

The Berkeley UNIX Consultant Project*†

Robert Wilensky
David N. Chin
Marc Luria
James Martin
James Mayfield
Dekai Wu**

Division of Computer Science
Department of EECS
University of California, Berkeley
Berkeley, CA. 94720

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 KNAME 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.

* 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 under contract N00039-84-C-0089, by the Office of Naval Research under contract N00014-80-C-0732, and by the National Science Foundation under grant MCS79-06543.

† This report is a revised version of report no. UCB/CSD 87/303, and has appeared in *Computational Linguistics* 14-4, December 1988.

** The current address of James Mayfield is Department of Computer Science, University of Maryland Baltimore County, 5401 Wilkens Ave., Baltimore MD 21228; of David Chin, Dept. of Information and Computer Science, Univ. of Hawaii at Manoa, 2565 The Mall, Honolulu HI 96822; of Marc Luria is Department of Computer Science, Technion, Haifa, Israel; and of James Martin, Dept of Computer Science, University of Colorado, Boulder, CO, 80309-0430

†† UNIX is trademark of Bell Laboratories

Acknowledgments

In addition to the authors of this paper, several other people have made significant contributions to the UC project and to this report. Richard Alterman played a major role in the overall design, organization, and execution of this project. He also made many specific contributions to its individual components, especially **UCGen** and **UCEgo**, and was involved in many of the technical discussions upon which this work was based. Peter Norvig wrote the original **KODIAK** interpreter and contributed significantly to the development of this representation language. Eric Karlson built the graphic interface program used to enter and edit **KODIAK** diagrams. James Mayfield took charge of coordinating and integrating the components of this report into a single document. Charles Cox and Anthony Albert contributed to previous versions of this document. Other members of our group who contributed to this effort include Yigal Arens, Michael Braverman, Margaret Butler, Paul Jacobs, Dan Jurafsky, Lisa Rau, and Nigel Ward.

Table of Contents

| | |
|---|----|
| 1. Introduction to the UNIX Consultant (UC) Project | 1 |
| 2. KODIAK | 15 |
| 3. The Analyzer | 23 |
| 4. The Concretion Mechanism | 25 |
| 5. The Goal Analyzer | 34 |
| 6. The Ego Mechanism | 41 |
| 7. The Planner | 54 |
| 8. The Expression Mechanism | 63 |
| 9. The Generator | 69 |
| 10. Knowledge Acquisition | 74 |
| 11. Problems | 79 |
| 12. References | 82 |



1. Introduction to the UNIX Consultant (UC) Project

Several years 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[†] 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.

[†]UNIX is trademark of Bell Laboratories

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 takes 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. UC Old and New

We initially built a prototype version of UC consisting largely of "off the shelf" components [Wilensky, Arens, & Chin 1984]. While this system seemed to suggest that our goal was feasible, it was deficient in many ways. There were whole components that needed to be included but were not. For example, the initial system made few inferences and was not capable of planning its own actions. In addition, each individual component was in need of much refinement.

Probably the most important deficiency was in the area of knowledge representation. The initial prototype of UC was implemented in PEARL [Deering, Faletti, & Wilensky 1981]. PEARL is an AI language and database-management package that supports

frame-like structures similar to those employed by other representation languages, with perhaps some more attention given to efficient retrieval. However, we found that our underlying representational system was inadequate. Unfortunately, the problems with our system were not unique to it, but were shared by most other efforts to represent and organize knowledge.

Much of the focus of our recent work has been to address and rectify these problems of knowledge representation. Our critiques of existing knowledge representation schemes, along with our new prescription for these deficiencies, can be found in Wilensky 1986. That report contains a description of **KODIAK**, the knowledge representation system that our work has led us to, and upon which our current implementation of the UNIX Consultant is based.

Since one's knowledge representation is generally fundamental to the structure of most of the modules of one's systems, developing a new one means redesigning each component around a new representational system. This report is a description of a new prototype of UC so designed.

1.3. 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 tasks, 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.

1.4. 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.

1.4.1. 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.

(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.

(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.

(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.

(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.

(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.

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.

1.5. 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.

1.5.1. Simple Examples

```
% UC
Welcome to UC (Unix Consultant) version 3.23
To a UC '#' prompt, please type in your questions
about the Unix le 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 le?
Use mv.
For example, to move the le named foo to the le named fool,
type 'mv foo fool'.
```

Do you know how to delete a file named deka?
Type 'rm deka'.

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 'winger'.

How can I find out wilensky's phone number?

Type 'winger 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 fool,
type 'mv foo fool'.

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 fool,
type 'cp foo fool'.

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 le.

Use more.

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

I want to edit a le.

Use vi.

For example, to edit the le named foo, type 'vi foo'.

I want to nd out Joe's address.

Type 'nger joe'.

I want to remove a directory named junk.

Type 'rmdir junk'.

What's the command to compact a le?

Use compact.

For example, to compact the le named foo, type 'compact foo'.

What is a directory?

A directory is a le that is used to contain les.

What is a le?

A directory is a container that is used to contain text,
code or les.

What is a rm?

Rm is a command that is used to delete les.

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.

1.5.2. More Complex Examples

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

Hi.
How can I help you?

# Is rn used to rename les?
No, rn is used to read news.
To rename a le to another le, use mv.
For example, to rename the le named foo to the le named fool,
type 'mv foo fool'.
```

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.

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

Hi.
How can I help you?

# How can I delete a le?
Use rm.
For example, to delete the le 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.

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

Hi.
How can I help you?

# How can I nd out the inode of a le?
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 8.

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

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

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 section 7.2.3 below.

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

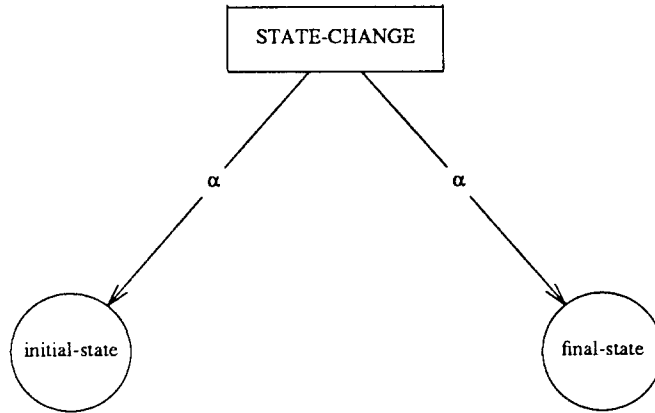
The knowledge used by UC is represented in **KODIAK**, as are the various stages of the processing of an utterance. **KODIAK** (Keystone to Overall Design for Integration and Application of Knowledge) is an implementation of CRT (Cognitive Representation Theory), an approach to knowledge representation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

KODIAK is a relation-oriented rather than object-oriented system. In an object-oriented system, one thinks of objects as having slots, and the representational weight is carried by such structured objects. In a relation-oriented system, objects are thought of as atomic elements, and knowledge is captured by the relations in which these objects participate.

For example, suppose we want to represent a concept like "state change", i.e., an event in which some state is thought of as changing into another state. Among other things, we would like to capture the fact that any state change has an initial state and a final state. In **KODIAK**, we would represent this by first creating objects corresponding to the concepts state change and state. Let us call them **STATE-CHANGE** and **STATE**, respectively. Then we would indicate that a relation, say, *initial-state*, holds between each instance of a state change and some instance of a state. Similarly, another relation, say, *final-state*, holds between each state change and some state.

This descriptions sounds equivalent to saying that *initial-state* and *final-state* are slots of **STATE-CHANGE**. For the purposes of this paper, we will not strenuously resist this interpretation. However, the underlying philosophy (and actual representation) is different. In particular, rather than accounting for the relatedness of **STATE-CHANGE** and *initial-state* and *final-state* by creating a primitive notion of slot and asserting that the latter two items are slots in the former, this relatedness is simply attributed to the fact that every **STATE-CHANGE** participates in an *initial-state* relation and a *final-state* relation. On the other hand, we think of these relations as being less closely related to **STATE** because not every **STATE** participates in these relations.

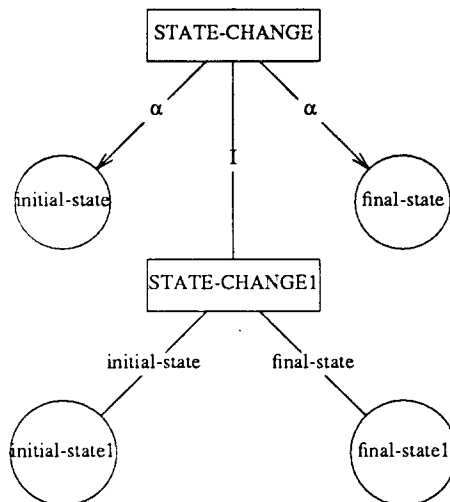
In our terminology, we refer to objects like **STATE** and **STATE-CHANGE** as absolutes, and the relations in which they participate, such as *initial-state* and *final-state*, as aspectuals. The motivation for having a more fine grain representation than objects with slots is described in Wilensky 1986. However, here we shall use a more conventional notation for the sake of exposition. For example, we will write the above facts about state changes using the following notation:



This diagram should be interpreted as follows: The individual instances of the concept STATE-CHANGE always participate in two relations, one called initial-state and one called final-state. That is, if we ever encountered an individual of the concept STATE-CHANGE, it will be in an initial-state and a final-state relation to something. The circles themselves represent the idea of being in a given relation to something. Thus, they correspond most closely to the roles of other systems.

Of course, there is much more to say about state changes. For example, we have not yet expressed the fact that the other argument to both initial-state and final-state must always be a state. Also, the state that is in a initial-state relation to a state change always occurs before the state that is in the final-state relation. STATE-CHANGE itself is a type of event. We will now describe how such facts are represented.

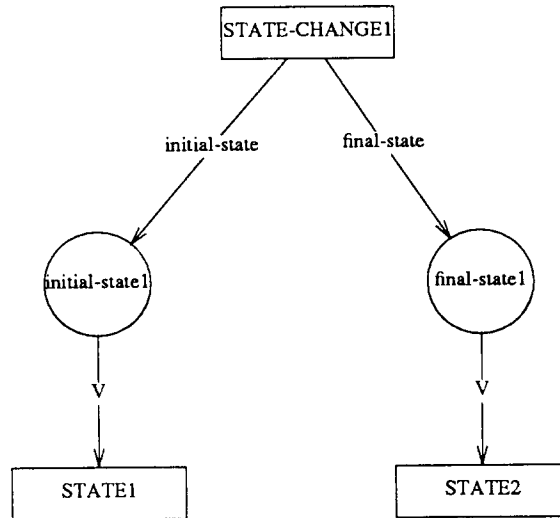
We represent individuals of a concept, and subtypes of a concept, using KL-ONE-like structured inheritance technique. For example, to represent an instance of a state change, we would create the following structure:



The **Instantiate (I)** link states that STATE-CHANGE1 is an individual STATE-CHANGE. initial-state1 should be interpreted as the assertion that the initial-state relation holds between STATE-CHANGE1 and some (as yet unspecified) value; the

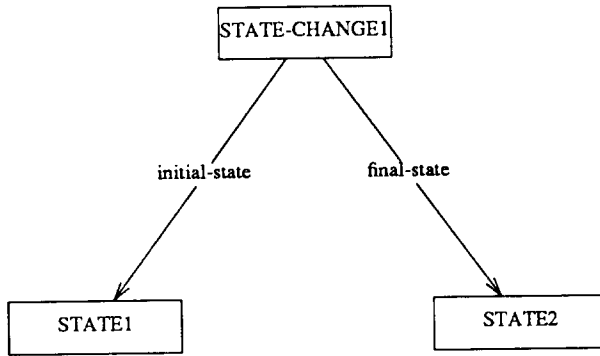
surrounding circle represents the idea of being in the initial-state relation to STATE-CHANGE1. We sometimes say that initial-state1 *plays the role* of the initial-state relation with respect to STATE-CHANGE1.

Note that role-play is a relation between a relation and its use, not a relation between a role and its filler. Indeed, the values to which initial-state and final-state connect STATE-CHANGE1 have not yet been depicted. To include these values in our representation, we use the **Value (V)** link. Thus, we would typically find instances like the above elaborated to include the following:

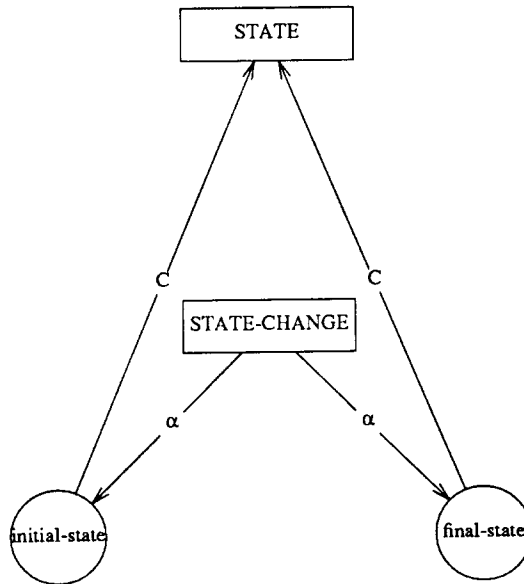


That is, the initial-state relation holds between STATE-CHANGE1 and something called STATE1, and the final-state relation holds between STATE-CHANGE1 and something called STATE2. In other terminology, the object at the end of the **Value** link represents the filler, and the object at the source, the slot. Presumably, we would also include what other information we knew about STATE1 and STATE2, namely, what categories they are instances of, etc.

We can abbreviate this notation if we have no need for the showing the slots of an individual, but merely their fillers. We do this by drawing a labeled link directly to the target of the **Value** link. For example, we could simplify the above two diagrams by drawing a link labeled initial-state between STATE-CHANGE1 and STATE1, and one labeled final-state between STATE-CHANGE1 and STATE2. Thus, it is sometimes convenient to draw the following:

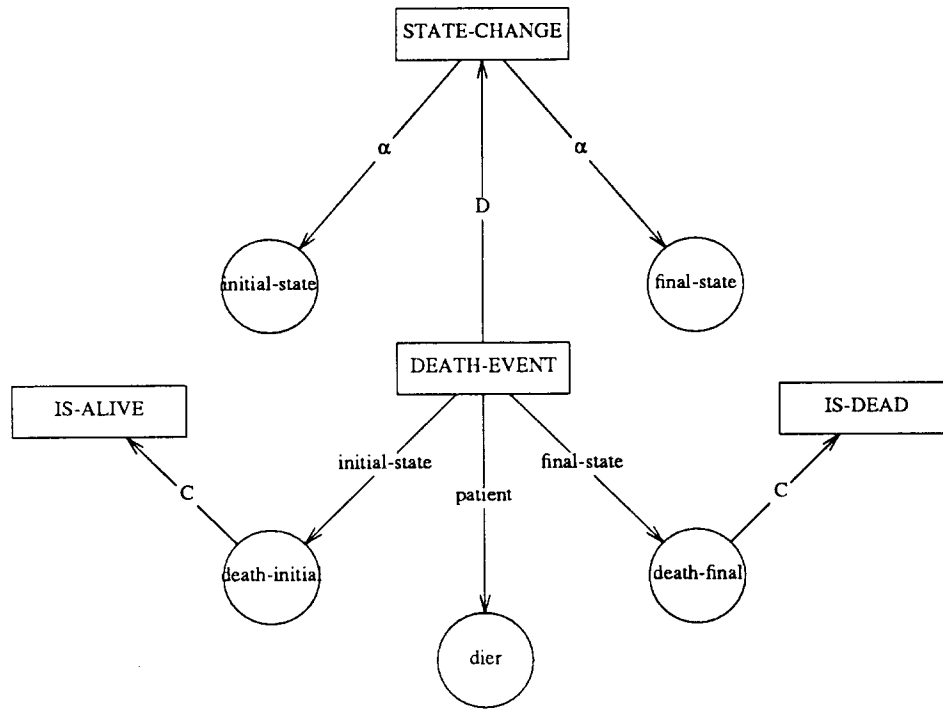


Usually, the aspectuals of a concept are constrained so that their other argument must be of a certain type. For example, to indicate that the other argument to initial-state and final-state must be an instance of a STATE, we would draw a **Constrain (C)** link from the circles labeled initial-state and final-state to a box labeled STATE, as in the following diagram:



These links would be interpreted as saying that any object in a **Value** relation with a use of initial-state or final-state must be of type STATE.

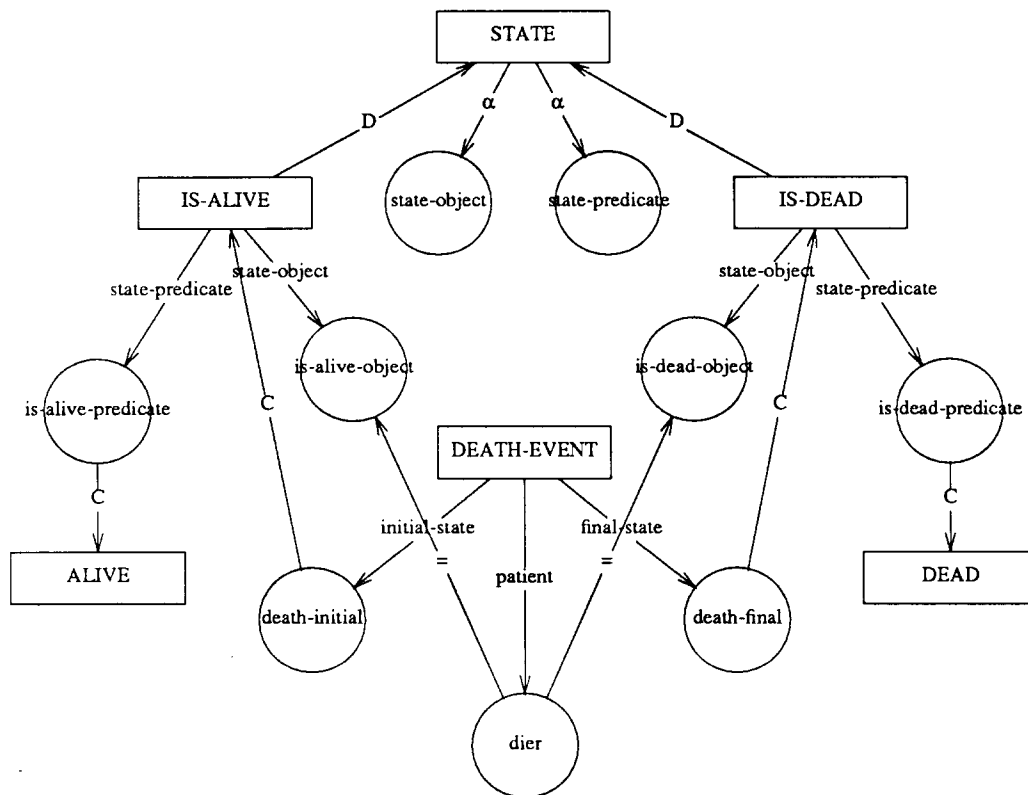
Subtypes are represented using a **Dominate (D)** link instead of an **Instantiate (I)** link. The aspectuals are created for the subtype than play the role of the aspectuals of the parent. These are typically differentiated from those of the parent concept by being subject to a further constraint. For example, we can begin to define dying as a kind of state change as follows:



We have represented DEATH-EVENT as being a kind of STATE-CHANGE in which the initial-state is constrained to be a state of something being alive, and the final-state to be that of something being dead.

In this diagram, we have not bothered to show the constraints on the aspectuals of STATE-CHANGE. In general, we will only show those parts of previous concepts that are pertinent to the knowledge being represented. Also, we introduced a new aspectual, dier, which specializes patient. Patient is inherited from some quite abstract absolute which ultimately dominates STATE-CHANGE, and which has the meaning "event that affects an object". Also, since patient is an aspectual of some concept dominating STATE-CHANGE, we should have specialized it at STATE-CHANGE, say, to state-change-object. However, when such specialized versions of relations are not substantially differentiated from their ancestors, we will use the ancestor's name instead. The KODIAK interpreter understands the occurrence of an inherited relation as being the most specific inheritable specialization of that relation, so using the more abstract name has the same semantics of using the more specialized one.

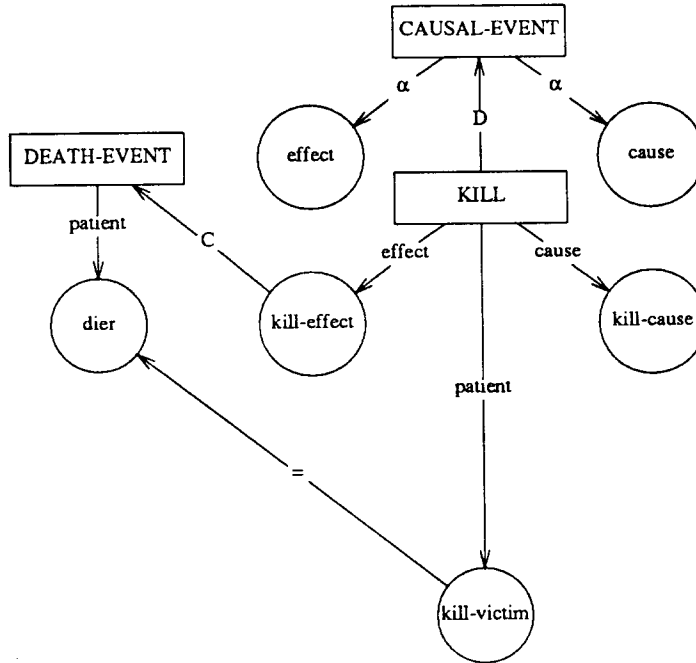
One item missing from this diagram is the fact that the fellow who dies is the same fellow as the one who was alive and then was dead. This fact can be captured with the aid of the **Equate** (=) link. This is similar to KL-ONE's role-chains. For example, we can improve our representation for dying by creating a state of being alive and a state of being dead, and then using **Equate** links to state the relationships between their components:



In this diagram, we state that the dier of DEATH-EVENT is the same as the is-alive-object of some IS-ALIVE state, and also the same as the is-dead-object of some IS-DEAD state.

The **Equate** links belong to a particular absolute. For example, in the diagram above, the **Equate** links emanating from dier belong to DEATH-EVENT. The significance of this fact is that it is true of dying that there must be an associated event of being alive, but it is not true of being alive that there must be a death event. We represent this aspect of **Equate**s by the direction of the arrows in the diagram.

Having defined DEATH-EVENT, it is easy to represent a concept like killing, assuming an analysis in which killing is a kind of causing in which the thing caused is a kind of death event:



Here we define KILL as being a subtype of CAUSAL-EVENT. CAUSAL-EVENT is a concept shown as participating in an effect and a cause relation. KILL is also shown as participating in a kill-effect and a kill-cause relation. Because effect and cause are written on the links to these relations, we know that kill-effect is just a specialized version of the effect relation, and that kill-cause is a specialized version of cause. To differentiate KILL from CAUSAL-EVENT, we need to capture the fact that the effect of a KILL is a death event. The additional semantics of kill-effect (and thereby, of KILL) come from the **Constrain** link on kill-effect. This says that the effect of a KILL is a DEATH-EVENT. Finally, we use an **Equate** link to state that the person killed is the same one as the one who died.

Of course, many parts of the meaning of these simple concepts have been omitted from these diagrams. For example, we have not stated any information about time, such as the fact that the initial-state precedes the final-state. In general, the diagrams presented in this paper have been simplified to present the crucial knowledge needed without being even more complex than they already are.

KODIAK has additional features that are not described herein. The reader is referred to Wilensky 1986 for a more complete description. A summary of all the KODIAK features used in this report is found in the legend in Figure 1. Note that the legend contains one additional abbreviation, namely, that drawing a link bearing the name of a relation between two categories is interpreted as stating that the individuals of the domain category participate in that relation with some member of the range category.

NODES

REPRESENTATION

MEANING

CONCEPT

an absolute

○

a relation

CONCEPT_n, n an integer

CONCEPT_n is an instance of CONCEPT

LINKS

The following is priority ordered (i.e. use first match for meaning):

CATEGORY - α -> ○

○ is an aspectual, i.e., a relation in which CATEGORY participates

CATEGORY1 -D-> CATEGORY2

CATEGORY2 dominates CATEGORY1.

INSTANCE -I-> CATEGORY

INSTANCE is an instance of CATEGORY.

○-C-> CATEGORY

The argument to ○ is constrained to be of type CATEGORY.

○-V-> OBJECT

The value of the argument to ○ is object.

○-==> ○

The first aspectual is constrained to have the same value as the second.

CATEGORY -rel-> ○

○ is an aspectual of CATEGORY, and ○ specializes *rel*, an aspectual of some concept dominating CATEGORY.

CATEGORY1 -rel-> CATEGORY2

Each member of CATEGORY1 participates in *rel* with some member of CATEGORY2.

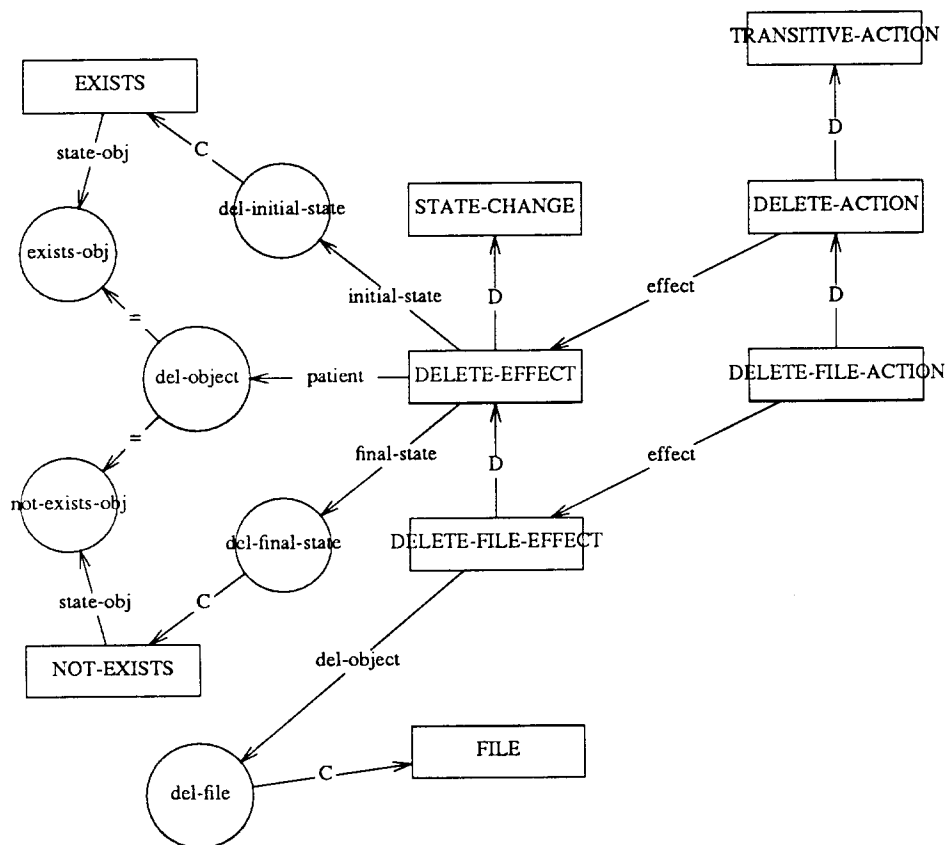
Figure 1
KODIAK Legend

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 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. The following diagram shows how knowledge about deletion is represented in UC:



The central node in this diagram is DELETED-EFFECT. DELETED-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.)

DELETED-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 DELETED-EFFECT to be the effect of DELETED-ACTION.

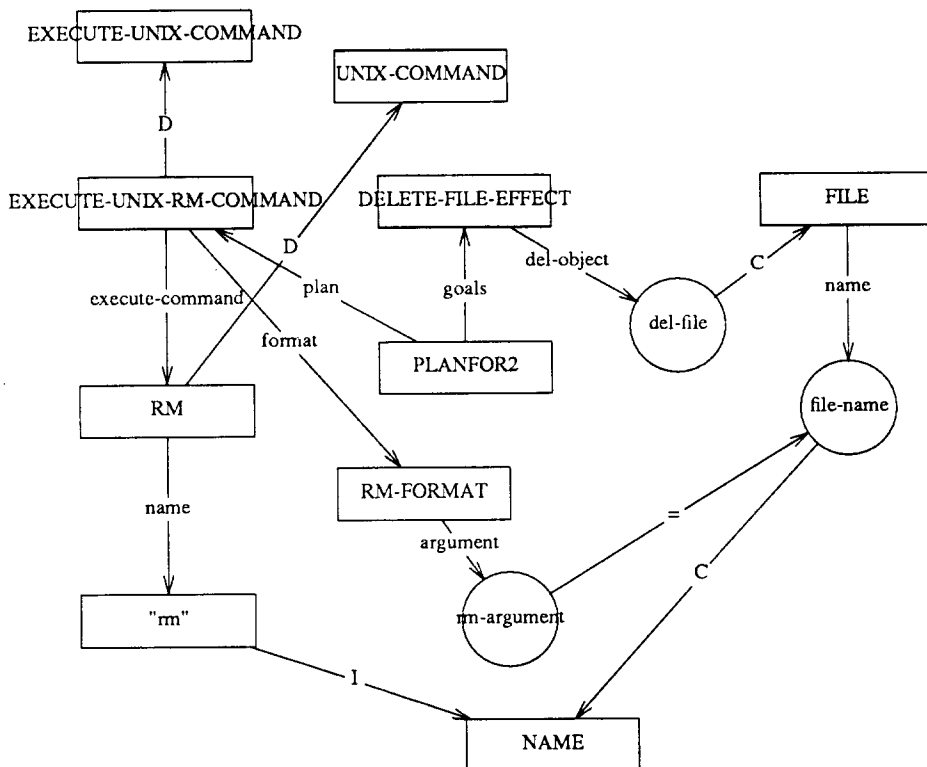
Using names like DELETED-EFFECT may be somewhat misleading. In particular, DELETED-EFFECT is not required to be the effect of anything - while DELETED-ACTION is defined as having DELETED-EFFECT as its result, this statement imposes a requirement on DELETED-ACTION, not on DELETED-EFFECT. We call such 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.

The next figure 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'.

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?"* 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 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.

*A kind of laser printer used at our site.

3. 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.

ALANA is a descendent of **PHRAN** [Wilensky & Arens 1980], the front-end natural-language component for the original UC [Wilensky, Arens & Chin 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 2 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.

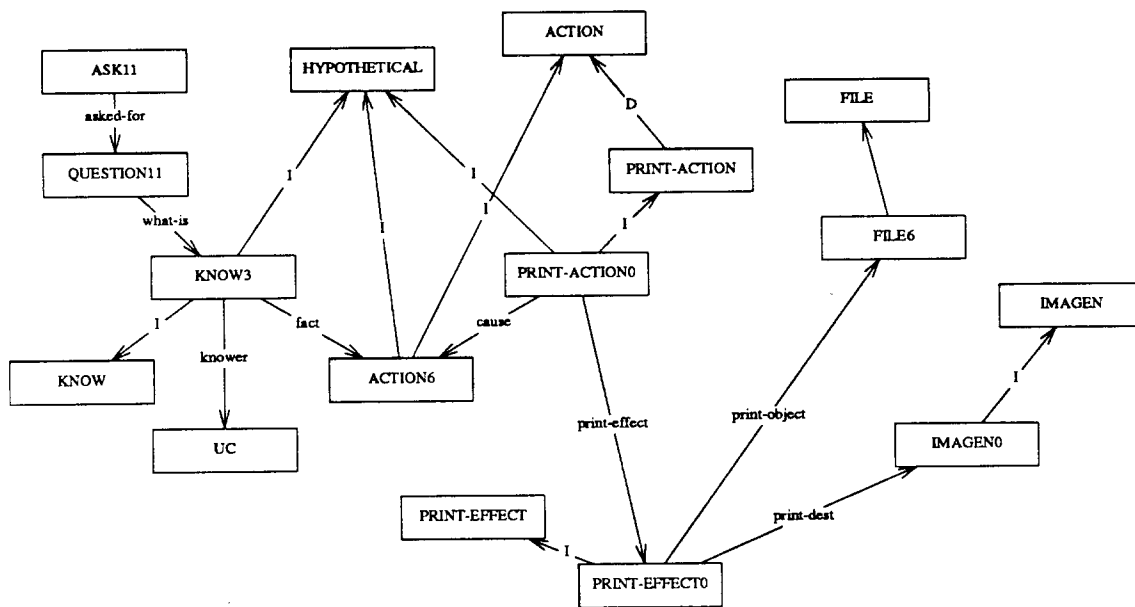


Figure 2
ALANA's output for "Do you know how to print a file on the Imagen?"

4. The Concretion Mechanism

4.1. Introduction

A *concretion inference* is a kind of inference in which a more specific interpretation of an utterance is made than can be sustained on a strictly logical basis [Norvig 1983], [Wilensky 1983]. Examination of contextual clues provides the means to determine which of many possible interpretations are likely candidates. An example of a simple type of concretion inference occurs in understanding that "to use a pencil" means to write with a pencil, whereas "to use a dictionary" means to look up a word.

Concretion differs classification such as that of KL-ONE [Schmolze & Lipkis 1983, Brachman & Schmolze 1985] in that a concretion inference may be incorrect. Ordinarily, "to use a pencil" implies writing; however, in a particular context, it may refer to propping a door open with a pencil. Nevertheless, in the absence of compelling evidence to the contrary, the natural interpretation is writing. Thus, concretion may be thought of as defeasible classification.

A process that performs concretion is called a *concretion mechanism*. A concretion mechanism attempts to find clues in a set of general concepts to generate concepts that are more specific. *Writing*, for instance, is a specific type of *using*, in which the tool being used may be a pencil. The use of such a mechanism permits a straightforward approach to manipulating hierarchical knowledge structures. The initial interpretation of an utterance may include concepts too general for the utterance to be considered understood. Such general concepts embody the common features of their descendent concepts, but for some reason insufficiently specify the meaning of the utterance. Thus, the concretion mechanism is responsible for making an appropriate interpretation of a concept by selecting one of its sub-concepts, found lower in the hierarchy.

In deciding when concretion operations should be performed, it is critical to consider how specific a concept's representation must be to be "understood". Different levels of categorization are considered adequate from situation to situation. For instance, it is perfectly acceptable in most circumstances to leave the interpretation of "eating" as "eating some food". However, in a context involving picnics, a more specific interpretation is likely to be made, namely, that the eating involved paper plates, that people sat on the ground or at a picnic table, that the food was of a certain type, and so forth. In general, in cases where a more specific category than "usual" is requisite, often some feature of the prototype of the supercategory is violated, resulting in a higher probability of selecting a subcategory where this feature is accommodated.

It is important that the mechanism be able to recognize from a wide variety of clues when there is sufficient evidence to concrete, as well as when an ambiguity needs to be resolved. A uniform method of representing the rules by which a concretion may be made is required. Naturally, wrong inferences can occasionally be made, and some means must therefore be provided to find and correct them when contradictory facts are learned.

4.2. Concretion in UC

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.

4.3. Example

Consider the example, "Do you know how to print a file on the imagen?". The subpart "print a file on the imagen" is parsed into the representation shown in Figure 3. 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, whose 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.

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

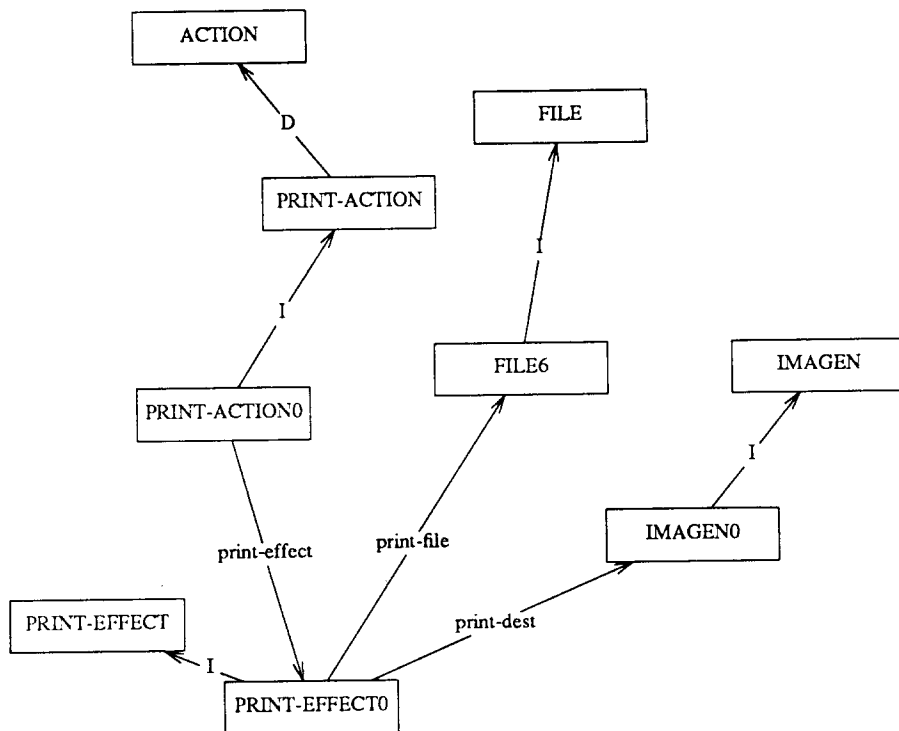


Figure 3
Representation of "print a file on the imagen"

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.

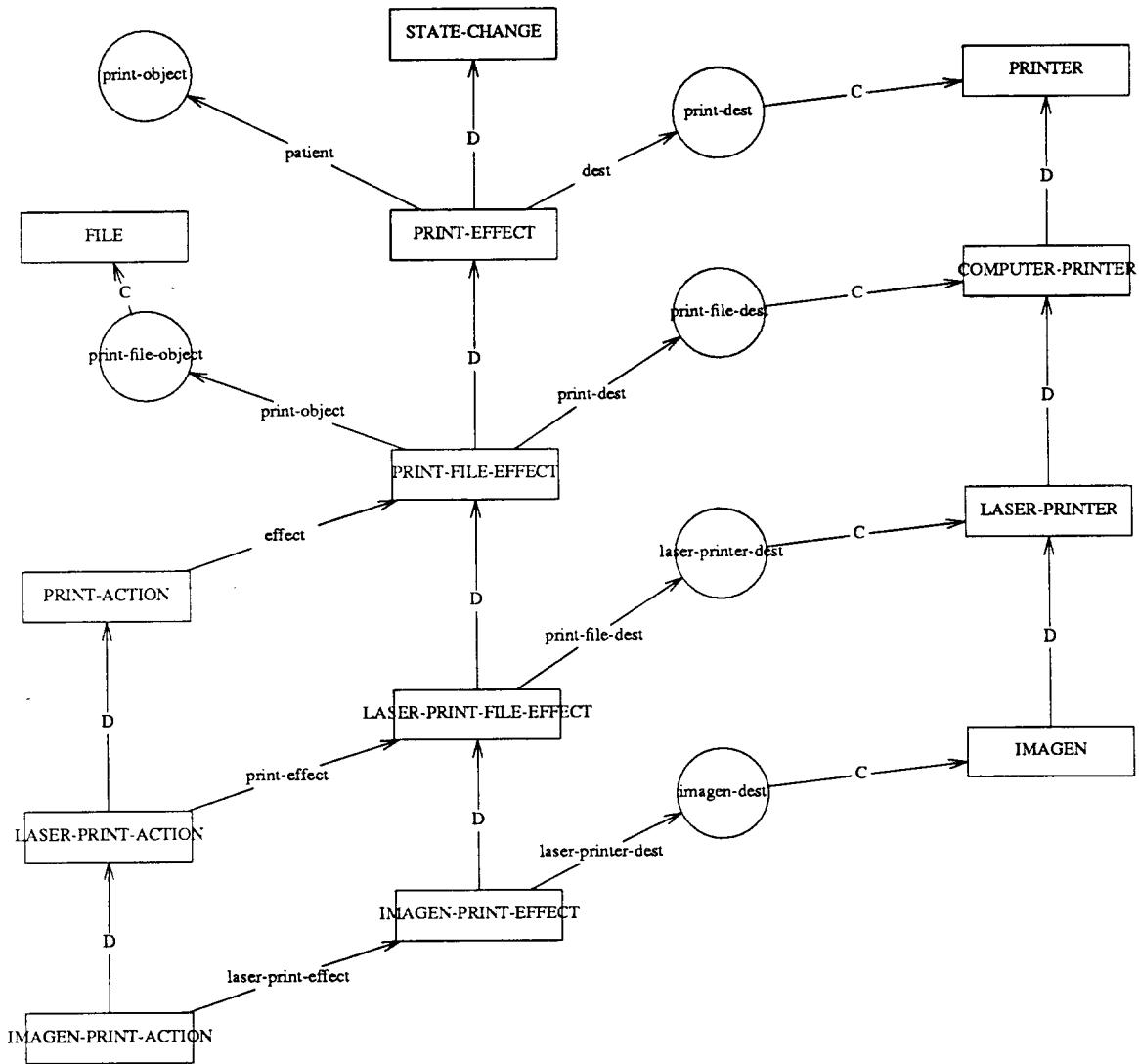


Figure 4
Some knowledge about printing.

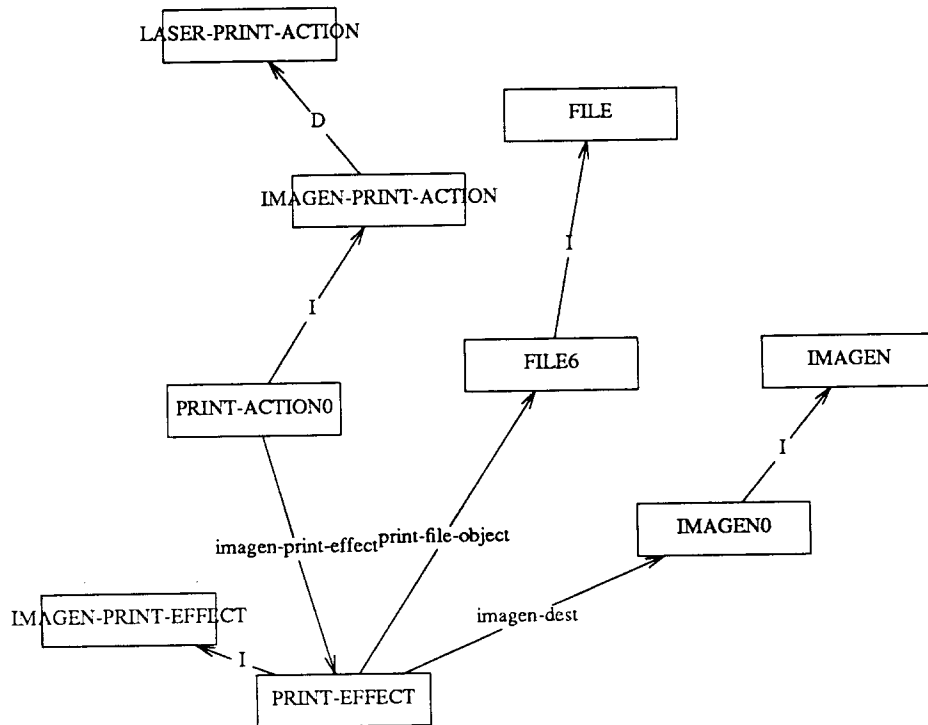


Figure 5
Result of concreting "print file on the imagen"

5. 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

(1) 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:

Q1 *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, (1) 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:

Q2 *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:

(2) Do you have write permission on the parent directory?

(3) Yes.

The ability to understand the full meaning of (3) is contingent on the realization that it relates directly and conventionally to (2). Thus, PAGAN will require knowledge of such sequences to correctly determine the goal underlying utterances such as (3).

5.1. Knowledge Representation for PAGAN

A *planfor* is a relation between a goal and a sequence of steps (called a *plan*) that constitutes a possible method of achieving that goal. All PAGAN's knowledge of conversation is stored as planfors.

Planfors provide a means to address the questions posed above. First, indirect speech acts can be expressed as planfors. For example, the generic indirect speech act underlying (3) can be expressed as:

PLANFOR1

GOAL: Speaker ask hearer how to perform task
PLAN: Speaker ask hearer whether hearer knows how to perform task

Second, planfors provide a means to express conventionalized relationships between utterances. Utterance (1) and its answer can be represented as:

PLANFOR2

GOAL: Speaker know how to perform task
PLAN: Speaker ask hearer how to perform task
Hearer tell speaker how to perform task

(In general, steps of a plan that are to be executed by someone other than the planner are interpreted as the planner waiting for that event to happen.)

Representing both speech-act knowledge and conversational knowledge with planfors has two advantages. First, it allows a single mechanism to handle the processing of both phenomena. The goal-analysis mechanism described below does just this. Second, it allows the two forms of knowledge to be combined into a single structure. For example, the two preceding planfors can be combined to express both the indirect speech act and the question-and-answer sequence:

PLANFOR3

GOAL: Speaker know how to perform task
PLAN: Speaker ask hearer whether hearer knows how to perform task
Hearer tell speaker how to perform task

The KODIAK representation of a planfor is shown in Figure 6. Figure 7 depicts PLANFOR3 in its KODIAK form. (It is called PLANFOR34 in the knowledge base). It is a PLANFOR whose goal is to know a fact which is itself a plan for some goal. (The names of the aspectuals not used to make a predication have been omitted from the diagram.) The plan of PLANFOR34 is for the person who wants to know this action to ask someone whether that person knows the action sought, and then, to have that same person tell the asker what was desired.

Note that planfors do not represent fundamental knowledge of causality. There is usually a causal relationship between a plan and a goal that are connected by a planfor. However, the planfor itself does not represent the causality. What a planfor does represent is a notion of typicality. It indicates that its plan is one that is typically or conventionally used to achieve its goal. For example, the unix 'rm' command may cause a file to be deleted. It may also cause the disk arm to be moved. It would be a mistake though to say that 'rm' should be connected to the goal of moving the disk arm by a planfor relation; 'rm' is not typically used to move the disk arm. On the other hand, 'rm' should be connected to the goal of deleting a file by a planfor relation, since this goal is what 'rm' is typically used for.

Traditional approaches to dialogue understanding have focused on the process of plan inference. Under this approach, utterances are viewed as steps of plans. Such plans

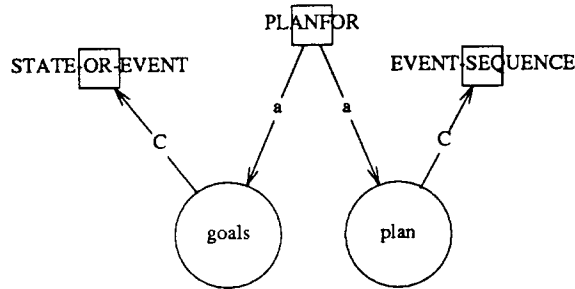


Figure 6
Definition of a PLANFOR

may themselves be parts of higher-level plans, and so on. Allen and Perrault [1980] developed a system that exemplifies this approach. Their system handled direct and indirect speech acts by plan analysis. Carberry [1983] extended this paradigm to deal more thoroughly with domain plans. Litman and Allen [1984] used the notion of meta-plans [Wilensky 1983] to facilitate the comprehension of subdialogues. Grosz and Sidner [1985] pointed out the need for attentional knowledge in understanding discourse. One problem that has persisted in the literature is an inadequate representation of the relationship between goals and plans. Planfors provide such a representation.

Planfors allow a goal-analysis mechanism to combine certain inferences that should be kept together. First, inferences about plans may be made at the same time as those about goals. This is in contrast with systems such as Wilensky's PAM system [1983] that use separate representations for inferring plans and goals. Second, inferences about plan recognition and inferences about intended-response recognition may be combined by including the intended response in the plan and associating this entire plan with a single goal. This is in contrast with systems such as Sidner's [1985] that first do plan recognition and then worry about what response was intended. The ability to do both kinds of inference simultaneously conforms to the intuition that no extra processing is required to determine, for example, that an answer is required once the realization is made that a question has been asked. Finally, planfors allow inferences about linguistic goals and about domain goals to be handled by a single inference engine. The separation of goal analysis into linguistic-goal reasoning and task-goal reasoning [cf. Allen, Frisch, & Litman 1982] is unnecessary, since the only difference between the two is the type of action that may comprise plan steps.

5.2. Goal Analysis

When knowledge of goals and plans is represented with planfors, goal analysis is the task of matching the representations produced by the language analyzer against the steps of plans stored in memory. The goal held by a speaker in making an utterance is then the goal that is associated with the matched plan via the planfor relation.

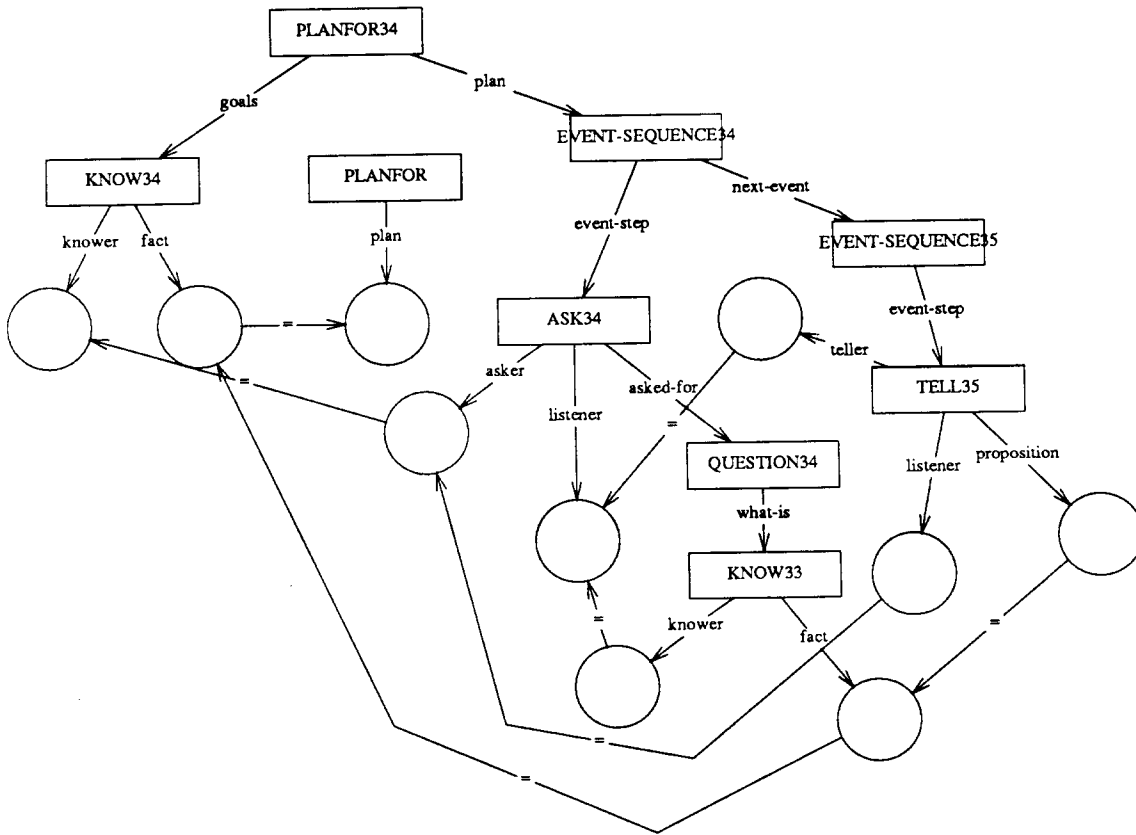


Figure 7
A plan for knowing is to ask if the hearer knows.

In the absence of any previous conversational context, an utterance to be analyzed is compared with the first plan step of each planfor that PAGAN knows about. When a match is found, the corresponding goal is taken to be the goal the speaker had in mind in making the utterance.

Several phenomena complicate this view of goal analysis. First, a speaker may intend a single utterance to be a part of more than one plan. For example, (1) is a plan for the goal of knowing how to print a file. Achieving this goal may in turn be part of a plan for actually printing a file. To handle such situations, PAGAN must apply the matching process recursively on each inferred goal. This matching process is repeated until no further higher-level goals can be inferred.

Second, preceding conversational events may set up expectations in relation to which an utterance is designed to be understood. For example, (3) cannot be readily interpreted when viewed in isolation. However, if it is used in response to a question such as (2), its interpretation is clear. Two additions must be made to the matching

algorithm to handle this and similar cases. First, before matching the utterance to plans in the planfor knowledge base, the utterance must be matched against the next step of any active planfor (i.e., any planfor already inferred but not yet completed). In this example, the representation of (3) would be matched against the second step of the question-and-answer plan started by (2) to determine if it is a response to the question. Second, when a match with a new planfor is found, PAPAN may also need to infer that the speaker has adopted the matched planfor. Suppose UC says (2) to the user. Here, UC is initiating a plan for the goal of knowing whether the user holds the indicated permission. But, at the moment, this goal is held only by UC; it is reasonable that the user might not address the goal at all. For example, the user might reply

(4) Does it matter?

instead of answering the question. If the user's response can be construed as an answer to the question, we say that the user has adopted the planfor, and we may then expect any additional steps in the plan to be pursued by the user.

Third, PAPAN may find more than one planfor in its long-term memory that matches the utterance. This is called *planfor ambiguity*. Planfor ambiguity is handled in one of two ways. If an alternative matches an expectation, as described in the previous paragraph, then that alternative is chosen as the correct interpretation. If no expectation is matched, PAPAN tries to reject an alternative as implausible. A planfor is implausible if PAPAN believes that the user believes that its goal already holds, if its goal does not appear to lead to some appreciable benefit for the user, or if PAPAN believes that the user believes that one of its plan steps could not be achieved.

5.3. Processing Overview

At PAPAN's core is a matching program that matches two KODIAK structures against one another. Two structures are said to match if they are isomorphic (i.e., they have the same link structure) and each pair of corresponding nodes matches. For two nodes to match, one must be equal to or an ancestor of the other. For example, Imagen would match Imagen or laser printer, and laser printer would match Imagen, but Imagen would not match Laserwriter.

PAPAN first tries to determine whether the utterance was expected. This is done by matching the representation of the utterance against those plan steps that have been inferred but not yet witnessed. Such expectations are stored in a separate structure to speed the matching process. Failing this, PAPAN attempts to match the representation of the utterance to the first steps of planfors stored in memory. If a single such match is found, this planfor is copied, forming a new planfor with the observed utterance as its first step. If more than one planfor is found to match, the resultant ambiguity is resolved either by matching its goal to an expected action or by consulting the user model to determine whether that goal and plan are plausible given what is known about the user.

5.4. Example

This section traces the processing performed by PAPAN to handle utterance (1). The input to PAPAN is the structure built by the analyzer from this utterance and refined by the concretion mechanism. A trace of PAPAN as it processes this structure is shown in Figure 8.

-
- {1} This utterance wasn't expected.
 - {2} This utterance wasn't an adopted plan.
 - {3} Matching ASK0 pattern against ASK11.
 - {4} Could not match KNOW23 pattern to ACTION6 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 8
Trace of PAGAN's processing of
"Do you know how to print a file on the imagen?"

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 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}. (The contribution of the user model is described in section 6.) 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-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 9.

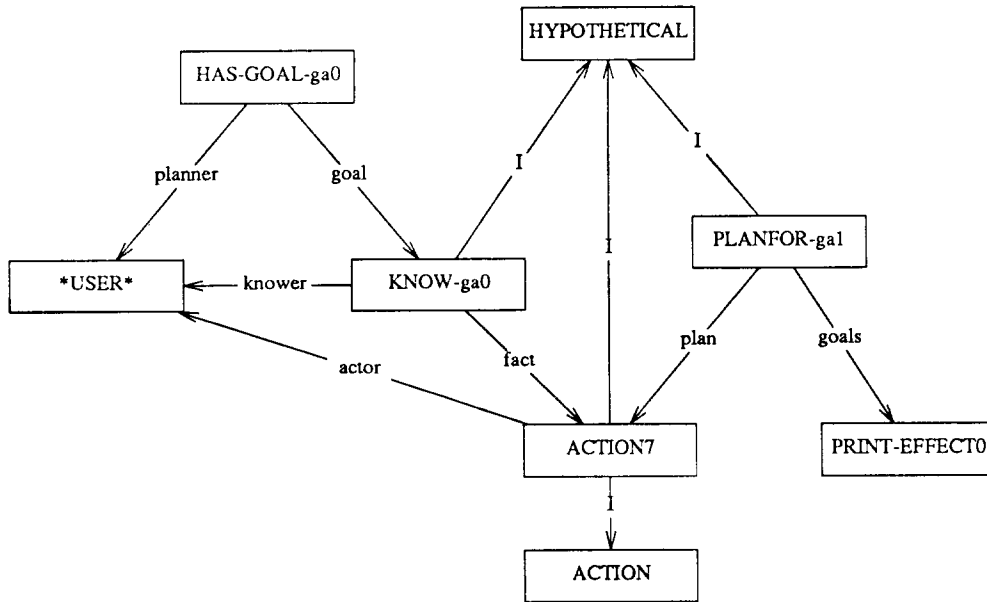


Figure 9
PAGAN output

In this figure, note that PAGAN has created a node labelled 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.

6. The Ego Mechanism

6.1. UCEgo

UCEgo is the component of UC that determines UC's own goals and attempts to achieve those goals. The input to UCEgo are the user's statements as interpreted by UC's conceptual analyzer and concretion mechanism, and the user's goals and plans as inferred by UC's goal analyzer. UCEgo draws on the UNIX planner component of UC to produce plans for doing things in UNIX. It passes the results to UC's expression mechanism, which prepares the conceptual information for generation into natural language.

The processing in UCEgo can be divided into two main phases: goal detection and plan selection. In goal detection [Wilensky 1983], UCEgo considers the current situation and detects appropriate goals for UC. The plan selection phase of UCEgo takes UC's goals and tries to produce a plan for satisfying them. The process of executing the plan normally results in a collection of concepts that are to be communicated to the user. UCEgo also includes an explicit user model, which encodes the user's knowledge state for use in goal detection and answer expression. Each of these subcomponents is described in greater detail below. A more substantial description of these subcomponents is described in Chin 1988.

6.1.1. Themes and goals

In UCEgo, goal detection is implemented by *if-detected demons*. If-detected demons are similar to the implication rules found in many semantic network type systems. If-detected demons contain two subparts, a detection net and an addition net. Both of these are networks of KODIAK concepts. Whenever the detection net of an if-detected demon matches what is in UC's memory, the addition net of the if-detected demon is copied into UC's memory. The detection and addition nets may share concepts, that is, share nodes in their networks. Here, the concepts that match the detection net are used in place of the corresponding concepts in the addition net. Thus, all the links in the addition net are preserved in making the copy, but some of the absolutes are from the result of the match. As described below, the matching process, which is somewhat different from that used by the goal analyzer, allows certain nodes to match instances of the concepts they specify. Therefore, these nodes function like the typed variables found in other systems.

When used in goal detection, the detection net of an if-detected demon represents the situation in which the goal should be detected, and the addition net of the if-detected demon contains the goal. Figure 10 shows an if-detected demon used in goal detection. This if-detected demon encodes the information that if UC has the goal (UC-HAS-GOAL3 in the diagram) of helping (HELP1) someone (PERSON4), and that person has the goal (HAS-GOAL0) of knowing something, then a plan for helping that person is for UC to satisfy (SATISFY1) the person's need to know.

Figure 10 shows an if-detected demon with an intersecting detection and addition net. In these diagrams, the detection net is designated by unlabeled arrows coming into the doubled circle labeled 'if-detected'. The net includes all those concepts plus all children of those concepts. The addition net is composed of those concepts pointed to by the if-detected double circle plus all their children. In the figure, the detection net consists of UC-HAS-GOAL3, HAS-GOAL0, and their child nodes. The addition net consists of

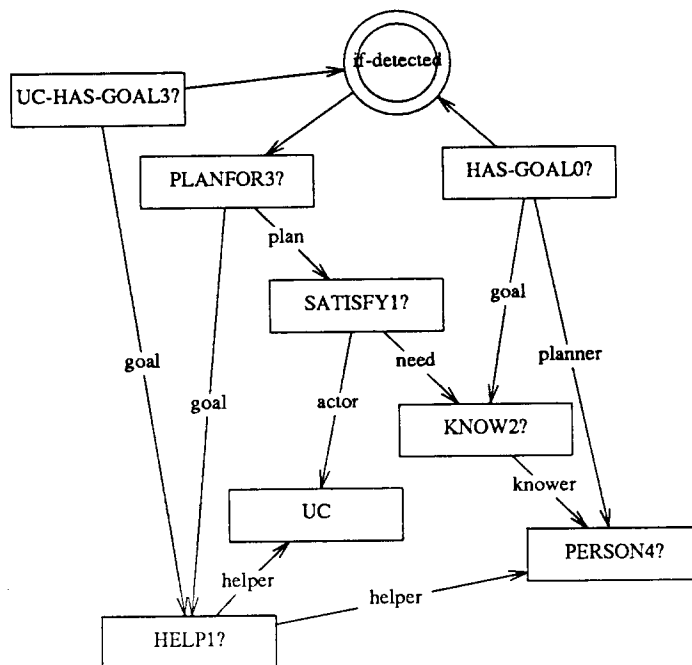


Figure 10
If-detected demon for adopting the user's goal of knowing.

PLANFOR3 plus all its child nodes. (UC-HAS-GOAL is a subtype of HAS-GOAL in which the planner is constrained to be UC, thus obviating the need to specify UC as the planner in each demon.) Thus, when PAGAN has inferred that the user wants to know something, and UC has the goal of helping the user (a recurrent goal that arises from UCEgo's computer-consultant-role theme), then UCEgo will detect the goal of satisfying the user's goal of knowing.

The use of demons in UCEgo is intended to represent its procedural knowledge of what to do in particular situations. For example, while a planfor structure used in both UCEgo and PAGAN might encode that some plan is appropriate for some goal, a demon is needed to cause UC to intend to use that plan in a situation in which that goal is present. Thus, demons represent the actions to be taken in a given situation, although both the situations and actions are described in declarative KODIAK format.

To capture generalizations effectively, the number of demons should be kept to the minimum and as much as possible should be represented as declarative KODIAK knowledge. For example, it is possible to have one demon for each situation in which a particular goal suggests adopting a particular plan; then we might be able to dispense with representing knowledge in planfors. However, doing so would not capture the generalization common to all these situations, namely, that expressed by the demon in Figure 10.

While we have attempted to keep the number of demons small, this version of UC certainly does not go as far as we would like in this direction. There are about 70 demons in the current version. We expect that a careful examination of them could result in reducing this number somewhat making the program more declaratively based.

UCEgo needs a complex control structure, because it has more varied tasks to perform than most of the other parts of UC. Indeed, to accommodate its needs, UCEgo uses a slightly different matching algorithm than some of the other components. In particular, the question marks in the diagrams are significant to the demon interpreter during both matching and copying. In matching, the question mark in a node means that the interpreter should look not just for exact matches, but also for any concepts that are members of the same categories as the node or specializations of those categories. For example, PERSON4? will be matched by any instances of either PERSON or specializations of PERSON such as USER. In copying the addition net, the interpretation of the question marks is to use the matched concept if the node is also a part of the detection net, or to create a new concept that is an instance of the same categories as the node. Nodes without question marks are used directly without copying.

These rules of interpretation extend only to using demons, and are purely a part of UCEgo's implement. That is, assertions made using these rules, when entered in the KODIAK knowledge base, have the same syntax and semantics as elsewhere in UC.

6.1.2. Extended goal detection

Besides situations where UCEgo simply adopts the user's goal of knowing, UCEgo also handles situations where it does not adopt the user's goal, such as when the user asks "How do I crash the system?" or "How can I delete UC?".

The cases where UCEgo does not tell the user the answer include examples of goal conflict where UCEgo's goal of wanting the user to know something conflicts with another of UCEgo's goals. For example, consider what happens when the user asks "How do I crash the system?". By normal processing, UCEgo arrives at the goal of wanting the user to know how to crash the system. However, crashing the system conflicts with UCEgo's recurrent goal of preserving the system, which arose from UCEgo's life theme of staying alive [Schank & Abelson 1977]. This theme is operationalized by yet another if-detected demon, which looks for the user wanting to crash the system, and results in UC having the goal of preventing the user from doing so. (This is one of only a few such states pertaining to the staying-alive theme that UCEgo has knowledge about.) Of course, it is the job PAGAN to determine that the user wants to crash the system from the user asking how to crash the system.

Figure 11 shows the if-detected demon that detects goal competition between UCEgo's goal of preserving something (SOMETHING1 in the diagram) and someone's goal (PERSON1 in the diagram) of altering it. In this example, UC-HAS-GOAL1 would be matched by UCEgo's recurrent goal of preserving the system, which arises from UCEgo's life theme of staying alive. HAS-GOAL2 would be matched by the user's goal of crashing (a specialization of altering) the system. As a result, UCEgo adopts the subgoal of preventing the user from crashing the system.

Next, the goal of preventing the user from crashing the system, with the information (inferred by UC's KNOME, cf. section 6.2) that the user does not know how to crash the

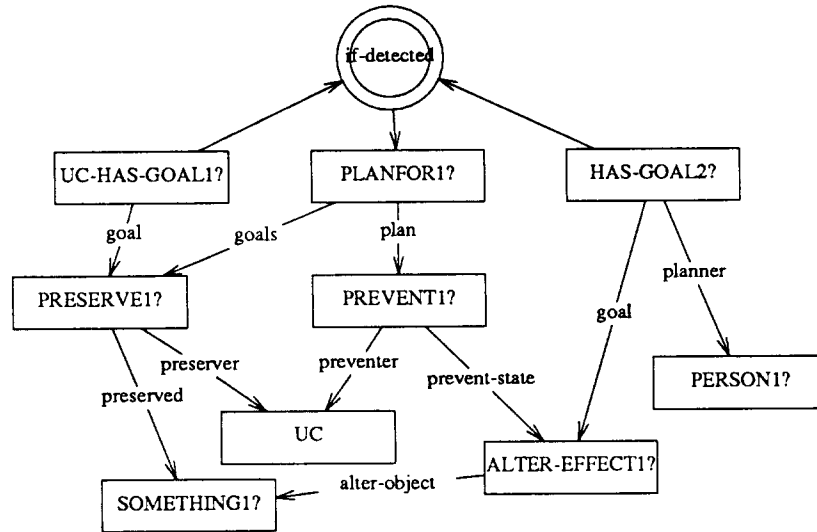


Figure 11
If-detected demon for detecting preserve/alter type goal conflicts.

system and the information that the user wants to know how (inferred by PAPAN), causes a new goal for UCEgo, namely, preventing the user from knowing how to crash the system. Figure 12 shows the if-detected demon responsible. This demon detects situations where UCEgo has a goal of preventing something from happening and where the person who desires this does not know how to do it and wants to know how. Here, UCEgo adopts the goal of preventing the person from knowing.

After detecting the subgoal of preventing the user from knowing how to crash the system, UCEgo will detect a goal conflict when it tries to adopt the usual goal of having the user know, in order to help the user. Figure 13 shows the if-detected demon that detects goal-conflict situations where UCEgo both has a particular goal and has the goal of preventing that goal. In such cases, UCEgo adopts the *meta-goal* [Wilensky 1983] of resolving the goal conflict. The general mechanism of dealing with such meta-goals is described in the next section. The eventual result of this mechanism here is that the goal of preventing the user from knowing is maintained and the other conflicting goal is abandoned.

6.1.3. Plan selection

After UCEgo has detected the appropriate goals, it then tries to satisfy these goals. This is done in the plan-selection phase of UCEgo. Plan selection in UCEgo is implemented using planfors, as described in section 5, on PAPAN. In UCEgo, planfors are indexed using if-detected demons. The if-detected demons serve to suggest application of a particular planfor whenever an appropriate situation arises. Such situations always include the goal of the planfor, and may include other factors relevant to the planfor. For

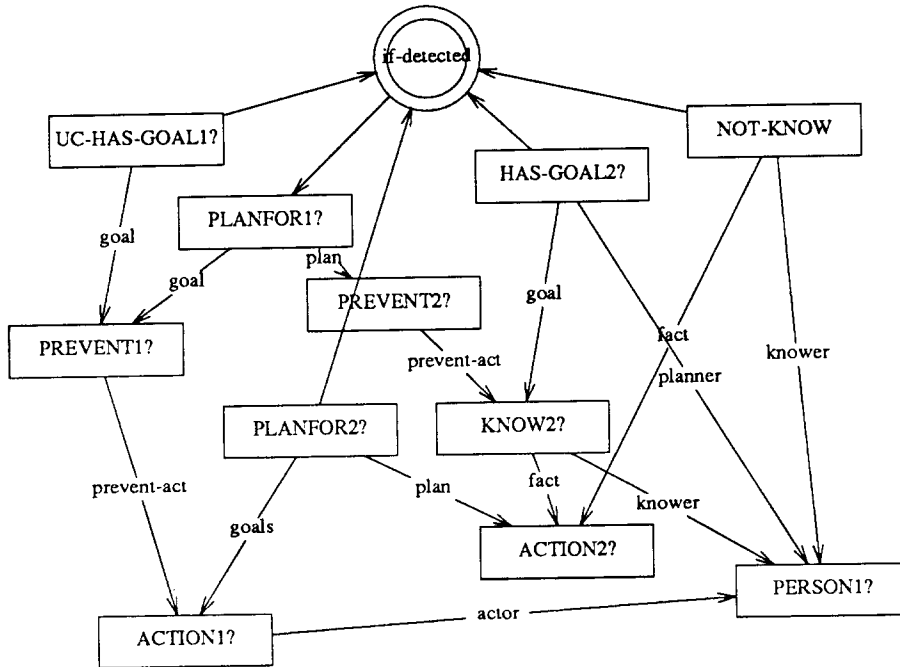


Figure 12

If-detected demon for preventing someone from knowing how to do something.

example, Figure 14 shows an if-detected demon that suggests the plan of telling the user the answer whenever it detects a situation where UC wants the user to know the answer to a query and there is an answer for that query.

Besides encoding the situations when UCEgo should be reminded of particular planfors, the if-detected demons also provide a unification service. For plan selection, unification serves to specialize the general plans stored in the planfors to fit the activating situations. For example, consider the demon shown in Figure 14. After the detection net of the demon is matched, UCEgo will create a new planfor with a plan of telling the user the particular proposition that matched SOMETHING2, which is the answer for the user's query.

After finding a plan, adopts the *intention* of executing that plan. An intention to execute a plan means that UCEgo has scheduled the plan for execution. UC's notion of intention is similar to that of Cohen & Levesque 1987, although UC is not concerned with issues of its own beliefs, since performing actions accidentally or unknowingly is not a real possibility for the system. Figure 15 shows the basic knowledge used by UCEgo to adopt a plan for execution. This if-detected demon states that whenever UC has some goal and UC knows that there is a plan for that goal, then UC adopts the intention of performing the plan.

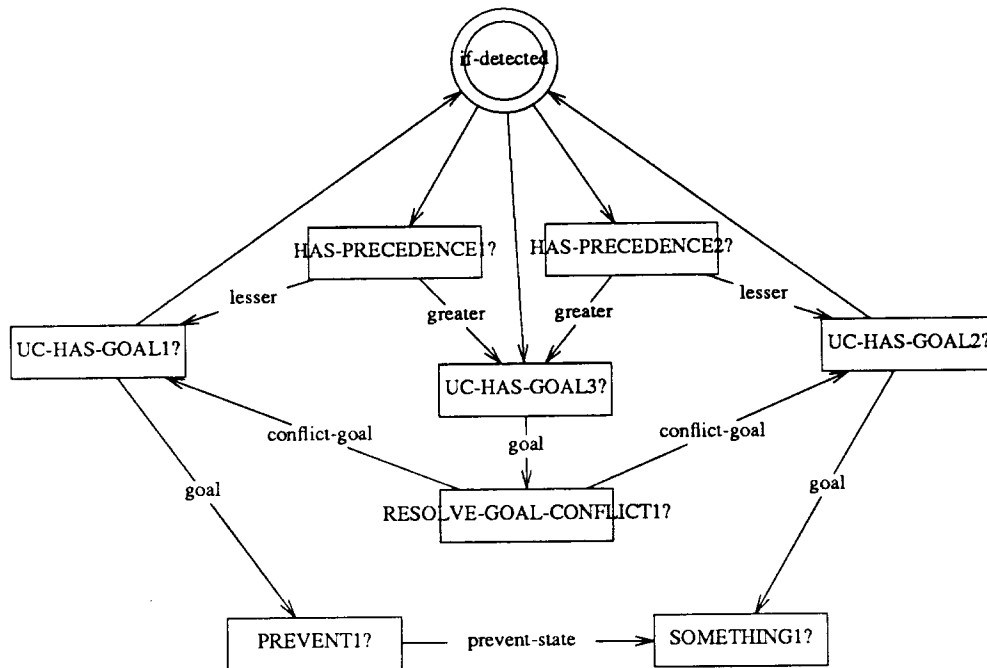


Figure 13
If-detected demon for detecting goal conflicts.

If UC had to choose between a number of plans for a goal, this demon would have to be replaced by something more complex. The idea behind our scheme is that, with a rich enough description of situations for which plans are appropriate, a planner will not typically consider many plans at any given junction. So far, UCEgo has been able to function by simply intending to perform the first plan it finds, and with no ability to for correction. If a more complex planning structure is eventually needed, we would realize it by having a more sophisticated mechanism for forming intentions. For example, the presence of multiple, recalled plans could be represented as a situation in which a meta-goal of choosing between them is to be detected.

After UCEgo has finished forming intentions to perform some actions, it attempts to execute them. UCEgo has some procedural knowledge about which intention to tend to first. For example, it will try to execute meta-plans before non-meta-plans, and subgoals before their parent goals. If there remain unfulfilled conditions of a plan, its execution will not be attempted. Otherwise, the selection among intentions is random.

Having selected a plan, UCEgo proceeds to execute the plan. UCEgo's demon interpreter calls the proper subcomponent to perform the action. An example of this is when UCEgo calls UCEexpress to perform a TELL action.

In the case of executing a meta-plan for a meta-goal, the procedure called by the interpreter typically alters the plan structure itself. For example, in the previous section,

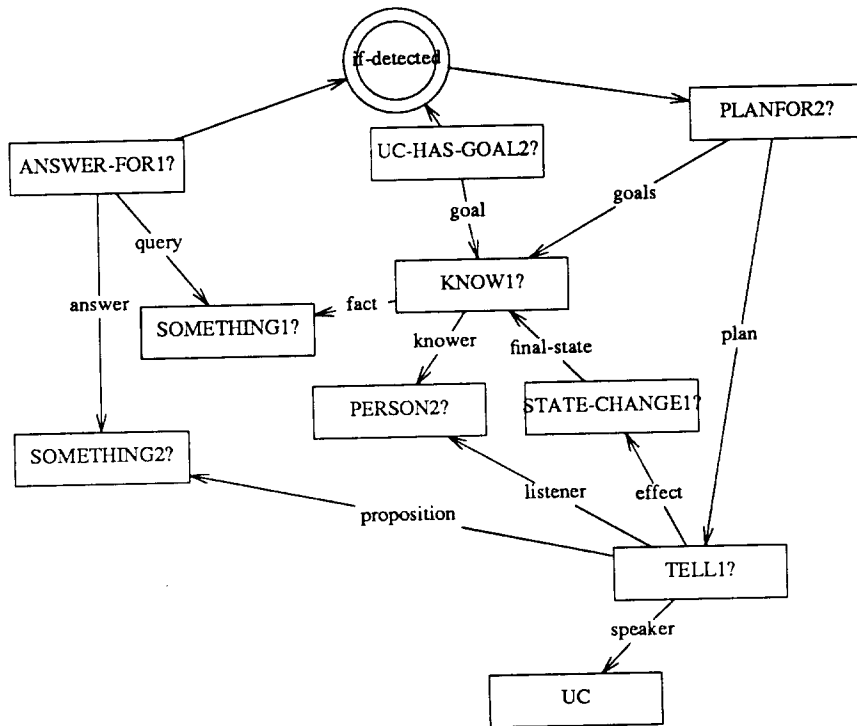


Figure 14
If-detected demon for suggesting the plan of telling the user.

UCEgo had inferred the meta-goal of resolving a conflict between two goals, one of helping the user and one of protecting itself. In plan selection, a demon would propose a meta-plan called ABANDON-GOAL to use to address this meta-goal. This meta-plan would cause a procedure to be invoked that resolves the conflict by abandoning the less valuable goal. To determine which goal is less important, ABANDON-GOAL first searches the data base for an explicit HAS-PRECEDENCE relation between the two goals. If such a fact cannot be found, ABANDON-GOAL examines the parent goals of the goals in question for such a fact. If that fails, the ultimate sources of the goals, usually some themes, are examined. All of UC's themes are in an explicit precedence relation. Thus, most conflicts can be resolved by appeal to the priority of themes. In the example from the previous section, UC's staying alive theme is found to have a higher precedence than its consultant theme, thus causing the latter goal to be abandoned and the former retained.

6.2. KNOME - UC's User Model

KNOME is a program that maintains UC's model of the user. This model represents UC's understanding of the user's knowledge of UNIX. No attempt is made to determine other user attributes, such as personality traits [Rich 1979], user preferences [Morik &

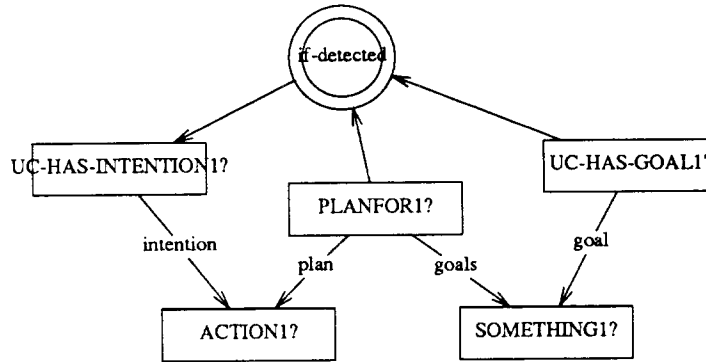


Figure 15
Principal if-detected demon used to adopt a plan.

Rollinger 1985], or user performance [Brown & Burton 1976]. This knowledge state is exploited by the goal-detection phase of UCEgo's processing and by KIP and UCExpress.

KNOME separates users into four categories or stereotypes, corresponding to different levels of expertise: novice, beginner, intermediate, and expert. Each category encodes information about the knowledge state of users that belong to that category. Conflicting information about an individual user's knowledge state can be encoded explicitly, and will override inheritance from the user's stereotype category. Thus, the user categories are prototypes that are used as reference points for inference [Rosch 1978].

Besides separating users according to expertise, KNOME also categorizes commands, command formats, terminology, and other relevant knowledge. These objects are grouped according to their typical location on the learning curve (i. e., when the average user would learn the information). The categories include *simple*, *mundane*, and *complex*. A further category, *esoteric*, exists for those concepts that do not consistently lie on any one area of the learning curve. These concepts are usually related to special-purpose requirements, and only users that require that special purpose will learn those concepts. Thus, esoteric concepts may be known by novices and beginners as well as by intermediate or expert users, although advanced users are still more likely to know more esoteric items simply because they have been using UNIX longer.

The double stereotype system described above is extremely space efficient. The core of KNOME's general knowledge of users is summarized in the five statements shown in Figure 16 (although the information is stored as assertions in KODIAK).

6.2.1. Inferring the user's level

During a session, KNOME builds a profile of the user and infers the user's level of expertise. This proceeds in a two-step process. First, KNOME infers particular facts about what the user does or does not know from the dialogue, i.e., from what the user actually said and from the output of the goal analyzer. Next, these facts are combined to

Expert users know all simple or mundane facts and most complex facts.
Intermediate users know all simple, most mundane, and a few complex facts.
Beginner users know most simple facts and a few mundane facts.
Novice users know at most a few simple facts (e.g., the login command).
Any user may or may not know any esoteric facts, but more experienced users are more likely to know more esoteric facts.

Figure 16
Summary of KNOPE's General Model of Users.

infer the user's level of expertise.

Inferring particular facts about what the user does or does not know is implemented using if-detected demons as a rule-based system. An example of such a rule is:

the user wants to know ?x → the user does not know ?x

This rule is implemented using the if-detected demon shown in Figure 17.

KNOPE currently distinguishes seven classes of inference rules. These are summarized as follows:

- Claim – inferences based on what the user claims to know
- Goal – inferences based on what the user wants
- Usage – inferences based on the user's usage
- Background – inferences based on the user's background
- Query-reply – inferences based on user's replies to questions
- No-clarify – inferences based on the user's not asking for clarification

Claims cover such items as the statement by a user that he or she knows some fact. Examples of *goal* inference are that, if a user asks how to do something, then it is reasonable to assume that the user does not know how to do that thing, and is also unfamiliar with the command that does it. *Usage* inferences include such notions as that the user can be assumed to know the commands the user has used. *Background* inferences mean that, should UC know the user's background and have a stereotype for that background,

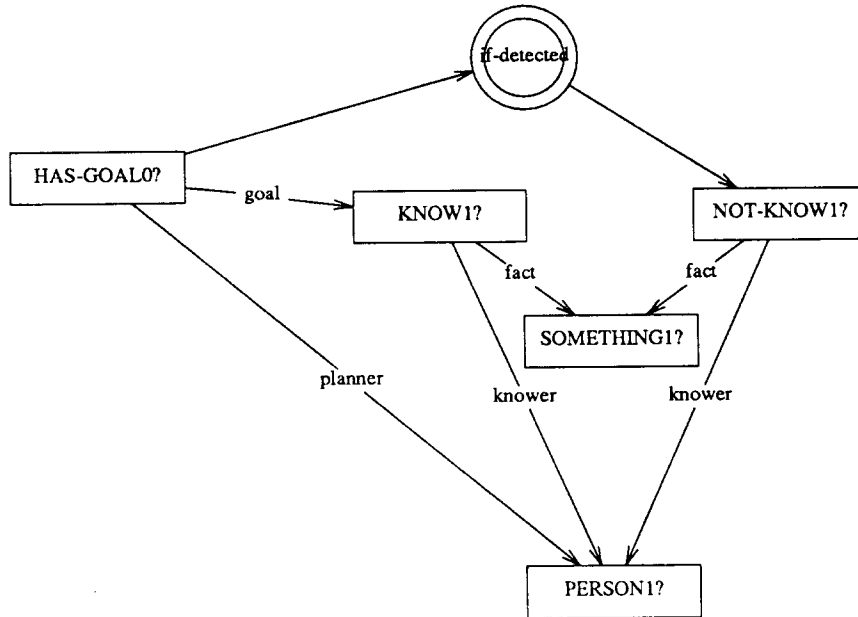


Figure 17

If-detected demon used for inferring that the user does not know something.

UC should assume that the user knows what is indicated by that background. *Query-reply* refers to the possibility that UC asks the user for information about the user's knowledge state. Finally, *no-clarify inferences* are assumptions that the user knows the items to which UC's terminology refers if the user does not ask for clarification of them.

Based on its understanding about what the user does or does not know, **KNOME** can infer the user's level of expertise. An example of such an inference rule is:

the user does not know a SIMPLE fact → it is **LIKELY** that the user is a **NOVICE**, it is **UNLIKELY** that the user is a **BEGINNER**, and it is **FALSE** that the user is an **INTERMEDIATE** or an **EXPERT**

Such evidence is combined to arrive at a likelihood that the user have a given level of expertise. Such ranking is updated continually as the interaction with a given user progresses. At any given point, the most highly ranked level is the one the user is assumed to be at. For more details on this and other issues addressed in UC's **KNOME** (e.g., dealing with the inherent uncertainty of information encoded in the model, representing individual users, etc.), see Chin 1986 and Chin 1988.

6.2.2. Some applications of KNOOME in UC

The user model is exploited in many places in UC. For example, to decide whether some goal involving the state of the user's knowledge is plausible, PAGAN will check to see if the user already knows the fact in question. If the user does, PAGAN will not attribute the goal to the user. Thus, if the user asks a question with both a possible direct and indirect interpretation, like "Do you know how to print a file on the imagen?", and the KNOOME concludes that the user knows how to print a file on the imagen, it will reject the indirect interpretation.

KNOOME is also used extensively by UCExpress, UC's expression mechanism. For example, generating a response to simple question once the answer is known is a function of the knowledge state of the user. In particular, UCExpress will illustrate a response with an example only if it believes the user is a novice. Also, UCExpress will use its "simile" format, i.e., using one command to illustrate another command, only when it believes the user knows the first command. Of course, establishing such a belief is the job of KNOOME. UCExpress is described further in section 8.

6.2.2.1. Detecting misconceptions

One of the more interesting uses of KNOOME in UCEgo is to detect user misconceptions. A misconception occurs when the user believes something that UC believes is false. An example of a user misconception occurs when the user asks, "What does ls -v do?". Here, the user believes that there is an 'ls' command, that '-v' is an option of the 'ls' command, and that there is a goal for the plan of 'ls -v'. Here, '-v' is actually not an option of 'ls', even though ls will accept and ignore extraneous options.

KNOOME is responsible for detecting what the user believes, comparing this with UC's knowledge, and then either deducing that the user knows the fact if what the user believes coincides with UC's knowledge, that the user has a misconception if the user's belief contradicts UC's knowledge, or that the user may know something that UC is unfamiliar with. The last possibility, namely, that UC does not know everything about UNIX, means that the system cannot use a simple closed-world hypothesis (which implies that if a fact cannot be deduced from the database, then it must be false) such as is used in other misconception-detection systems (e.g., Mays 1980, Kaplan 1983, Webber & Mays 1983, and McCoy 1983). The other possibility is an open-world hypothesis, where if a fact cannot be deduced from the database, then the system has no information about it. Using a pure open-world hypothesis, a system would have to encode complete information about what cannot be the case, in order to detect misconceptions. This is inefficient at best and at worst not possible.

What KNOOME does instead is to augment an open-world hypothesis with *meta-knowledge*. Here, the term 'meta-knowledge' is used to denote knowledge that KNOOME has about what UC itself does or does not know. For example, KNOOME contains the information that UC knows all the command options of all simple commands. Hence, if a particular option is not represented in UC's knowledge base as a possible option for a particular simple command, then that is not a legal option for that command. Using such meta-knowledge, KNOOME is able to infer that '-v' is not an option of 'ls', hence, that the user has a misconception. This fact is passed on to UCEgo, which adopts the goal of correcting the user's misconception.

6.3. Example

To see how UCEgo works in more detail, consider again the example, “Do you know how to print a file on the imagen?”. Recall that 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.

The first part of the annotated trace below shows the goal-detection phase of UCEgo. The explanatory comments that have been added to the trace are printed in italics.

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))))
```

UC-HAS-GOAL19 represents UC’s goal of helping the user (HELP2).

HAS-GOAL-ga0, which is also shown in Figure 9, 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.

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))))
```

UC-HAS-INTENTION6 represents UC’s intention to satisfy KNOW-ga0.

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*))))))
```

UC-HAS-GOAL20 represents UC’s goal of the user knowing how to print a file on the imagen.

Annotated trace of UCEgo’s goal detection process.

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 detection net of the if-detected demon shown in Figure 10. On activation, the if-detected demon adds a copy of its addition net to UC's memory. Here, the addition net consists of 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, 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, which is shown in Figure 15. 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. The following annotated trace shows this part of the processing (additional explanations are in italics):

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))
```

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.

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))))))
```

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.

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 &))))))
```

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.

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 &)))
```

UC-HAS-INTENTION7 represents UC's intention of telling the user (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))))))))))
```

UCexpress1 represents a call to *UCExpress* to execute *TELLA*.

Annotated trace of *UCEgo*'s plan selection process.

The first step of the plan is to call **KIP**, the planner component of UC. Figure 18 shows the if-detected demon that calls **KIP**.

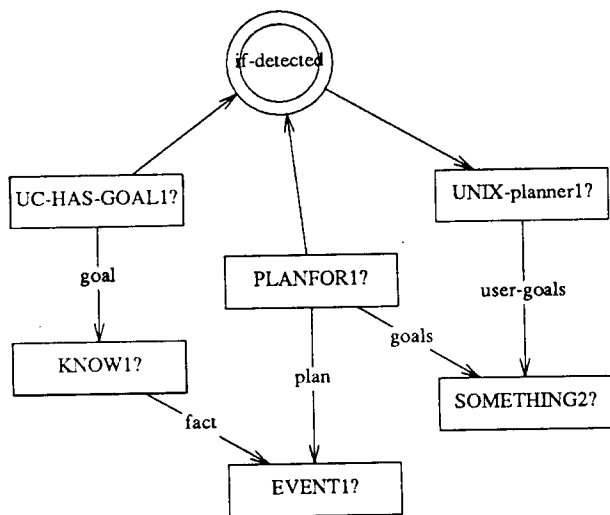


Figure 18
If-detected demon for calling the UC planner, **KIP**.

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 the if-detected demon shown in Figure 14 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 the answer leads to UC's intention to tell the user the answer (*UC-HAS-INTENTION7*) via the if-detected demon shown in Figure 15. 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.

7. The Planner

7.1. Introduction

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.

7.2. Planning Process in **KIP**

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.

7.2.1. Goal detection

KIP begins with the user's domain goals, passed to it by **UCEgo**. As it proceeds with the planning process, other goals may be detected. These fall into the following categories: A condition that **KIP** believes is unmet and whose omission would cause the plan to fail becomes a new goal (i.e., a subgoal of a current goal). Another kind of goal arises when some current plan being considered impinges upon an *interest* of the user. An interest is some situation, real or hypothetical, toward which the user has a positive stance. In UNIX, examples of interests are having access to files and maintaining a low system-load average.

In UC, interests generally give rise to goals of defending against a threat to that interest. Since such a threat is generally the inadvertent consequence of some plan under consideration, there is often a goal conflict between a user goal and the goal resulting from the threatened interest. As in **UCEgo**, such conflicts are handled by detecting a meta-goal of resolving that conflict. For example, if a proposed plan involves deleting a file, this goal might conflict with the goal of having access to that file. The meta-goal of resolving this conflict is therefore detected.

7.2.2. Plan selection

Plan selection is the process of determining a potential plan to satisfy the user's goals. This potential plan is then examined during the rest of the planning process. If the plan is judged adequate, it is returned to **UCEgo**; otherwise, this plan is modified or another plan is selected.

One simple method for performing plan selection is to choose a stored plan that is indexed in the knowledge base as addressing the goal to be achieved. This is done by using the same matching procedure used by **PAGAN** to examine knowledge structures relating plans to goals, using the same representational format used by **PAGAN**. Such knowledge structures are used to represent only the conventional functions associated with commands, e.g., that 'rm' is a plan to delete a file, or that 'lpr -Px' prints on the printer x.

7.2.2.1. New plans

When **KIP** has no stored plan for a particular goal, it employs a kind of *means-ends* analysis strategy [Newell & Simon 1972]. **KIP** assumes that the best way to reduce the difference between the user's goal and the present state is by determining the goal most similar to the current goal, and trying to fulfill that goal. **KIP** finds a similar goal by using its taxonomy of goals to locate a goal that is dominated by the same parents as the user's goal. This algorithm for finding a plan is called the Goal Similarity Matching Algorithm (GSMA).

For example, there is no command in UNIX for moving a file to another machine. This goal is represented as achieving the conditions of having a file of the given name and contents on the target machine, and not having such a file on the source machine. In this case, **KIP** searches for a plan of a goal most similar to the goal of moving a file to another machine. It does this by finding a goal that shares more common parents with

moving a file to another machine than any other goal. Since moving a file to another machine is dominated by Ethernet (machine-machine links) goals and file-transfer goals, **KIP** searches for plans of goals that are dominated by these two goals. Figure 19 shows that one command, 'rcp', falls in this category. This command is used to copy a file from one machine to another.

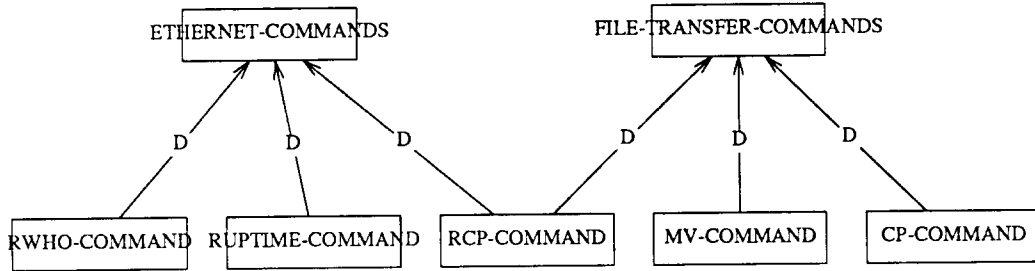


Figure 19
Hierarchy of Ethernet and File Transfer Commands

KIP selects the plan of using the 'rcp' command as a *potential* plan to move a file to another machine. When this plan is tested in a subsequent phase of the planning process, it will be found not to satisfy all the goals of the user, since the goal of not having such a file on the source machine is left unaddressed. Plans for these remaining goals will be selected by subsequent iterations of the planning algorithm.

7.2.2.2. Plan specification

Once a plan has been selected, **KIP** makes a copy of the plan with specific values filled in. For example, the user asks:

How do I delete the file named junk from my directory?

The general plan for this planning situation is to use the 'rm' command. This is stored in the knowledge base as shown in Figure 20.

During plan specification, **KIP** creates a new instance of the EXECUTE-UNIX-COMMAND and fills in the appropriate specific values of its arguments by looking at the general plan. This specific plan, shown in Figure 21, specifies that the value of the del-object (the file to be deleted) is FILE1 (whose filename is *junk*) and the value of the argument to the 'rm' command is the string "junk". In other words, this represents that the 'rm' command with the argument "junk" is a plan for deleting a file whose name is *junk*. It is this specific plan that is tested during the rest of the planning process.

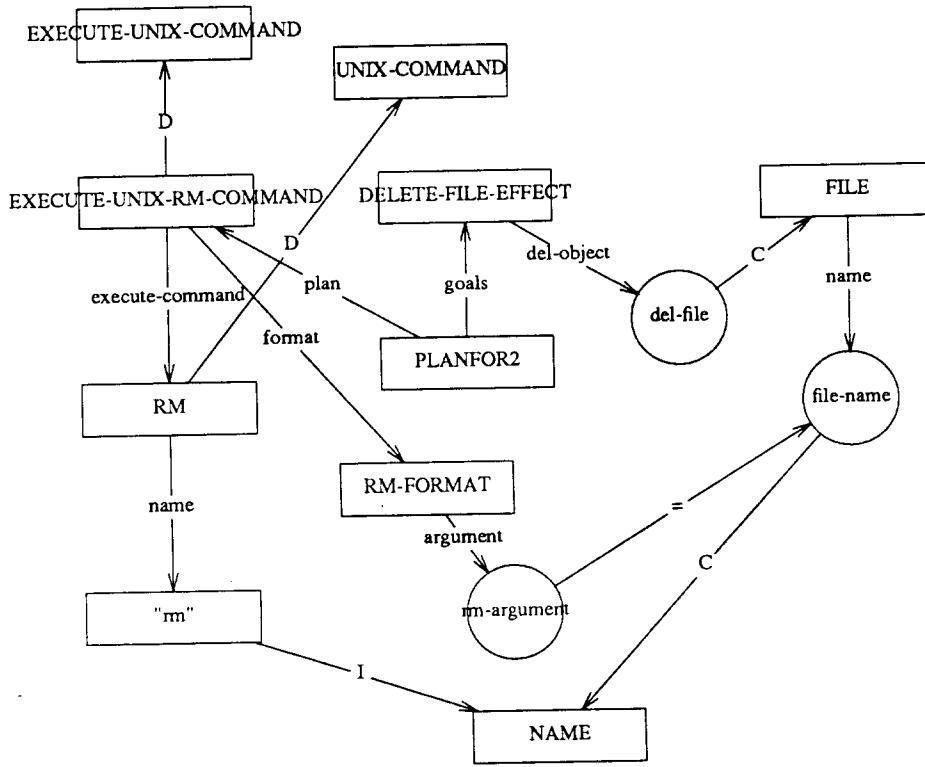


Figure 20
Representation of plan to delete a file by using the 'rm' command

7.2.3. Projection

It is next necessary to test whether the plan as developed so far would actually execute successfully. Potential problems in the plan include both conditions that must be satisfied and goal conflicts that must be resolved because of the effects of the plan.

While detailed descriptions of all UNIX commands are in the knowledge base, KIP does not actually check to see that all the condition of success are met or that no goal conflict is possible. First, there are too many conditions to check, even in very simple situations. Second, UC usually does not have the information to determine the answer. For example, in order to print a file, the printer must be in working order, there must be a continual supply of electricity, the user must have an account, etc. Rather than checking all such conditions, it would be desirable to check only those that seem to constitute a reasonable cause for concern.

This is done in KIP by storing a knowledge structure called a *concern*. A concern represents an aspect of a plan that should be considered a potential cause of plan failure. These encode some user's experience. For example, presumably, the printer's being in working order is not a frequent obstacle to printing a file, but having read permission

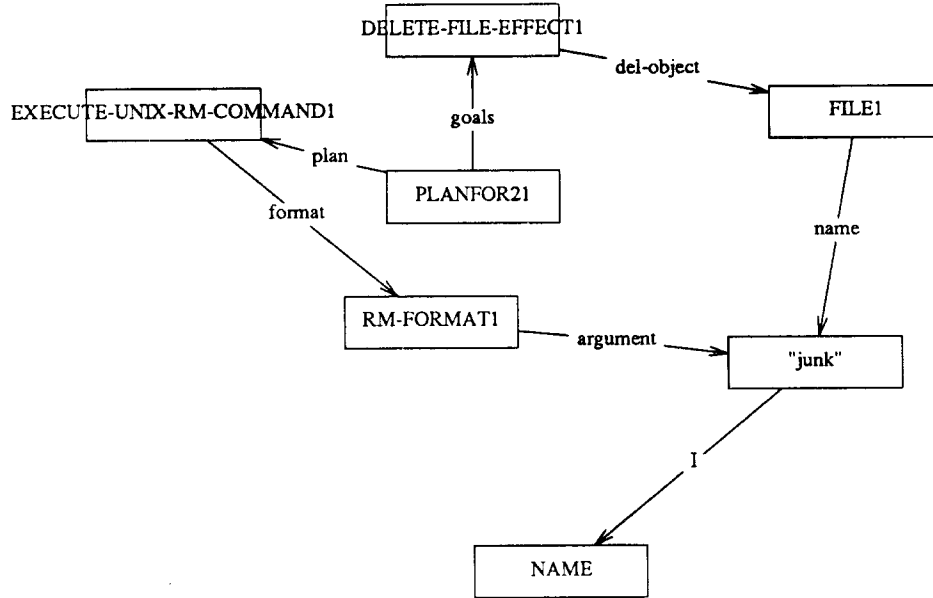


Figure 21
Representation of specific plan to delete a file named *junk*
by using the 'rm' command with argument "junk".

might be if the file to be printed belongs to someone else.

There are two basic kinds of concerns in KIP, condition concerns and goal-conflict concerns. Condition concerns specify those conditions of a plan that are likely to be unmet, while goal-conflict concerns specify the consequences of a plan that are likely to threaten a user's interest. KIP examines only those aspects highlighted by a concern in determining whether to accept a plan. In general, concerns that are judged insignificant are ignored, those judged highly significant lead to plan failure, and moderate concerns are passed on to be expressed as warnings to the user. For example, in the following transaction, a moderate condition concern is found:


```
# How do I print out Jim's le named test1?  
Type 'lpr test1'. This plan will not work if you don't have read  
permission on Jim's le.
```

Since not having read permission on someone else's file is stored as a moderate cause for concern, the planner accepts the plan, but passes along the concern to the expression mechanism. Doing so leads to the warning statement following the suggestion to use the particular plan.

Note that in the example above, the concern is retrieved only because the user made a request that involves access to someone else's files. It is typical of concerns that some are appropriate only when a default condition is violated. **KIP** believes that the default for using UNIX is to access one's own files, so it retrieves concerns about access only when this situation does not pertain.

KIP makes considerable use of default knowledge in processing concerns. Also, concerns may be attached to plans at any point in the hierarchy, so that they will be accessed only for the specific conditions in which they might apply. For example, **KIP** knows that the Apple printer has a small paper tray and often runs out of paper. This knowledge is encoded by specifying as a default that this printer is out of paper. Thus, without additional knowledge of the state of the printer, **UC** will produce the following interaction:

```
# How do I print a le on the apple printer?  
Use 'lpr -Pap'.  
But rst, ll the printer with paper.
```

In contrast, if we explicitly assert to the user model that the print is full of paper, we get the following response:

```
# How do I print a le on the apple printer?  
Use 'lpr -Pap'.
```

A goal conflict concern is illustrated by the following example:

```
# How can I free up disk space?  
Use rm. However, any file that is removed cannot be recovered.
```

In this example, the only plan KIP can come up with to free up disk space is to delete files. However, a goal conflict concern informs the planner that this plan is likely to conflict with the user's of preserving access to his or her files. KIP therefore generates a warning to accompany this plan.

Another example of the use of concerns occurs in the following interaction:

```
# How do I find out the files in the directory named /usr/local on my sun?  
Type 'ls /usr/local | more'.  
However, you should first change the current directory to the directory  
named /usr/local, by typing cd /usr/local.
```

Here KIP detects two concerns. One pertains to a bug in the SUN implementation of the UNIX 'ls' command, in which asking for a listing of a directory mounted on a different disk than one's current directory occasionally causes no output to result. This bug has been made into a concern about using the 'ls' command in such a situation. The concern causes the planner to issue a warning to change one's current directory. The second concern is that listing large directories may cause output to scroll off the screen. Since the directory in question is typically large, PAGAN alters its plan to pipe the output through the 'more' filter. This example also makes use of default knowledge in that KIP assumes that the user might not be connected to the same place as /usr/local, and that this directory is large.

KIP has a rather elaborate taxonomy of concerns and a method of processing them. In addition to classifying concerns by whether they are appropriate to violated defaults or not, or to conditions or goal conflicts, whether the effect of an command is intended by the user is important for generating a concern. For example, it is probably inappropriate to generate a warning about not being able to recover files if the user asked to delete a file, for example. The details of KIP and how it handles concerns are described in Luria 1987 and Luria 1988.

7.3. Example and Trace Output of KIP

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 able 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 22 along with its most pertinent connections. Following that is an annotated trace of KIP's processing for

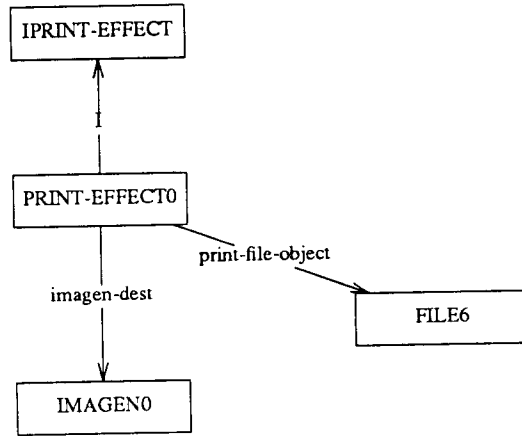


Figure 22
Input to KIP from "Do you know how to print a file on the imagen?":
PRINT-EFFECT0 and important links

this example:

Planner is passed:

```
(PRINT-EFFECT0  
  (print- le-object = FILE6)  
  (imagen-dest = IMAGEN0))
```

The planner was passed a single goal.

```
Looking for stored plan for PRINT-EFFECT0  
Looking for speci c 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 con ict concerns
```

```
Making a new instance of EXECUTE-UNIX-IPR-COMMAND
```

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 23.

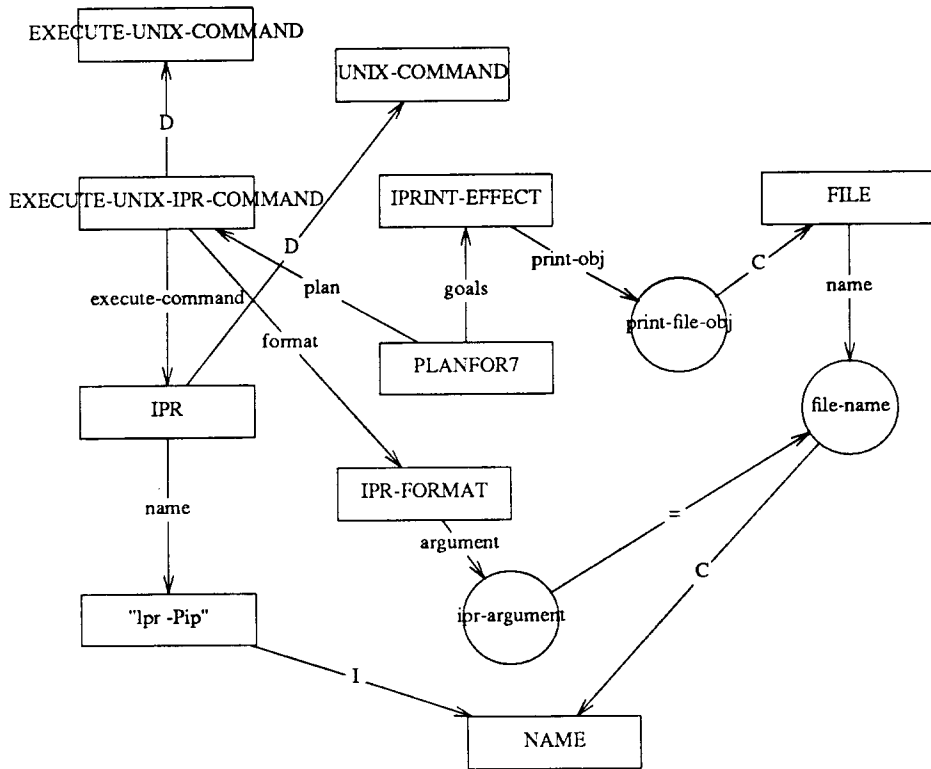


Figure 23
Representation of plan for printing a file on the imagen

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 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.

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- le 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

A copy of the plan has been made. Note that **KIP** has made NAME6 a hypothetical object, because it is the name of FILE6, which is itself hypothetical.

Planner returns:

PLANFOR70

Planner produces:

```
(PLANFOR70 (goals = PRINT-EFFECT0)
  (plan =
    (EXECUTE-UNIX-IPR-COMMAND0
      (ipr- le = FILE6)
      (ipr-execute-command = "lpr -Pip")
      (ipr-format = (IPR-FORMAT0
        (ipr-format-arg = NAME6))))))
```

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 24.

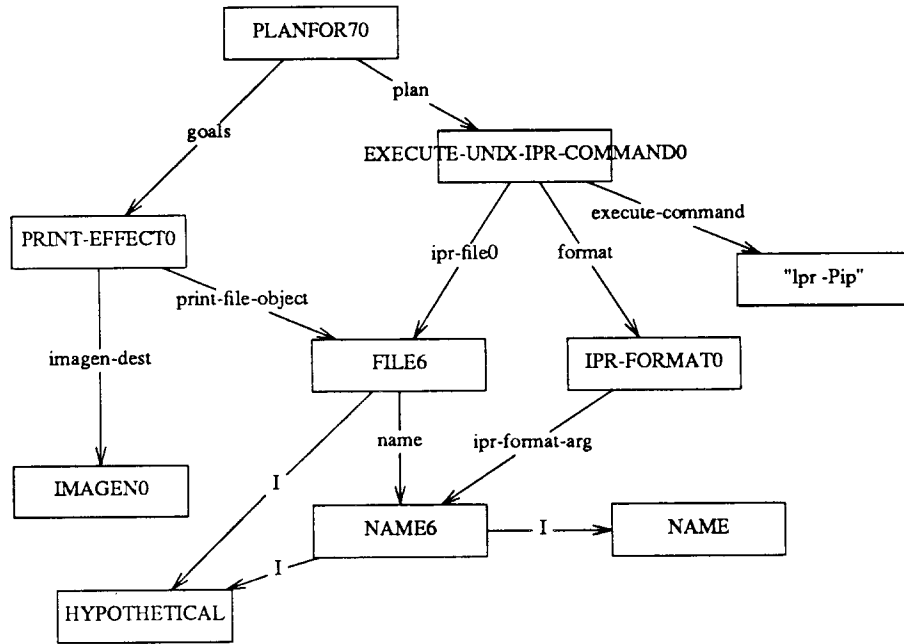


Figure 24
Output from KIP for example "Do you know how to print a file on the imagen?"

8. The Expression Mechanism

8.1. Introduction

After UC has determined the correct conceptual response to the user, this response must be expressed to the user in a clear, concise manner. Consider the following example:

User: What is a directory?

A1: A directory is a file.

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

A3: A directory is a file. Only empty directories can be deleted. Directories cannot be edited. Directories contain files. Directories form a tree-like structure. Directories always contain themselves and their parents. A directory can be listed by using the 'ls' command.

The different replies are all correct, but contain differing amounts of information. The first answer does not give enough information to distinguish between files and directories. The second answer provides just enough information in a concise form and is the best answer in such a situation. The third answer is too verbose and overwhelms the user with irrelevant information. An answer like A3 would be more appropriate in cases where the user had requested, "Tell me all you know about directories." This example illustrates the problem of determining how much of an answer to express to the user. This is similar to Luria's work [1982] on answering *why* questions in a story-understanding system. He pointed out that answering why-questions requires two mechanisms, one to find the causal chain that represents the answer and another *expression mechanism* to determine how much of the causal chain to tell the user.

Another problem in answer expression is deciding what format to use in presenting the answer. This problem corresponds to Grice's Maxim of Manner [Grice 1975]. Consider the following scenario:

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

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

A2: Use 'rcp' to copy the file to another machine and then use 'rm' to delete it.

For example, to move the file foo to the file foo2 on machine dali, type 'rcp foo dali:foo2'.

The first answer is correct and general, but it is so verbose that it is undecipherable. On the other hand, the second answer is succinct and gives the user information in an easily readable form, but is considerably less general. The second answer is somewhat inaccurate, since it applies only to copying a file named foo to a file named foo2. It is up to the reader to apply analogous reasoning for other cases. Despite this lack of generality, the second answer form is superior to the first. Note that for a program to format the answer in the second form requires additional computation to transform the general solution into an example. A natural-language system needs to incorporate knowledge about when and how to use special presentation formats like examples to more clearly convey information to the user.

The concerns of how much information to present to the user and of what format to use correspond respectively to Grice's Maxims of Quantity and Quality [Grice 1975]. Such considerations can be considered part of language generation; however, there are enough differences in both the necessary knowledge and the processing to separate such strategic concerns from the tactical problems of generation such as agreement and word selection. Such strategic problems are the domain of an expression mechanism.

8.2. Approach

UC's expression mechanism, UCExpress, uses a two step process: pruning and formatting. In the pruning stage, UCExpress marks those concepts it does not wish to communicate to the user. In the formatting stage, the concepts are transformed by a planning process into more easily understood formats. The result is a set of concepts that are ready for generation into natural language.

The guiding principle of the pruning process is to avoid telling the user anything that the user already knows. Currently, UC models two classes of such information. The first is semantic information about the user's knowledge of UNIX-related facts. Such knowledge is modeled by KNOME (cf. section 6.2). The second class of information is episodic knowledge from a model of the conversational context. The current conversational context is tracked by keeping an ordered list of the concepts that were communicated in the current session. Any concept that is already present in the conversational context or that KNOME deduces that the user already knows is marked and not communicated to the user.

For example, suppose a user asked UC the following question:

How can I compact a file?

The conceptual answer passed to UCExpress is a rather complex network. If it were directly paraphrased into English, the result might be something like the following:

A plan for compacting a file is to use the 'compact' command with the format being 'compact' followed by the name of the file.

However, UCExpress prunes the concepts corresponding to “compacting a file”, since they are already in the conversational context. If UC believes that the user is at least a beginner, then the format of the ‘compact’ command would be pruned as well, since KNOOME will deduce that the user knows this fact from its belief that such users know the format of simple commands. In this case, UC will eventually generating the following reply:

```
Use compact.
```

On the other hand, if the user were a novice, then UC could not assume that the user already knew this format, and would have provided a more elaborate answer. However, rather than generate the answer as an abstract but more detailed description, UCExpress would favor the example format, as described below, and produce the following reply:

```
Use compact. For example, to compact the le foo, type 'compact foo'.
```

Indeed, for the examples in this paper, we have generally caused UC to believe that the user is a novice, in order to illustrate the formatting mechanism.

After pruning, UCExpress enters the formatting phase, where it tries to apply different *expository formats* to express the concepts in a clearer manner. Currently, there are three format types: *example*, *definition*, and *simile*.

Definition format is used to explain terminology. Generally, the need to do so is a result of the user asking a direct question, such as “What is a directory?” The heuristic procedure used to satisfy such requests is to find a category dominating the one in the query, and then find information that distinguishes the subcategory from it. The following are examples of definition format:

```
# What is a directory?  
A directory is a le that is used to contain les.
```

```
# What is a le?  
A directory is a container that is used to contain text,  
code or les.
```

```
# What is a rm?  
Rm is a command that is used to delete les.
```

```
# 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.
```

A directory represented as a kind of file. Moreover, it is distinguished from other kinds of files in that it contains files. Similarly, a file is represented as a kind of container that is distinguished from other containers in that it contains some sort of information. The portion of the network pertaining to these concepts is used to generate the first two responses above.

Simile format expresses information in terms of other information the user is believed to know. This format is invoked when UCExpress attempts to explain a command that has sibling or a parent that the user is believed to know, and which is thought to be similar enough to the command to be expressed. The judgment of similarity is made by counting the number of concepts at which the two networks differ. If this is below some arbitrary threshold, the simile format is adopted. An example of the use of the simile format is the following:

```
# What does ruptime do?  
Ruptime is like uptime, except ruptime is for all machines on the network.
```

The most common expository format is the *example* format. Examples were shown by Rissland to be important for conveying information [Rissland 1983, Rissland et al. 1984]. In UC, examples are used to illustrate plans such as those involving the format of UNIX commands. Unlike Rissland's examples, which are prestored and fetched, UC's example format requires additional computation to transform the general plan into an example. This is accomplished by stepping through a plan and transforming general categories into specific ones. First, if a category dominates other categories, one of those is chosen (at random); then, where the representation can be interpreted as stating that an individual is needed, an individual is created to replace a general categories. Where an individual name is required, it is chosen from a list at random, taking care to avoid conflicts with existing names.

Consider the following UC dialogue:

```
# How can I change the read permission of a le?  
Use chmod.  
For example, to remove group read permission from the le named foo,  
type 'chmod g-r foo'.
```

In the conceptual answer, the last argument of 'chmod' was a pointer to "the name of the file whose protection is to be changed." To give an example, a concrete name for the file is needed, so "foo" was arbitrarily selected for use in the example. Since the user did not explicitly mention the type of permission (user, group, or other), this was specified to be group permission in the example. Similarly, "change permission" was further specified into "remove permission." Since these items are not yet known to the user, they will cause expressions like "the file named foo" to be produced when the example is generated.

8.3. Example

To see how UCExpress works in more detail, consider the example, "Do you know how to print a file on the imagen?" 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 25.

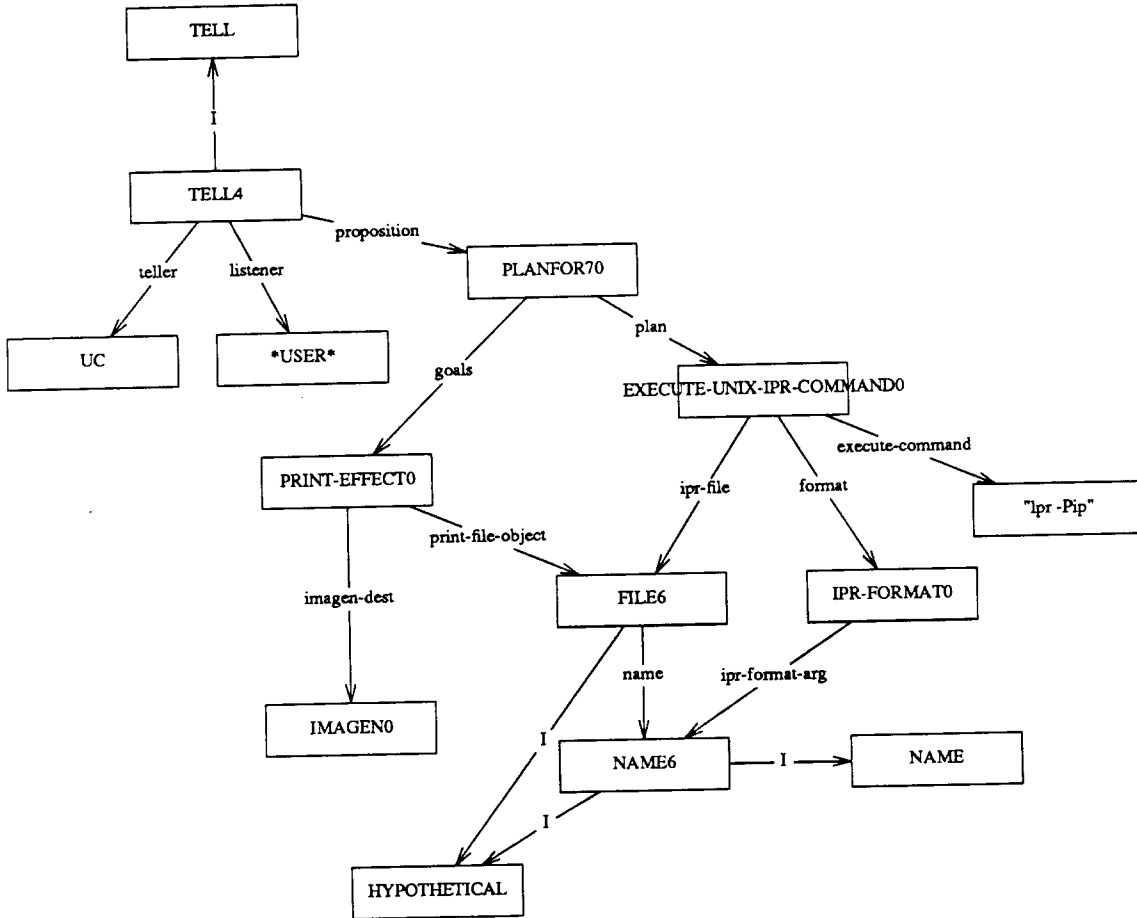


Figure 25

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

This input conceptual network is UCEgo's plan to tell the user the plan that KIP produced (Figure 26, section 7.3). 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.

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:

```
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
      (exemplar =
        (PLANFOR31 (goals = PRINT-EFFECT4)
          (plan =
            (TYPE-ACTION0
              (actor = *USER*)
              (type-string =
                (CONCAT-STRO
                  (string1 = "lpr -Pip")
                  (string2 = "foo"))))))))
      (exemplified = PLANFOR70))))))
```

Trace 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 26 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:

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.

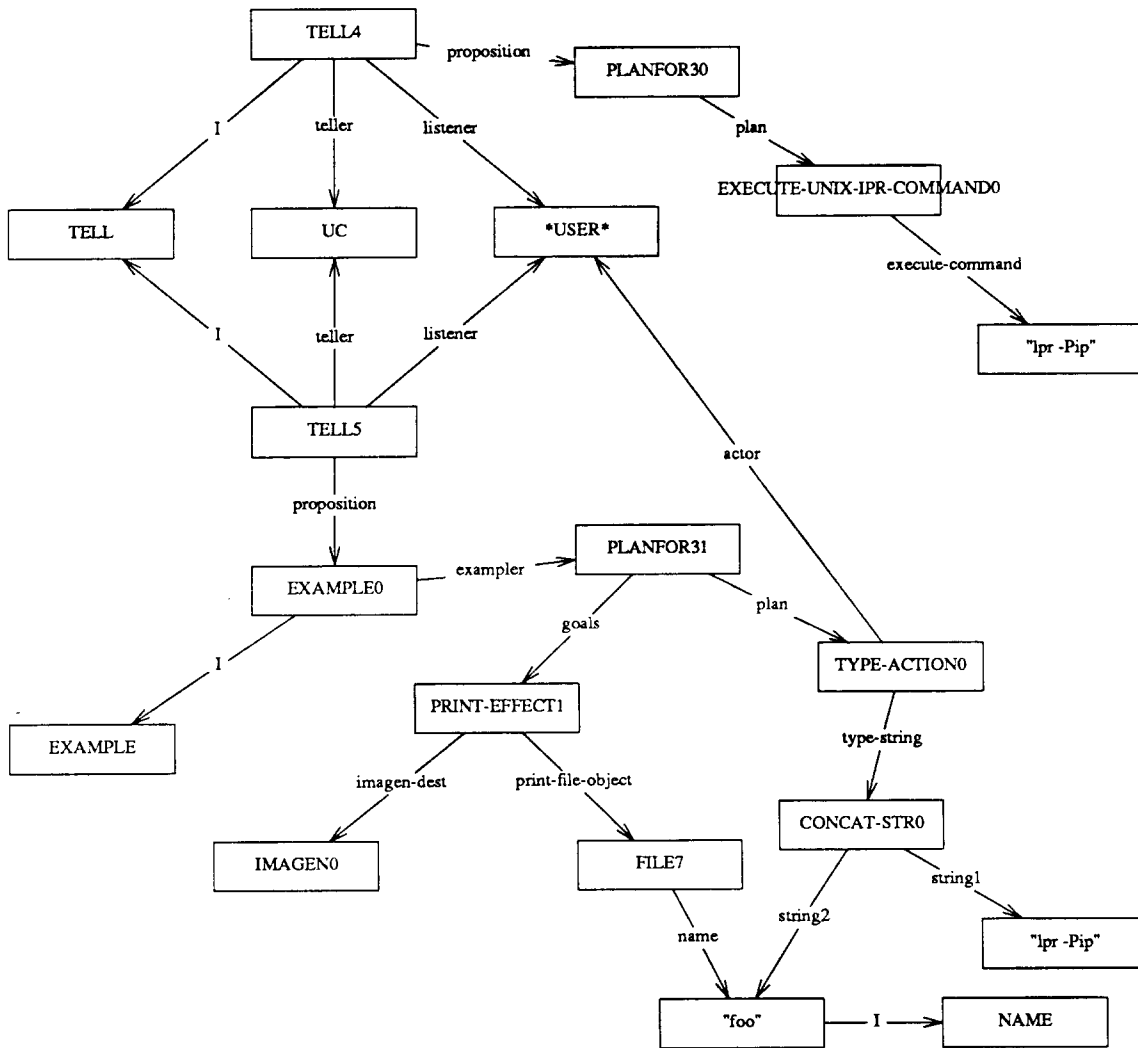


Figure 26
The output of UCExpress for the example "Do you know how to print a file on the imagen?"

9. The Generator

9.1. Introduction

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.

9.2. Design

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.

9.3. Example

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 27. A trace produced while generating this output is given in Figure 28.

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 exemplar)'. This sets up a recursive call to gen with the value of the exemplar 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)'.

The las-pr-file-obj specification causes the generator to find an 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

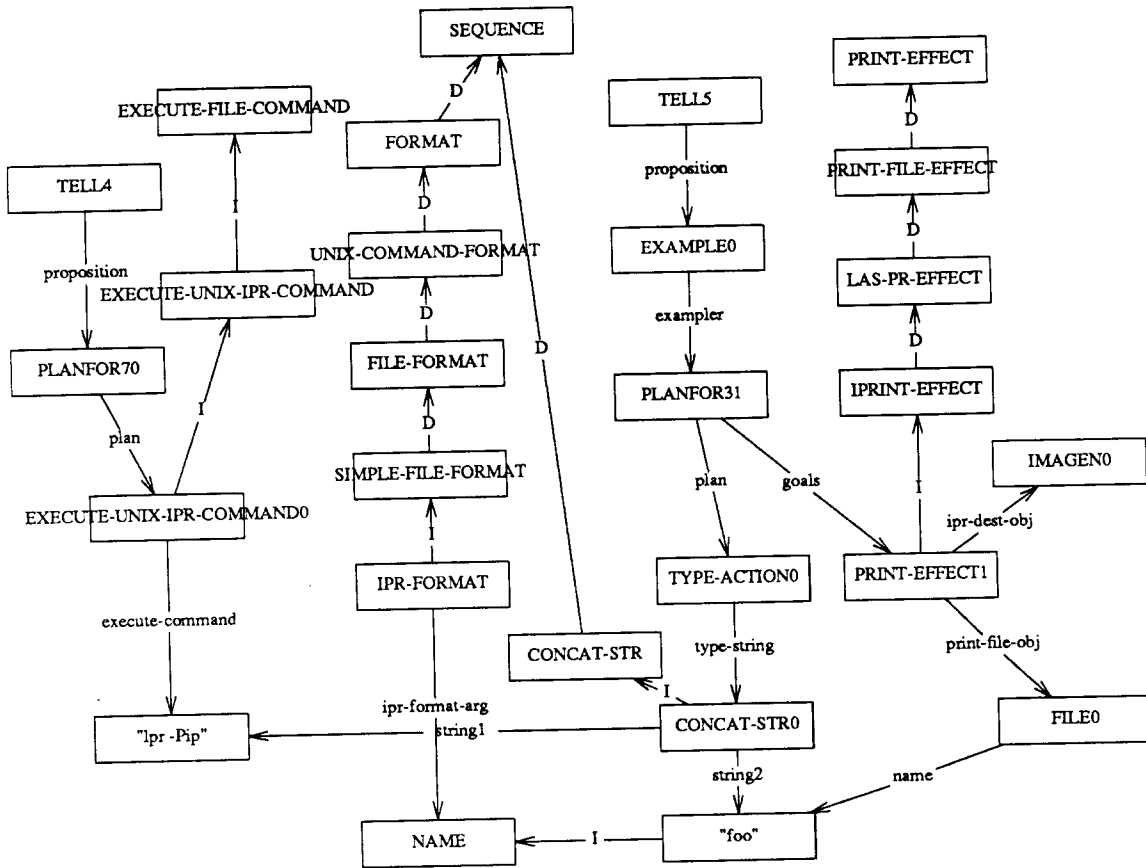


Figure 27
 Knowledge pertaining to generating response to "Do you know how to print a file on the imagen?"

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 exemplar))
Value for exemplar 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- le-obj) on the (las-pr-dest-obj))
Value for las-pr- le-obj is: FILE0
Pattern for FILE0 is: (le named (gen name))
Value for name is: "foo"
Phrase derived from (le named (gen name)) is: (le named foo)
Phrase derived from (las-pr- le-obj) is: (the le 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 le 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 exemplar) is:
(to print the le named foo on the imagen comma type lquote lpr -Pip foo rquote)

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

Figure 28
UCGen trace for the example question.

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

10. Knowledge Acquisition

10.1. Introduction

UC is knowledge dependent. It is the task of the knowledge-acquisition system to facilitate the construction and subsequent extension of the knowledge base. The traditional approach to knowledge acquisition has been to provide the system implementers with special-purpose software tools such as text editors, form systems, and graphic network editors [van Melle 1981].

These approaches require that the person adding the knowledge have detailed knowledge of the knowledge-representation language being used and of the overall design of the system. The typical situation, however, is that the people with the domain knowledge do not have the knowledge necessary to add information to a complex knowledge base. The system builders must therefore extract the information from the experts and then program it in themselves. This is a tedious, error-prone process that is only viable as long as the system builder is in contact with the program. If the system is in use in a real environment, the knowledge base is essentially frozen.

10.2. The UC Approach

Our solution to this problem is to provide a mechanism to allow the interactive transfer of knowledge from the domain expert to the knowledge base using natural language. This is similar to the approach taken in the **Nano-KLAUS** system [Haas & Hendrix 1980]. **UCTeacher** [Martin 1985] is the system being built to provide this facility for UC. The UNIX expert can engage **UCTeacher** in a mixed-initiative dialogue to add more information to UC's knowledge base.

UCTeacher provides both the acquisition of UNIX domain knowledge, and the acquisition of linguistic knowledge. In particular, **UCTeacher** has a mechanism that allows it to guess the metaphoric use of a term in UNIX, based on its knowledge of the metaphoric use of this term outside of UNIX. In this report, we focus largely on domain knowledge acquisition. The mechanism of metaphor extension is beyond the scope of this report, and is described only briefly.

10.3. Acquiring domain knowledge

In the following examples of **UCTeacher**, the knowledge base of UC is set up so as not to contain knowledge of the commands to be illustrated. Also, the prompt 'UCT>' is used to distinguish an interaction with **UCTeacher** from one with UC proper:

(1)

```
% UCTEACHER
UCT> There is a new command.
What is its name?
UCT> sort
What is it used for?
UCT> It is used to sort les.
You mean that typing 'sort lename' will sort the le lename.
UCT> Yes.
```

(2)

```
UCT> Use the rm command to delete les.
You mean that typing 'rm lename' will delete the le named
lename.
UCT> Yes.
```

(3)

```
UCT> Use compact to compress a le.
You mean that typing 'compact lename' will compress the le
named lename.
UCT> Yes
```

UCTeacher uses ALANA and the concretion mechanism to produce a representation of the initial utterance by the expert. In a typical interaction, the initial utterance will provide very sketchy information. The main job of UCTeacher is realize what information is needed, determine what it is, and use it to flesh out the representation. When it believes it has enough information, UCTeacher will paraphrase its understanding to the user for conformation. The user then has the opportunity to confirm or correct UCTeacher's understanding.

UCTeacher has three main sources of knowledge: the user's utterance, the contents of UC's knowledge base, and a store of meta-knowledge associated with concepts in the knowledge base. This use of knowledge plus meta-knowledge makes the knowledge-acquisition process a knowledge-based task (Davis 1976).

UCTeacher uses these three sources of knowledge to acquire all the necessary information from the expert in as natural a dialogue as possible. In example (1), we see the system generating appropriate questions to acquire information that the expert has not initially provided. This ability frees the expert from having to know all the relevant information that must be provided. The system will prompt for all the necessary items. Examples (2) and (3) show the system making full use of the expert's utterance to avoid asking redundant questions.

10.3.1. A Detailed Example

If UC is given a request it cannot service, it generally asks the user to rephrase it. Consider the following sequence of interactions:

(4)

```
# How can I print a le?  
I'm sorry, I didn't understand what you said. Could you please  
rephrase that?
```

At this point, UC does not know how to print a file. Now, an expert can enter UCTeacher and provide the necessary information, as shown in the following interactions:

```
% UCTEACHER  
UCT> There is a new command.  
What is its name?  
UCT> lpr  
What is it used for?  
UCT> It is used to print les.  
Where does the output go?  
UCT> the lineprinter  
You mean that typing 'lpr lename' will print the le  
lename on the lineprinter.  
UCT> Yes.
```

The system can now give a correct response to the user's query.

```
# How can I print a le?  
Use lpr.  
For example, to print the le named foo, type 'lpr foo'.
```

Here ALANA represents the content of the initial utterance as an instance of the quite general concept EXECUTE-UNIX-COMMAND. UCTeacher then collects from UC's knowledge base all the pertinent relations that EXECUTE-UNIX-COMMAND participates in. These include the format aspectual of EXECUTE-UNIX-COMMAND, the plan aspectual of a PLANFOR that EXECUTE-UNIX-COMMAND constrains, and the name relation that of UNIX-COMMAND, which constrains the command aspectual of EXECUTE-UNIX-COMMAND. Figure 29 illustrates the knowledge in which these concepts are embedded.

The relations needed to complete a EXECUTE-UNIX-COMMAND do not appear to conform to any obvious pattern. For example, the fact that a command has a name is several links away, and the fact that the command should have a purpose is represented by its participation in a PLANFOR. The difficulty of determining the appropriate relations and concepts needed to specialize or instantiate a concept in the network has led to the development of the notion of a *Minimal Aspectual Set* (MAS). The MAS constitutes

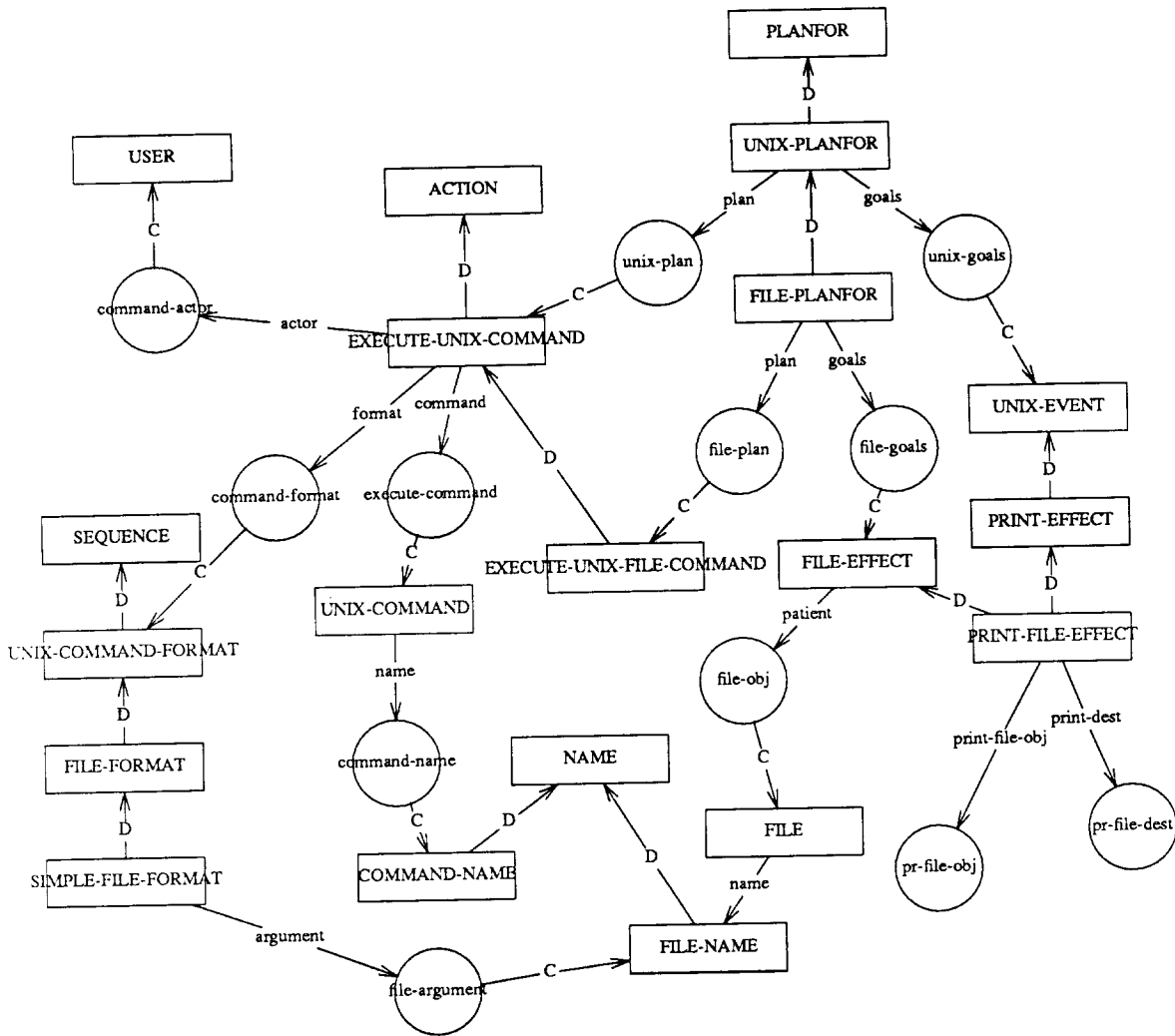


Figure 29
Knowledge for learning about commands

the minimal set of concepts that must be present for a concept to be considered complete by UCTeacher. UCTeacher uses the MAS for the concept being acquired to limit its search through the knowledge base. A relation not directly connected to the main concept being acquired is only followed if it connects to a concept that is present in the MAS.

For example, the MAS of EXECUTE-UNIX-COMMAND includes the command-name aspectual of UNIX-COMMAND, the unix-goals aspectual of UNIX-PLANFOR, and the command-format aspectual of EXECUTE-UNIX-COMMAND. From this description, UCTeacher can construct all the necessary components to specialize EXECUTE-UNIX-COMMAND.

The system's first question in this example is an attempt to find a value for the relation command-name. A piece of meta-knowledge used here, namely, that the only way to fill in the value of a name relation is to ask the user, as it cannot be inferred from anything else about the command. In the case where asking the user is the method, a pattern that is appropriate is included as part of the meta-knowledge of that concept. Here, the pattern gives rise to the question, and the answer is connected to the representation. Note that this requires the creation of a number of links and nodes, including an execute-command link to an instance of UNIX-COMMAND and a command-name link to the actual name.

Whenever UCTeacher adds a new fact to the representation it is creating, it calls the concretion mechanism to see if it can classify the node to something more specific. If so, it may be able to use the more specific category to determine what else it needs to learn. In this case, the concretion mechanism is called, but it cannot concrete any further.

The second question from UCTeacher is an attempt to instantiate the unix-goals relation of the UNIX-PLANFOR. Again, the user is queried, and the response represented as a PRINT-FILE-EFFECT. A specialization of UNIX-PLANFOR is constructed, and a unix-plan link is made to EXECUTE-UNIX-COMMAND, and a unix-goals link to PRINT-FILE-EFFECT. Also, PRINT-FILE-EFFECT requires a destination to be complete. This leads to the attempt to fill in the print-dest relation by asking the last question.

Again, a concretion is attempted. Here the concretion is successful; we are able to classify the concept being build as a EXECUTE-FILE-COMMAND fact, rather than just a EXECUTE-UNIX-COMMAND fact. At this point, UCTeacher searches for any new relations that have been inherited because of the concretion and adds them to the list of relations to learn. It also recomputes the MAS that it is using, based on any additions from the concretion.

The next relation that gets examined is command-format. Note that UCTeacher does not query the expert for the argument format. This is because meta-knowledge about the format of UNIX file commands specifies that the default format for such commands is a function of the files involved; in this case, the default is SIMPLE-FILE-FORMAT.

Believing its knowledge acquisition process to be complete, UCTeacher now generates a statement reflecting what it has just learned. This gives the expert the opportunity to correct any incorrect inferences made by the system. For example, if this command did not take the usual argument format, the user would have an opportunity to correct it.

The final phase of UCTeacher's processing is to make the changes permanent. Each absolute and relation that has been instantiated is converted to a linear tuple form and printed to a file containing UC's knowledge base. In the current version of UC, the linguistic knowledge used by the parser and generator are kept in two, separate, non-KODIAK forms. This poses a problem for UCTeacher, which requires a rich knowledge-representation system to function properly. As a temporary solution, templates are attached to concepts that have linguistic counterparts (such as command names); these templates are instantiated and written out to the appropriate files during the final phase of processing. This will be necessary until all linguistic knowledge is represented in KODIAK.

10.4. Extending Metaphoric Word Sense

Much of the vocabulary with which user's talk about computer systems is taken from other domains. For example, consider utterances:

You can *get into* the editor by typing emacs to the shell.
How can I *exit* lisp?
To *leave* the mail system, type 'exit'.
You can *kill* a process by typing ^C.
My editor just *died*, how can i save the buffers?
Run your file *through* spell to *fix* the spelling.

These examples illustrate two facts: First, the ordinary language we use to talk about computation is rife with metaphor. Second, the metaphors are systematic, as is argued at length in Lakoff & Johnson 1980. For example, underlying the utterances "kill a process" and "a process died" is the understanding that a process can be thought of as a living thing, and the termination of the process as that thing's death. Similarly, an interactive process is thought of as a kind of region or container, which one may enter, exit, leave, get into, etc.

In addition, the metaphoric use of these terms is not confined to the computer domain. For example, not only can one kill a process, but also, a light, a six-pack of beer, a conversation, and a car engine, although each of these has a different interpretation. Thus, if we view these usages extensions of non-metaphoric terms, then the kinds of metaphoric extensions we have for the computer domain are related to those outside this domain. For example, "kill a process" means terminating that process, while "kill an engine" means terminating the process in which the engine is engaged.

We have attempted to use this fact to aid in knowledge acquisition. The strategy is as follows: First, represent the central senses of a word, and some of the common metaphoric extensions of that term. Then, when such a term is encountered in a particular application, such as UC, in a context in which it cannot be understood, an attempt is made to compute a metaphoric extension. This is done by finding the closest metaphoric extension from another domain. This sense of the term is then modified to produce a sense applicable to the current context.

In its current incarnation, UCTeacher can perform the following functions: Given that UC has the requisite knowledge to directly understand the utterances containing the construction "enter lisp", UCTeacher can compute the likely intended meaning of utterances involving "exit lisp" and "enter mail"; given that UC knows about "kill a conversation", it can generalize to "kill a process"; from "kill a process", it can generalize to "process died"; from "Open a file" to "close a file", and finally, outside the UNIX domain, from "give a cold" to "have a cold", "get a cold", "give the flu", and "give an idea".

To perform these functions, UCTeacher needs above all a representation for metaphoric structures. The basis of such a representation is given in KODIAK by a device called a *view*. A view allows a kind of structure mapping between two concepts. Views are described in Wilensky 1986, but have been significantly extended by Martin (Martin

1986a, Martin 1986b, Martin 1987 and Martin 1988). A detailed description of this representation, and of UCTeacher's algorithm for extending word senses, is beyond the scope of this paper, but may be found in above references.

11. 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 87) 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*.

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

*This "feature" has recently been changed to produce an error message on some versions of UNIX.

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.

12. References

- Allen, James F.; Frisch, Alan M. and Litman, Diane J. 1982. ARGOT: the Rochester dialogue system. In *Proceedings of the National Conference on Artificial Intelligence*. Pittsburgh, PA.
- Allen, James F. and Perrault, C. Raymond. 1980. Analyzing intention in utterances. *Artificial Intelligence* 15: 143-178.
- Austin, John L. 1962. *How To Do Things With Words*. Harvard University Press, Cambridge, MA.
- Brachman, Ronald and Schmolze, J. 1985 An overview of the KL-ONE knowledge representation system. In *Cognitive Science* 9: 171-216.
- Brown, John S. and Burton, Richard R. 1976. A tutoring and student modelling paradigm for gaming environments. In the *Symposium on Computer Science and Education*, Anaheim, CA: 236-246.
- Carberry, Sandra. 1983. Tracking user goals in an information-seeking environment. In *Proceedings of the National Conference on Artificial Intelligence*. Washington, DC.
- Chin, David N. 1986. User modeling in UC, the UNIX consultant. In *Proceedings of the CHI-86 Conference*. Boston, MA.
- Chin, David N. 1988. Intelligent Agents and a Basis for Interactive Consultation Systems. Berkeley Computer Science Technical Report UCB/CSD 88/396.
- Cohen, Philip R. & Levesque, Hector J. 1987. Persistence, Intention, and Commitment. SRI International Technical Report 415.
- Cox, Charles A. 1986. *ALANA: Augmentable LAnguage Analyzer*. Computer Science Division, University of California, Berkeley, Report No. UCB/CSD 86/283.
- Davis, Randall. 1976. *Applications of Meta Level Knowledge to the Construction, Maintenance and Use of Large Knowledge Bases*. Stanford CS Report, STAN-CS-76-552, Stanford, CA.
- Deering, Michael; Faletti, Joseph and Wilensky, Robert. 1982 *Using the PEARL AI Package*. Computer Science Division, University of California, Berkeley, Memorandum No. UCB/ERL M82/19.

- Ernst, G. and Newell, Allen. 1969. *GPS: A Case Study in Generality and Problem Solving*. Academic Press, New York.
- Fikes, Richard E., and Nilsson, Nils J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(3-4): 189-208.
- Grice, H. Paul. 1975. Logic and conversation. In Cole, P. and Morgan, J. L., Eds., *Studies in Syntax*, vol. III. Seminar Press: 41-58.
- Grosz, Barbara and Sidner, Candace L. 1985. *The Structures of Discourse Structure*. Center for the Study of Language and Information Report No. CSLI-85-39.
- Haas, N. and Hendrix, Gary. 1980. An approach to acquiring and applying knowledge. In *Proceedings of the National Conference on Artificial Intelligence*, Stanford, CA: 235-239.
- Jacobs, Paul S. 1984. *PHRED: A Generator for Natural Language Interfaces*. Computer Science Division, University of California, Berkeley, Report No. UCB/CSD 84/189.
- Jacobs, Paul S. 1985. *A Knowledge-Based Approach to Language Production*. Ph.D. thesis, University of California, Berkeley.
- Kaplan, S. Jerrold. 1983. Cooperative Responses from a Portable Natural Language Database Query System. In Brady, Michael and Berwick, Robert C., Eds., *Computational Models of Discourse*. MIT Press, Cambridge, MA.
- Lakoff, George. & Johnson, Mark. 1980. *Metaphors We Live By*. University of Chicago Press, Chicago, IL.
- Litman, Diane J. and Allen, James F. 1984. A plan recognition model for clarification subdialogues. In *Proceedings of the Tenth International Conference on Computational Linguistics*, Palo Alto.
- Luria, Marc. 1982. Dividing up the question answering process. In the *Proceedings of National Conference on Artificial Intelligence*, Pittsburgh, Pennsylvania: 71-74.
- Luria, Marc. 1985. Commonsense planning in a consultant system. In *Proceedings of the 1985 IEEE International Conference on Systems, Man, and Cybernetics*. Tucson, Arizona: 602-606.
- Luria, Marc. 1987. Goal Conflict Concerns. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*. Milan, Italy.

- Luria, Marc. 1988. Knowledge Intensive Planning. Berkeley Computer Science Technical Report No. UCB/CSD88/433.
- Martin, James. 1986a. Views from a kill. In *Proceedings of the 8th National Conference of the Cognitive Science Society*. Amherst, MA: 728-733.
- Martin, James. 1986b. Representing and acquiring knowledge about metaphors. In *Proceedings of the 3rd Workshop on Theoretical Issues in Conceptual Information Processing*. Philadelphia, PA: 49-54.
- Martin, James. 1987. Understanding new metaphors. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, Milan, Italy.
- Martin, James. 1988. Knowledge acquisition: Understanding new metaphors. Berkeley technical report (forthcoming).
- Mays, E. 1980. Failures in Natural Language Systems: Applications to Data Base Query Systems. In *Proceedings of 1980 National Conference on Artificial Intelligence*, Stanford, CA.
- McCoy, K. F. 1983. Correcting Misconceptions: What to say when the user is mistaken. In *Proceedings of the CHI'83 Conference*, Boston, MA: 197-201.
- Morik, K. and Rollinger, C-R. 1985. The real estate agent -- modeling the user by uncertain reasoning. In *AI Magazine* 6(2): 44-52.
- Newell, Allen, and Simon, Herbert A., 1972. *Human Problem Solving*. Prentice-Hall, Englewood Cliffs, N. J.
- Norvig, Peter. 1983. Frame Activated Inferences in a Story Understanding Program. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, Karlsruhe, West Germany: 624-626.
- Norvig, Peter. 1987. A Unified Theory of Inference for Text Understanding. UC Berkeley Computer Science Report No. UCB/CSD/87/339.
- Rich, Elaine. 1979. User modeling via stereotypes. In *Cognitive Science* 3: 329-354.
- Rissland, Edwina L. 1983. Examples in Legal Reasoning: Legal Hypotheticals. In *Proceedings of the Eight International Joint Conference on Artificial Intelligence*, Karlsruhe, West Germany: 90-93.

- Rissland, Edwina L.; Valcarce, E. M. and Ashley, Kevin D. 1984. Explaining and Arguing with Examples. In *Proceedings of the National Conference on Artificial Intelligence*, Austin, Texas: 288-294.
- Rosch, Eleanor. 1978. Principles of categorization. In Eleanor Rosch and B. B. Lloyd, *Cognition and Categorization*, Lawrence Erlbaum. Hillsdale, N. J.
- Sacerdoti, Earl, 1974. Planning in a hierarchy of abstraction spaces. In *Artificial Intelligence* 5: 115-135.
- Schank, Roger C., 1975. *Conceptual Information Processing*. North Holland, Amsterdam.
- Schmolze, J. G. and Lipkis, T. A. 1983. Classification in the KL-ONE knowledge representation system. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*. Karlsruhe, West Germany.
- Schubert, Lenhart K. 1976. Extending the expressive power of semantic networks. *Artificial Intelligence* 7: 163-195.
- Searle, John R. 1969. *Speech Acts; An Essay in the Philosophy of Language*. Cambridge University Press, Cambridge, England.
- Searle, John R. 1979. Literal Meaning. In Searle, John R., Ed., *Expression and Meaning*. Cambridge University Press, Cambridge, England.
- Shapiro, Stuart. 1979. The SNePS Semantic Network Processing System. In N. Findler, Ed., *Associative Networks*. Academic Press, New York.
- Sidner, Candace L. 1985. Plan parsing for intended response recognition in discourse. *Computational Intelligence* 1: 1-10.
- Teitelman, Warren, 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*. Heuristic Programming Project Report No. HPP-80-22, Computer Science Department, Stanford University, CA.
- Webber, Bonnie L. and Mays, E. 1983. Varieties of User Misconceptions: Detection and Correction. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, Karlsruhe, West Germany: 650-652.

Wilensky, Robert. 1983. *Planning and Understanding: A Computational Approach to Human Reasoning*. Addison-Wesley, Reading, Mass.

Wilensky, Robert. 1986. *Some Problems and Proposals for Knowledge Representation*. Computer Science Division, University of California, Berkeley, Report No. UCB/CSD 86/294.

Wilensky, Robert. 1987. *Primal Content and Actual Content: An Antidote to Literal Meaning*. Computer Science Division, University of California, Berkeley, Report No. UCB/CSD 87/365.

Wilensky, Robert and Arens, Yigal. 1980. *A Knowledge-based Approach to Natural Language Processing*. University of California, Berkeley, Electronic Research Laboratory Memorandum No. UCB/ERL/M80/34.

Wilensky, Robert; Arens, Yigal and Chin, David. 1984. Talking to Unix in English: an overview of UC. *Communications of the Association for Computing Machinery* 27(6).