/19, Computer Science Division: University of California, Berkeley, CA.

Ernst, G. & Newell, A. (1969). *GPS: A Case Study in Generality and Problem Solving*. Academic Press: New York.

Fikes, R. E. & Nilsson, N. J. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence* **2**(3–4): 189–208.

Grice, H. P. (1975). Logic and conversation. In Cole, P. & Morgan, J. L. (eds.) *Studies in Syntax* **III**, 41–58. Seminar Press: New York.

Grosz, B. & Sidner, C. L. (1985). *The Structures of Discourse Structure*. CSLI–85–39, Center for the Study of Language and Information: Stanford University, Stanford, CA.

Haas, N. & Hendrix, G. (1980). An Approach to Acquiring and Applying Knowledge. In Proceedings of *First National Conference on Artificial Intelligence*, 235–239. Stanford, CA: AAAI Press.

Jacobs, P. S. (1984). *PHRED: A Generator for Natural Language Interfaces*. UCB/CSD 84/189, Computer Science Division: University of California, Berkeley, CA.

Jacobs, P. S. (1985). *A Knowledge-Based Approach to Language Production*. Ph.D. diss., Computer Science Division, University of California, Berkeley, CA. Also available as UCB/CSD 86/254, Computer Science Division, University of California, Berkeley, CA.

Kaplan, S. J. (1983). Cooperative Responses from a Portable Natural Language Database Query System. In Brady, M. & Berwick, R. C. (eds.) *Computational Models of Discourse*. MIT Press: Cambridge, MA.

Lakoff, G. & Johnson, M. (1980). *Metaphors We Live By*. University of Chicago Press: Chicago, IL.

Litman, D. J. & Allen, J. F. (1984). A Plan Recognition Model for Clarification Subdialogues. In Proceedings of *Tenth International Conference on Computational Linguistics*, 302–311. Palo Alto: MIT Press.

Luria, M. (1982). Dividing up the Question Answering Process. In Proceedings of *The Second National Conference on Artificial Intelligence*, 71–74. Pittsburgh, PA: AAAI Press.

Luria, M. (1985). Commonsense Planning in a Consultant System. In Proceedings of *1985 IEEE International Conference on Systems, Man, and Cybernetics*, 602–606. Tucson, Arizona: IEEE Press.

Luria, M. (1987). Goal Conflict Concerns. In Proceedings of *Tenth International Joint Conference on Artificial Intelligence*, 1025–1031. Milan, Italy: Morgan Kaufmann Publishers.

Luria, M. (1988). *Knowledge Intensive Planning*. UCB/CSD 88/433, Computer Science Division: University of California, Berkeley, CA.

Martin, J. (1985). Knowledge Acquisition through Natural Language Dialog. In Proceedings of *Second Annual Conference on Artificial Intelligence Applications*, 728–733. Miami, FL: AAAI Press.

Martin, J. (1986a). Views from a Kill. In Proceedings of *Eighth National Conference of the Cognitive Science Society*, 728–733. Amherst, MA.

Martin, J. (1986b). Representing and Acquiring Knowledge about Metaphors. In Proceedings of Third Workshop on Theoretical Issues in Conceptual Information Processing, 49–54. Philadelphia, PA.

Martin, J. (1987). Understanding New Metaphors. In Proceedings of *Tenth International Joint Conference on Artificial Intelligence*, 137–139. Milan, Italy: Morgan Kaufmann Publishers.

Martin, J. (1988). *A Computational Theory of Metaphor*. Ph.D. diss., Computer Science Division, University of California, Berkeley, CA. Also available as UCB/CSD 88/495, Computer Science Division, University of California, Berkeley, CA.

Martin, J. (2000). Representing UNIX Domain Metaphors. In Hegner, S., Mc Kevitt, P., Norvig, P. & Wilensky, R. (eds.), *Intelligent Helps Systems for UNIX*. Dordrecht, The Netherlands: Kluwer Academic Publishers (this volume).

Mayfield, J. (1989). *Goal Analysis: Plan Recognition in Dialogue Systems*. Ph.D. diss., Computer Science Division, University of California, Berkeley, CA. Also available as UCB/CSD 89/521, Computer Science Division, University of California, Berkeley, CA.

Mayfield, J. (1992). Controlling Inference in Plan Recognition. *User Modeling and User-Adapted Interaction* **2**(1–2): 83–115.

Mayfield, J. (2000). Evaluating Plan Recognition Systems: Three Properties of a Good Explanation. In Hegner, S., Mc Kevitt, P., Norvig, P. & Wilensky, R. (eds.), *Intelligent Helps Systems for UNIX*. Dordrecht, The Netherlands: Kluwer Academic Publishers (this volume).

Mays, E. (1980). Failures in Natural Language Systems: Applications to Data Base Query Systems. In Proceedings of *First National Conference on Artificial Intelligence*, 327–330. Stanford, CA: AAAI Press.

McCoy, K. F. (1983). Correcting Misconceptions: What to Say when the User is Mistaken. In Proceedings of *CHI '83 Conference*, 197–201. Boston, MA: ACM Press.

Morik, K. & Rollinger, C-R. (1985). The Real Estate Agent – Modeling the User by Uncertain Reasoning. *AI Magazine* **6**(2): 44–52.

Newell, A. & Simon, H. A. (1972). *Human Problem Solving*. Prentice-Hall: Englewood Cliffs, NJ.

Norvig, P. (1983). Frame Activated Inferences in a Story Understanding Program. In Proceedings of *Eighth International Joint Conference on Artificial Intelligence*, 624–626. Karlsruhe, West Germany: Morgan Kaufmann Publishers.

Norvig, P. (1987). *A Unified Theory of Inference for Text Understanding*. Ph.D. diss., Computer Science Division, University of California, Berkeley, CA. Also available as UCB/CSD 87/339, Computer Science Division, University of California, Berkeley, CA.

Rich, E. (1979). User Modeling via Stereotypes. *Cognitive Science* **3**: 329–354.

Rissland, E. L. (1983). Examples in Legal Reasoning: Legal Hypotheticals. In Proceedings of *Eight International Joint Conference on Artificial Intelligence*, 90–93. Karlsruhe, West Germany: Morgan Kaufmann Publishers.

Rissland, E. L., Valcarce, E. M. & Ashley, K. D. (1984). Explaining and Arguing with Examples. In Proceedings of *Fourth National Conference on Artificial Intelligence*, 288–294. Austin, TX: AAAI Press.

Rosch, E. (1978). Principles of Categorization. In Eleanor Rosch & Lloyd, B. B. (eds.) *Cognition and Categorization*. Lawrence Erlbaum: Hillsdale, NJ.

Sacerdoti, E. (1974). Planning in a Hierarchy of Abstraction Spaces. *Artificial Intelligence* **5**: 115–135.

Schank, R. C. (1975). *Conceptual Information Processing*. North Holland: Amsterdam.

Schank, R. C. & Abelson, R. P. (1977). *Scripts, Plans, Goals, and Understanding*. Lawrence Erlbaum: Hillsdale, NJ.

Schmolze, J. G. & Lipkis, T. A. (1983). Classification in the KL-ONE Knowledge Representation System. In Proceedings of *Eighth International Joint Conference on Artificial Intelligence*, 330-332. Karlsruhe, West Germany: Morgan Kaufman Publishers.

Schubert, L. K. (1976). Extending the Expressive Power of Semantic Networks. *Artificial Intelligence* **7**: 163–195.

Searle, J. R. (1969). *Speech Acts; An Essay in the Philosophy of Language*. Cambridge University Press: Cambridge, England.

Searle, J. R. (1979). Literal Meaning. In Searle, J. R. (ed.) *Expression and Meaning*. Cambridge University Press: Cambridge, England.

Shapiro, S. (1979). The SNePS Semantic Network Processing System. In Findler, N. (ed.) *Associative Networks*. Academic Press: New York.

Sidner, C. L. (1985). Plan Parsing for Intended Response Recognition in Discourse. *Computational Intelligence* **1**: 1–10.

Teitelman, W. et al. (1978). *The Interlisp Reference Manual*. Xerox PARC.

van Melle, W. (1980). *A Domain Independent System That Aids in Constructing Knowledge-Based Consultation Programs*. HPP-80-22, Heuristic Programming Project, Computer Science Department: Stanford University, Stanford, CA.

Webber, B. L. & Mays, E. (1983). Varieties of User Misconceptions: Detection and Correction. In Proceedings of *Eighth International Joint Conference on Artificial Intelligence*, 650–652. Karlsruhe, West Germany: Morgan Kaufman Publishers.

Wilensky, R. (1983). *Planning and Understanding: A Computational Approach to Human Reasoning*. Addison-Wesley: Reading, MA.

Wilensky, R. (1986). *Some Problems and Proposals for Knowledge Representation*. UCB/CSD 86/294, Computer Science Division: University of California, Berkeley, CA.

Wilensky, R. (1987). *Primal Content and Actual Content: An Antidote to Literal Meaning*. UCB/CSD 87/365, Computer Science Division: University of California, Berkeley, CA.

Wilensky, R. & Arens, Y. (1980). *A Knowledge-Based Approach to Natural Language Processing*. UCB/ERL M80/34, Electronic Research Laboratory: University of California, Berkeley, CA.

Wilensky, R., Arens, Y. & Chin, D. N. (1984). Talking to Unix in English: An Overview of UC. *Communications of the Association for Computing Machinery* **27**(6): 574–593.