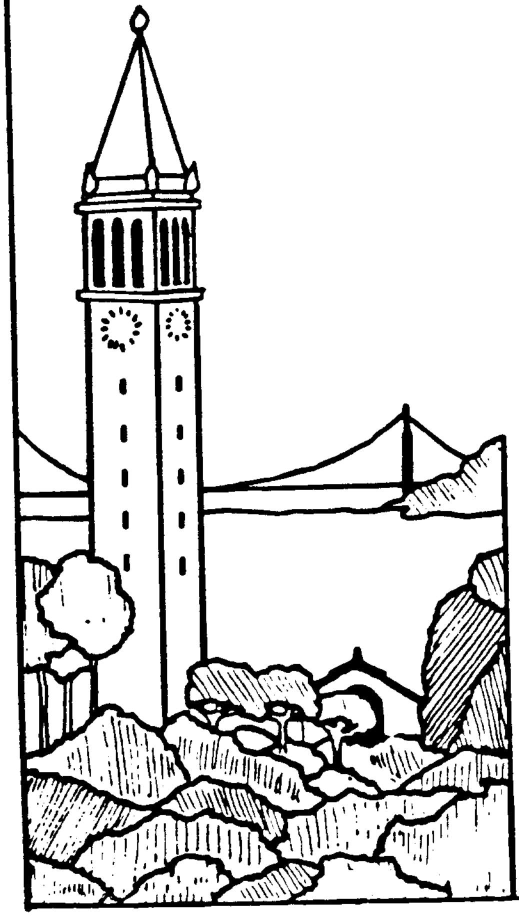
UC - A Progress Report

Robert Wilensky, James Mayfield, Anthony Albert,
David Chin, Charles Cox, Marc Luria,
James Martin and *Dekai Wu*



Report No. UCB/CSD 87/303

July 1986

Computer Science Division (EECS)
University of California
Berkeley, California 94720

UC – A Progress Report*

Robert Wilensky
James Mayfield
Anthony Albert
David N. Chin
Charles Cox
Marc Luria
James Martin
Dekai Wu

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

ABSTRACT

UC 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 UCPlanner, 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 builds a model of the user that represents UC's assessment 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. 3041, monitored by the Naval Electronic Systems Command under contract N00039-82-C-0235, by the Office of Naval Research under contract N00014-80-C-0732, and by the National Science Foundation under grant MCS79-06543.

†UNIX is trademark of Bell Laboratories

Acknowledgements

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. 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

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 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 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 requires 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 his 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. 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 process a request by the user correctly 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.

We did not feel that it was necessary or appropriate to produce a product that could actually be used in a real-world setting in order for our system to be considered a success. However, we did feel we should show that one could develop such a system along the

†UNIX is trademark of Bell Laboratories

lines our research suggested. This would be accomplished by developing an extendible prototype.

1.1. UC Old and New

We initially built a prototype version of UC consisting largely of "off the shelf" components [Wilensky, Arens, and 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 that 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, and Wilensky 1981]. **PEARL** is an AI language and data base 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 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.

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 mechanism of each component is described. Finally, we conclude with some discussion of the deficiencies of our current design.

1.2. The Structure of UC

The current version of UC involves a number of components that are invoked in a more or less serial fashion. Each of these is now briefly described:

(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.

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.

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 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.

(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 ''cat is on the mat'' example 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 sentences 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 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**.

(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** (Plan And Goal ANalyzer), 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 the UNIX Consultant, **UC** must now decide what its own goals should be. Generally, we would expect a system like **UC** to do what the user requested. But this is not always appropriate. For example, if the user asked how to crash the system, it would be inappropriate for a consultant to give the user the superuser password in order to help. This is inappropriate because a consultant probably has other goals, such as maintaining the integrity of the system.

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 adopt the goal of helping the user by finding out what the user wants to know, and then telling it to the user. As the example above illustrates, **UCEgo** must also detect conflicts between such goals and other goals it may have. As another example of such an interaction, **UC** also attempts to be educational. Thus, if the user asks **UC** to actually perform some request, such as telling the user who is on the system, **UC** should decide to tell the user *how* to perform such a function himself, rather than do what the user requested. **UC** needs to have some notion of its own goals to decide how best to perform some action other than what the user requested.

UCEgo is one important way in which UC differs from systems designed to be interfaces. While interfaces are generally thought of as passive conduits through which information flows, UC is best thought of as an agent. The agent listens to the user, and is generally helpful. But the agent has its own agenda, and the requests of the user are merely a source of input to the agent.

(5) User Model

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 he is testing the consultant's knowledge.

Assessing the knowledge state of the user is the function of the user model. The user model is built up by UCEgo, primarily because they were designed by the same individual. It is exploited by several components, including the Expression Mechanism described below.

(6) Domain Planner (UCPlanner)

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

(7) Expression Mechanism (UCExpress)

Having gotten UCPlanner 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 UCPlanner that, for some reason, it appears 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.

The result of UCExpress is an annotated KODIAK network, where the annotation specifies which part of the network 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; **UCPlanner** is given the goal of solving some underlying UNIX problem. Thus **PAGAN** is a goal analyzer, that is, a program that attempts to determine someone else's goals, while **UCEgo** and **UCPlanner** are planners, i. e., programs that determine and produce plans for their own goals.

UCEgo and **UCPlanner** 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 **UCPlanner**. Such an integration will have to await a revision of our system.

1.3. UC Examples

The following interactions illustrate the scope of the current UC prototype. They by no means exhaust the capabilities of the system. Instead, they are intended to show the kind of requests the system can handle, as well as its scope.

Each sentence below preceded by the prompt "User:" is supplied by the user. The rest of the text is a computer generated response.

```
% UC
Welcome to UC (Unix Consultant) version 2.7
To a UC '#' prompt, please type in your questions about Unix in English.
To leave, just type a '^D' or '(exit)'.

User: Can you tell me how to move a file?
Use mv.
For example, to move the file named foo to the file named fool,
type 'mv foo fool'.

User: Do you know how to delete a file named dekai?
Type 'rm dekai'.
```

```
User: Do you know how to delete a file?
Use rm.
For example, to delete the file named foo, type 'rm foo'.

User: How can I delete the file charlie?
Type 'rm charlie'.

User: How can I delete the file named tony?
Type 'rm tony'.

User: How can I get rid of the file named boob?
Type 'rm boob'.

User: I want to delete a file named peter.
Type 'rm peter'.

User: I want to delete a file.
Use rm.
For example, to delete the file named foo, type 'rm foo'.

User: I want to delete the file dave.
Type 'rm dave'.

User: 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'.

User: 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'.

User: How can I change the group execute permission of a file?
Use chmod.
For example, to add group execute permission to the file named
foo, type 'chmod g+x foo'.
For example, to remove group execute permission from the file
named foo, type 'chmod g-x foo'.

User: How can I change the group read permission of a file?
Use chmod.
For example, to remove individual read permission from the file
named foo, type 'chmod u-r foo'.

User: How can I remove the group write permission from the file named bimbo?
Type 'chmod g-w bimbo'.
```

```
User: 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'.

User: How can I compact a file?
Use compact.
For example, to compact the file named foo, type 'compact foo'.

User: How can I create a directory?
Use mkdir.
For example, to create the directory named foo, type 'mkdir foo'.

User: How can I find a file?
Use whereis.
For example, to find the file named foo, type 'whereis foo'.

User: How can I find out how much disk space I am using?
Type 'du'.

User: How can I find out who is using the system?
Type 'finger'.

User: How can I find out wilensky's phone number?
Type 'finger wilensky'.

User: How can I list a directory?
Use ls.
For example, to list the directory named foo, type 'ls foo'.

User: How can I logout?
Type 'logout'.

User: Tell me how to leave.
Type 'logout'.

User: 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'.

User: 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'.

User: 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'.

User: How do I change my directory?
Use cd.
For example, to connect to the directory named foo, type 'cd foo'.

User: 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'.

User: How do I copy the file foo to the file blah?
Type 'cp foo blah'.

User: How do I create a directory?
Use mkdir.
For example, to create the directory named foo, type 'mkdir foo'.

User: How do I print a file named info?
Type 'lpr info'.

User: I want to print a file.
Use lpr.
For example, to print the file named foo, type 'lpr foo'.

User: I want to change my working directory.
Use cd.
For example, to connect to the directory named foo, type 'cd foo'.

User: 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'.

User: I want to delete a directory.
Use rmdir.
For example, to delete the directory named foo, type 'rmdir foo'.

User: I want to remove a directory.
Use rmdir.
For example, to delete the directory named foo, type 'rmdir foo'.

User: What's the best way to delete a directory?
Use rmdir.
For example, to delete the directory named foo, type 'rmdir foo'.
```

```
User: I want to display a file.
Use more.
For example, to display the file named foo, type 'more foo'.

User: I want to edit a file.
Use vi.
For example, to edit the file named foo, type 'vi foo'.

User: I want to find out Joe's address.
Type 'finger joe'.

User: I want to remove a directory named junk.
Type 'rmdir junk'.

User: What's the command to compact a file?
Use compact.
For example, to compact the file named foo, type 'compact foo'.
```

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) was developed to address certain weaknesses of other representational systems. In particular, an attempt was made to make the interpretation of the objects in one's representation a bit clearer. To do this, we found it necessary to make KODIAK relation-oriented rather than object-oriented, and to eliminate the frame/slot or node/link distinction found in frame-based and semantic network-based systems, respectively. In their place is introduced a new distinction, called the *aspectual/absolute* distinction.

To motivate this distinction, consider how facts are stored in frame-based systems. Typically, a frame for "person," say, will have a slot for name, one for age, and one for address. Similarly, a frame for "physical-object" may have a slot for color, size, weight, etc. However, what these slots *mean* is not represented. That is, there is nothing in the representation system that captures the fact that age, for example, is the amount of time between the creation of something and some reference point in time, or that objects have colors in a different way than people have addresses. Moreover, there appears to be no principle to determine which slots belong to which frames. Similar arguments can be made for the nodes and links of semantic network systems.

In KODIAK, rather than start with objects with slots, we begin with relations. For example, we might posit a "dwelling-has-address" relation. This relation would take two arguments, which we might call a "dwelling-with-address" and the "address-of-dwelling." That is, we might say that the idea of a place with an address involves a particular kind of relation between something playing the role of a dwelling and something acting as an address. We call concepts like "dwelling-with-address" and "address-of-dwelling" *aspectuals,* since they are meaningful only in relation to some other idea (in this case, to the relation "dwelling-has-address").

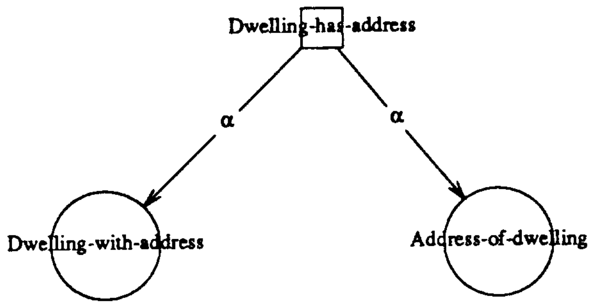
To make these concepts meaningful, we need to connect them to other concepts. For example, we need to specify that, in general, whatever is in the "dwelling-with-address" role must be some sort of location. Similarly, we need to specify that objects in an "address-of-dwelling" position should be "address"-type objects. These objects in turn have a certain structure, for example, one type of address has a state, city and street address, etc.

Moreover, we might introduce another relation, one we could call "person-dwells-at-residence." This might hold between a "dweller" and a "dwelling." Thus, in KODIAK, saying that something is the address of a person would be represented as something equivalent to "some address object plays the 'address-of-dwelling' role to some location playing the 'dwelling-with-address' role. This same location plays the 'dwelling' role to some person's 'dweller' role."

As another example, consider the idea of being a part of something. It seems there is a concept "being a part," but it is only meaningful with respect to something it can be a part of (i. e., something just can't be a part, period). In KODIAK, we would say that "part" and "whole" are aspectuals of a "part-whole" relation. In most other languages, one says that some object has a slot for some kind of part, for example, that an engine has a part slot for crankshaft. In KODIAK, we would say that a part-whole relation holds between the objects, that is, that the general concept "engine" and "crankshaft" were in a part-whole relation.

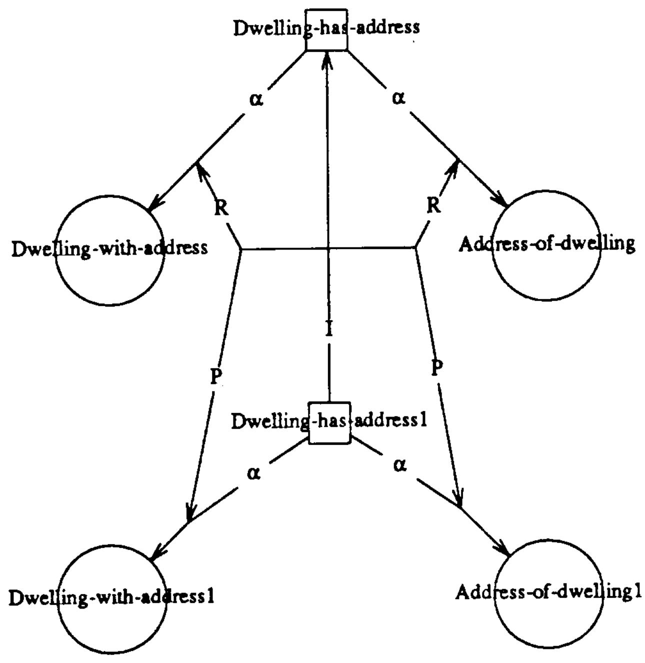
Concepts that seem to be meaningful without respect to some other concept are called *absolutes*. Relations themselves are absolutes, as are most objects.

We diagram relations and their aspectuals using the following notation:

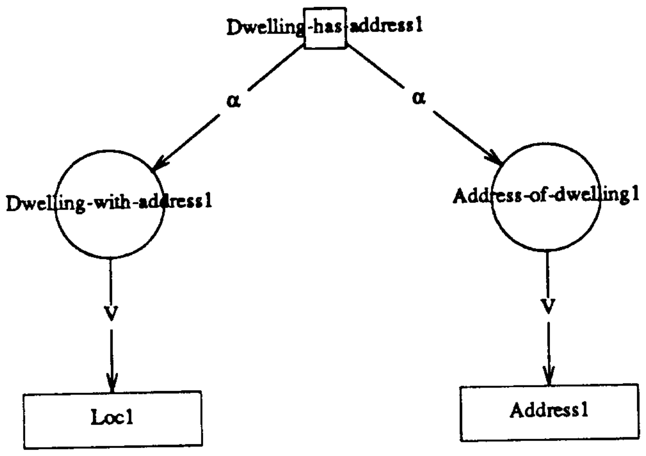


In these diagrams, absolutes, including relations, are represented as boxes, and aspectuals as circles. The labeled arrows represent various **KODIAK** *epistemological* relations, that is, relations known to the **KODIAK** interpreter. In this example, the links labeled α indicate the aspectuals of a relation. We sometimes say that an aspectual *manifests* its aspectuals.

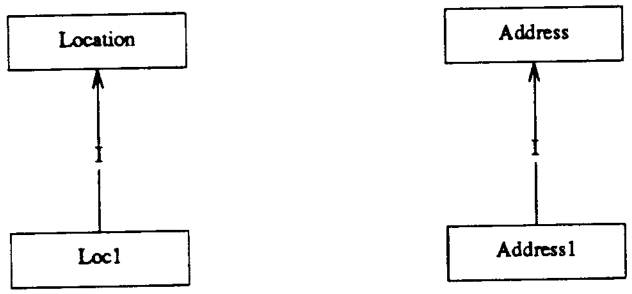
We represent the fact that some particular dwelling has a particular address by creating a new relation with new aspectuals. This new relation represents an individual fact, and is connected to the generic relation using an **Instantiate** (I) and **role/play** (R/P) links. For example, we could represent such a fact as follows:



To include in our representation the particular objects in this relation, we connect them to the aspectuals of the particular fact using **Value** (V). Thus we might elaborate the above to include the following:



Furthermore, **Loc1** and **Address1** would be represented as instances of the appropriate object categories:



More details would be included using additional relations. For example, **Address1** might be in a **City-of-Address1** relation to **Berkeley**, which would be an instance of a city.

Generally, a relation in **KODIAK** will be a subtype of another relation. This is represented using a **Dominate (D)** link. We differentiate the aspectuals of such relations by constraining them, using a **Constrain (C)** link. This means that an individual version of the relation must have its aspectuals played by aspectuals whose values conform to the constraint.

For example, in **KODIAK** we posit a **Causal-Event** relation that holds between the aspectuals **Cause** and **Effect**. The idea of killing is represented as a kind of causing in which the thing caused is a kind of death event. This is depicted in Figure 1. This example uses an abbreviated notation, which we will use throughout this paper. Once we have defined a relation with aspectuals, we can represent role-play links by labeling a link with an aspectual name. For example, in Figure 1, the link labeled **Effect** that connects **Kill-effect** with **Kill** is just an abbreviation for a role-play link. It should be interpreted as saying that (1) **Kill-effect** is an aspectual of **Kill**, and (2) **Kill-effect** plays the role of **Effect** with respect to the dominating **Causal-event** concept.

Figure 1 should be interpreted as follows: A killing (i. e., **Kill**) is a kind of causing (i. e., **Causal-event**) in which the thing caused (the **Effect**) is constrainted to be a kind of death event. This particular death event (namely, **Kill-death-event**) is not distinguished from the general death event (i. e., **Death-event**) except that it is a death resulting from killing. A death event is a kind of state change (**State-change**) in which a person (the

dier) goes from an initial state of being alive (represented by the statement **Dier-is-alive**) to the final state of being dead (represented by **Dier-is-dead**).

This figure also makes use of another KODIAK relation, called **EQUATE** (=). This is really a bidirectional link between two aspectuals, meaning that instances of the concepts manifesting these aspectuals must have corresponding aspectuals that have the same value. In Figure 1, for example, **EQUATE** links are used to state that the fellow who started out alive and the fellow who ended up dead are the same fellow, namely, the dier of the death event.

KODIAK has many additional features that are not described herein. The reader is referred to Wilensky [1986] for a more complete description.

In the following sections, we briefly describe each component of UC. Each component is illustrated on the processing of the sentence ''Do you know how to print a file on the imagen*?'' In some cases, a module may be capable of doing a great deal more than is required for this example. However, the example is useful for illustrating the kind of processing that is performed for a typical request.

*A kind of laser printer used at our site.

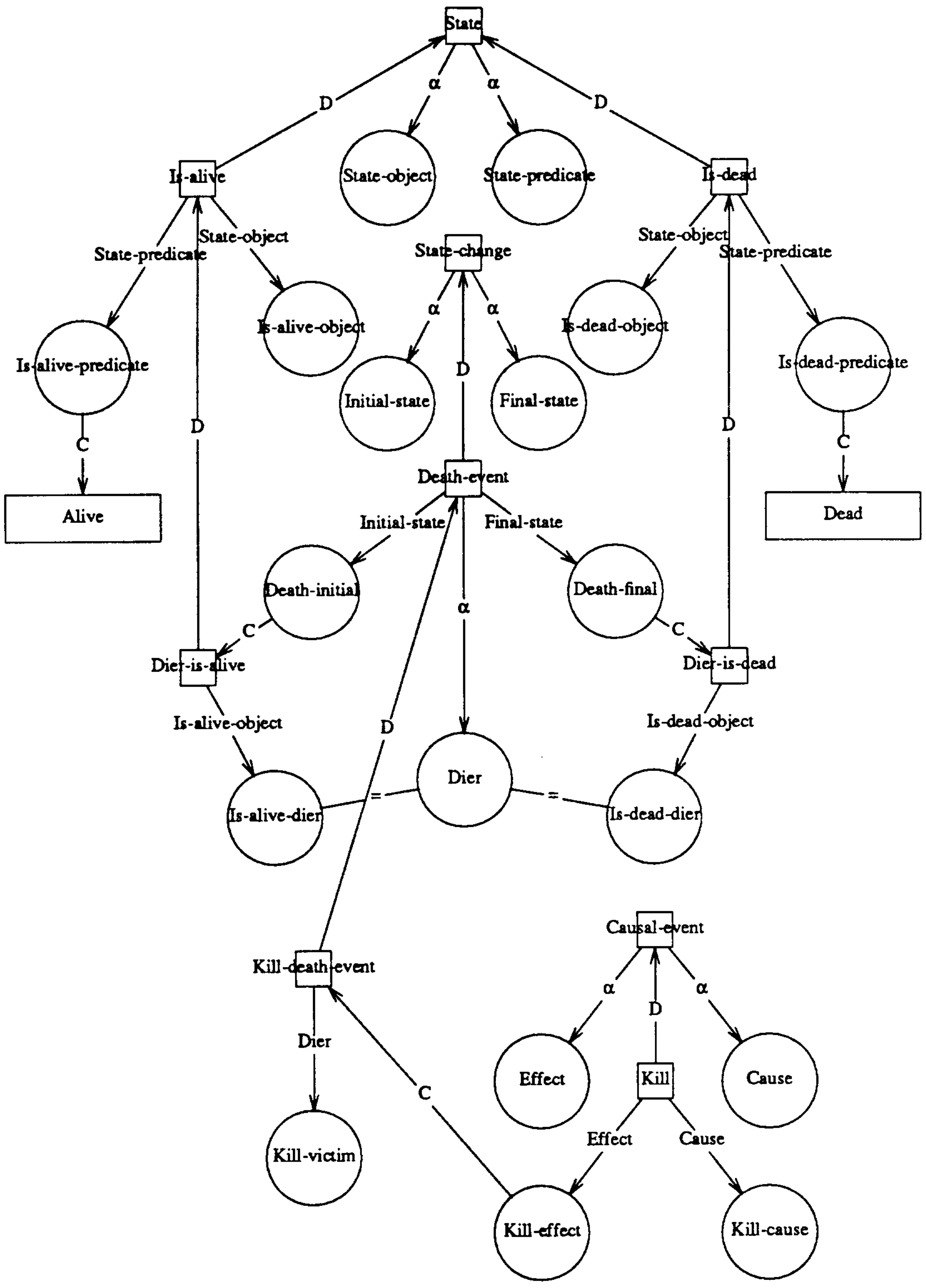


Figure 1
KODIAK Representation of Killing

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 and Arens 1980], the front-end natural language component for the original UC [Wilensky, Arens and 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.

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. More detailed criticisms of PHRAN's implementation and how they are addressed in ALANA can be found in Cox [1986].

3.1. ALANA's Knowledge Representation

First, we will look at what ALANA's knowledge looks like. Then we will examine in detail how the actual analysis is done and how representations get built.

The main idea behind ALANA is that pattern knowledge is used to aid in the syntactic analysis, which takes place at the same time as semantic analysis. ALANA uses these patterns to suggest parses.

All pattern matching knowledge, from the individual word up to syntactic rules, are handled as Pattern-Concept-Test triples. The triple consists of: A *pattern*, which is matched against a parse chart, an optional *test*, which is called once the pattern is matched to determine if the associated *concept* should be instantiated and asserted, and the *concept* itself, which is a declarative representation of the meaning of the phrase that makes use of the patterns that have already been matched.

For example, consider some of the patterns used to analyze the question "Do you know how to print a file on the imagen?" These include <Aux> <NP> <VP>, whose associated concept denotes a question, and <Print> <NP$_1$> <on> <NP$_2$>, whose concept denotes a printing event in which the destination of the object to be printed (NP$_1$) is NP$_2$.

Tests are used to check for such things as subject-verb agreement and nominal case.

The *concept* part of the triple is written in a notation that describes what KODIAK structures should be built and how they should be connected. For example, the concept associated with the pattern <print> is

```
(builds (i PRINT-ACTION
   witha pr-effect (i PRINT-EFFECT))))
```

The form (i PRINT-ACTION ...) specifies the creation a new instance of PRINT-ACTION (a **KODIAK** relation defined in the general UC knowledge base). Should this pattern be chosen, a new relation, say PRINT-ACTION3, would be created and made an instance of PRINT-ACTION. The 'witha aspectual$_i$ value$_i$' inside the *i* expression specifies the creation of new instance aspectuals for this new relation and calls for *role-play* links to the corresponding definitional aspectuals (i.e., those aspectuals of the relation of which this new relation is an instance). Similarly, the (i PRINT-EFFECT) calls for the creation of a new instance of PRINT-EFFECT.

A more complicated pattern, <Print> <NP> is associated with the concept

```
(builds
 (i HAS-PRINT-OBJECT
   witha pr-obj-obj (get-value-of pr-effect (concept-of Print))
   witha pr-object (concept-of NP))
 (concept-of Print)))
```

The forms *(concept-of NP)* or *(concept-of Print)* refer to pointers to the **KODIAK** structure that were built and/or returned at the time the *Print* or *NP* patterns were being instantiated. This is how previously built concepts are linked to those currently being built. The effect of the above *builds* expression can be seen as part of the **KODIAK** representation in Figure 2.

3.2. ALANA's Processing

ALANA processes a sentence one word at a time, left to right. As it processes each word, it simultaneously builds a chart representing the syntactic information as well as a **KODIAK** network representing the meaning of the input utterance. Thus inferences are being formed and rejected before clauses end. It is this idea of integrating grammatical processing with other semantic and reasoning processing that gives **ALANA** efficiency and speed (in understanding) over syntax-first parsers.

When a pattern has been selected by **ALANA**, the anticipation of the next word of the linguistic component of that pattern serves as an expectation. For example, seeing a Wh-word such as How, **ALANA** anticipates a question, because there is a question pattern whose first word is How. To completely integrate such analysis at "question-read" time, **ALANA** should ideally run as a coroutine. If so, during the reading, the other components could signal to **ALANA** that a hypothesis or a pattern should be rejected or that another hypothesis should be considered. Although **ALANA** is capable of this sort of hypothesis rejection, the functionality is currently unused in UC. The current implementation of UC is organized as a pipelined process with **ALANA** acting as the front end.

3.2.1. Chart Parsing

The pattern matching method used in ALANA is designed to match simultaneously all levels of patterns, ranging from individual words and morphemes (stored as patterns) to the more abstract linguistic constructions such as questions of the form <Wh-word> <Auxiliary> <Sentence>. To allow simultaneous parallel parses of the input sentence, the pattern-matcher keeps the matched patterns organized as a chart. The chart ALANA uses represents all possible parses of the input along with all possible parses of substrings.

3.2.2. Pattern Storage

As with PHRAN, patterns in ALANA are stored in a discrimination net. The discrimination net is used to store the concepts and tests with the pattern components acting as an index to them. As the matcher matches a pattern, it follows the path in the discrimination net by matching each pattern element with a link in the net. When it finds that a particular discrimination net node has information attached, it either builds an instance of the associated concept, calls a test for feasibility, sets a global variable, or some combination of the above.

3.2.3. Open and Closed Patterns

To keep pattern matching as general as possible, we need a way to match components no matter where they begin or end in an input sentence. The structure of the parsing chart itself gives us the ability to point to any position in the input. Combined with the ability to point anywhere in the pattern discrimination net, we are able to have a special data structure called an *open-pattern*, which ties the discrimination net to the parsing chart. Such a data structure keeps track of where a pattern match began in the input, where it ends, and the matchings of edges to pattern component labels.

3.2.4. Spelling Corrector

Any time a new word is entered, we expect at least one open-pattern to advance. If none can advance, we have an unknown or misspelled word. ALANA does allow wildcards in its patterns, which expect unknown or arbitrary words at certain points; open-patterns containing such wildcards in their patterns would automatically advance. Thus if no open pattern can advance, we truly have an unknown word at a place we did not expect to find one. Such an unknown word is likely to be misspelled. If the correct spelling could be known, we could insert the correct word into the chart and try to match against it.

ALANA makes use of a spelling corrector implemented by James Mayfield, based on the algorithm used in DWIM [Teitelman 1978]. When ALANA is stopped at a word where no open-pattern can advance, it calls the spelling corrector with the unknown word and the list of expected words, drawn from the patterns that have already started matching. The corrector then returns a list of valid candidates from the list of possibilities. All the possibilities are entered into the chart at the point where the misspelled word is located.

3.2.5. Example

We now present a simplified trace of **ALANA**'s actions while reading "Do you know how to print a file on the imagen?"

Read `do you`

> **ALANA** recognizes `do` as an Auxiliary and `you` as a pronoun, matching the beginning of the question pattern that handles questions of this type, where the subject and the auxiliary are inverted. The next item in this pattern is a verb-phrase. This information is then put on the parsing chart. Nothing yet gets built into **KODIAK**.

Read `know`

> **ALANA** now expects an S with which to build an instance of the **KODIAK** KNOW concept, and assert a VP on the parsing chart (which will then fulfill the expected VP from above).

Read `how`

> **ALANA** expects a how-question.

Read `to print`

> **ALANA** has a several patterns specific to *print*. These patterns have concepts that describe PRINT-ACTIONs and PRINT-EFFECTs. At this point, **ALANA** adds to its list of expectations an object associated with the PRINT-EFFECT.

Read `a file`

> This fulfills the expectation of a file for the PRINTing (pr-object in Figure 2). **ALANA** instantiates this to a NP. We note that at this point, in reading from left to right, an entire question has just been completed, namely "Do you know how to print a file?" Since **ALANA** does all the processing it can before reading the next word, it indicates in the parsing chart that a question has been asked. It is prevented, however, from using this question as its final interpretation because there are words remaining in the input. The remaining words will form a longer and more specific question, which will be returned by **ALANA**. **ALANA** favors the longer, specific patterns over the shorter abstract patterns.

Read `on the imagen`

> Two patterns are matched here. One is the general <NP$_1$> <on> <NP$_2$>. The other pattern is part of the <Print> <NP$_2$> <on> <NP$_2$>. The latter pattern builds a **KODIAK** structure indicating a destination for the print object, the *imagen*.

Now that we have read all the words in the sentence, the structure built so far is included in another **KODIAK** structure indicating that it is part of an *ASK* with a *QUESTION* whose *what-is* is the KNOW of that structure. **ALANA**'s complete output is shown in Figure 2.

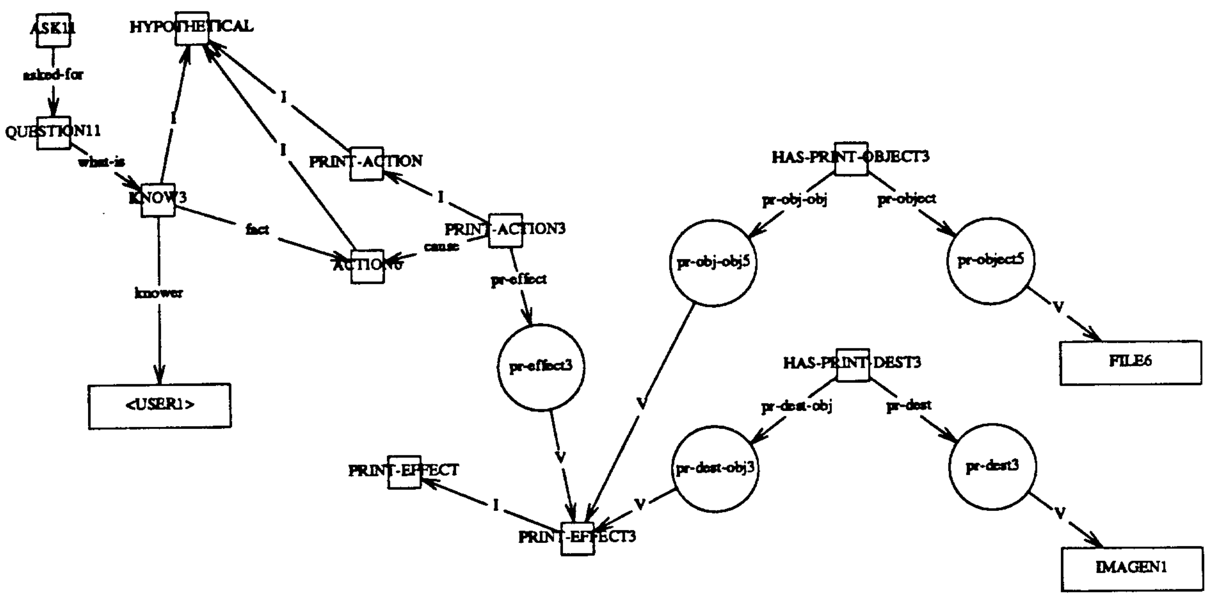


Figure 2
ALANA's output for "Do you know how to print a file on the Imagen?"

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 theory differs from traditional classification theories such as that of KL-ONE [Schmolze and Lipkis 1983, Brachman and 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.

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. This illustrates the following point: 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

A prototype concretion mechanism has been developed based on the KODIAK knowledge representation language. It is being used in the UNIX Consultant system as part of the interpretation mechanism, thereby reducing the number of specific patterns needed for language analysis. For example, the use of this mechanism delays semantic analysis that the language analyzer would otherwise accomplish by creating patterns for specific verb-object relations (such as "delete file"). Were the language analyzer to have specific linguistic patterns for every possible object of "delete," the resulting proliferation of patterns would not only require increased memory space but would fail to capture the generalization that all "delete"-object patterns share a common linguistic structure.

The mechanism concretes by using information about inheritance and value constraints, as well as by considering relation information between concepts. A concept that may be overly general is passed to the mechanism as a possible concretion target. 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 (indirectly retaining its status as an instance of the original category).

A simple example illustrating the role of an aspectual constraint in the concretion process is found in the categorization of "delete file" as a more specific type of deletion. Shown in Figure 3 is a partial representation of concepts associated with deletion. A stripped-down version of the concepts DELETE-EFFECT and DELETE-FILE-EFFECT appears in Figure 4. The salient difference between these two concepts is the additional requirement imposed by the del-file subcategory of del-object. DELETE-FILE-EFFECTs inherit all features of DELETE-EFFECT, and furthermore specify that the object deleted (which is the value of del-object) must be a file (which is the constraint on del-file).

Thus, if the concretion mechanism is passed a DELETE-EFFECT instance whose value for the del-object aspectual is an instance of FILE, the subcategorization conditions are met and it becomes instead a DELETE-FILE-EFFECT instance. Figure 5 shows the initial and final configurations of a DELETE-EFFECT instance being concreted to DELETE-FILE-EFFECT.

A slightly more complex concretion occurs when the mechanism is passed a DELETE-ACTION instead (see Figure 3 again). Here, the additional requirement that the del-effect must be a DELETE-FILE-EFFECT must be met before a concretion to DELETE-FILE-ACTION is permissible. The concretion mechanism must therefore determine eligibility for such parallel conceptual structures. This is done by recursively examining related concepts.

Moreover, in the special case that the file is a directory, concretion must continue to DELETE-DIRECTORY-ACTION.

Constraints are imposed on categories not only by aspectual values, but by instances of relations as well. For example, in Figure 6, the category PRINT-FILE-EFFECT is related to HAS-PRINT-FILE0, thus specifying the requirement that any instance of PRINT-FILE-EFFECT must participate in a HAS-PRINT-FILE relationship. This example is described in greater detail in the following section.

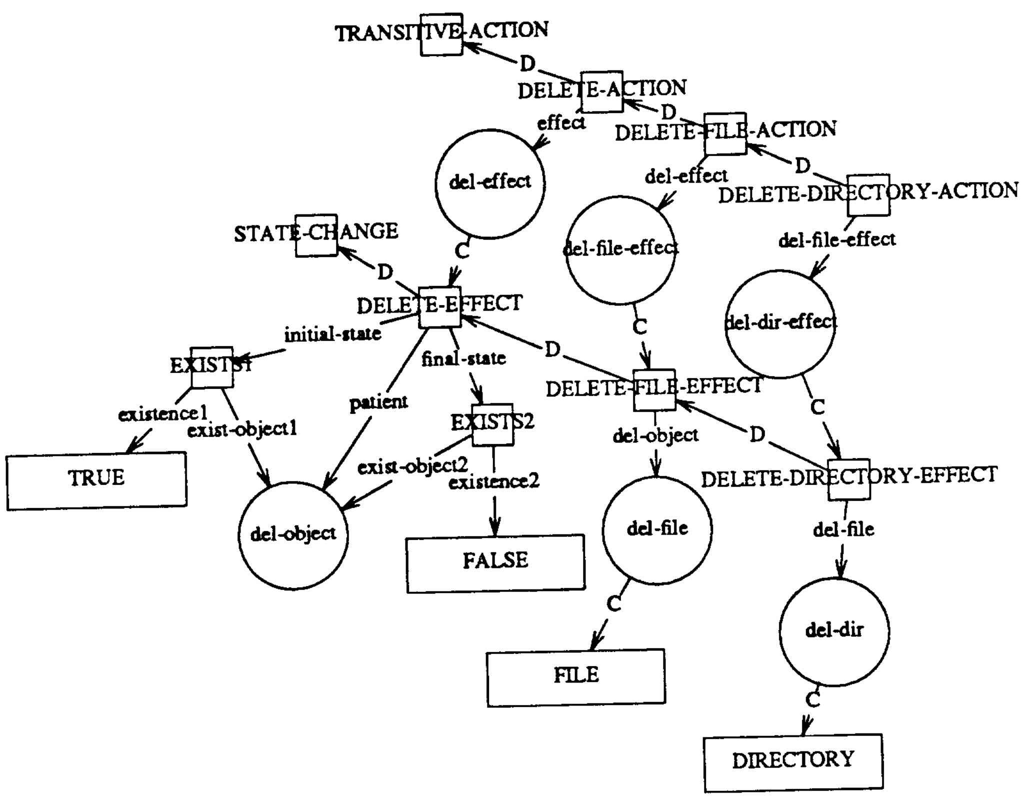


Figure 3
Partial representation of concepts associated with deletion.

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 7. PRINT-EFFECT3 is an instance of PRINT-EFFECT, which refers to the general concept of mechanical printing effects. Parts of the representation of printing are shown in Figures 8 and 9.

Besides the printing of the contents of a computer file, PRINT-EFFECT is 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 are all satisfied. Here, the only additional constraint on PRINT-FILE-EFFECT is a relation constraint, HAS-PRINT-FILE0, an instance of HAS-PRINT-FILE, which plays the role of HAS-PRINT-OBJECT. The pr-file aspectual, which plays the role of pr-object, must have a value that is a descendant of FILE. Since the value of pr-object5, FILE6, is indeed an instance of FILE, a concretion is made.

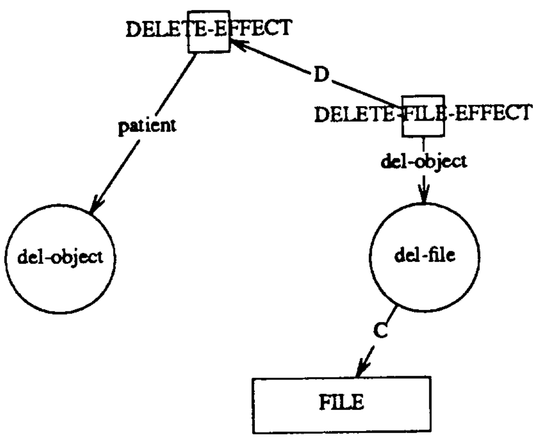


Figure 4
Stripped-down representation of deletion effects.

Parallel concretion occurs when multiple concepts must be simultaneously concreted to satisfy the constraints. Here, it is necessary to concrete PRINT-EFFECT3 to an instance of PRINT-FILE-EFFECT and HAS-PRINT-OBJECT3 to an instance of HAS-PRINT-FILE. Additionally, non-constraining relations should also be concreted when found in parallel. In this example, HAS-PRINT-DEST3 should be concreted to HAS-PRINT-FILE-DEST, although no HAS-PRINT-FILE-DEST relation constrains PRINT-FILE-EFFECT.

Continuing in this fashion, the mechanism concretes PRINT-EFFECT3 to LAS-PR-EFFECT, since the pr-dest3 value of IMAGEN2 is dominated by LASER-PRINTER. HAS-PRINT-OBJECT3 is simultaneously concreted to HAS-LAS-PR-FILE, and HAS-PRINT-DEST3 is concreted to HAS-LAS-PR-DEST. Finally, PRINT-EFFECT3 becomes an instance of IPRINT-EFFECT, HAS-PRINT-OBJECT3 becomes a HAS-IPRINT-FILE, and HAS-PRINT-DEST3 becomes a HAS-IPRINT-DEST.

We noted that HAS-PRINT-FILE-DEST does not constrain PRINT-FILE-EFFECT. The reason is that if a computer file is being printed, it can be assumed that the destination is a computer printer. A mechanism for supplying defaults is provided by the concretion mechanism. If no HAS-PRINT-DEST relation is supplied by the analyzer, and all other constraints on PRINT-FILE-EFFECT are satisfied, the default non-constraining relation that the destination is a printer is instantiated when the concretion is made.

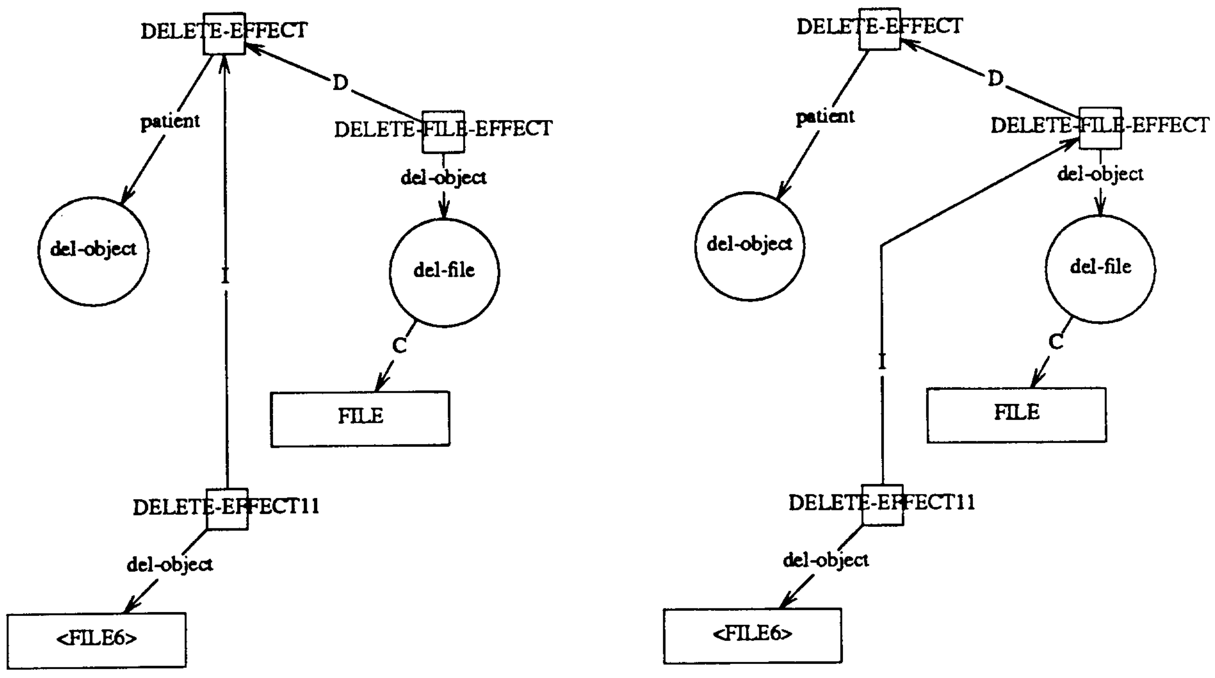


Figure 5
Initial and final representations of **DELETE-FILE-EFFECT** concretion.

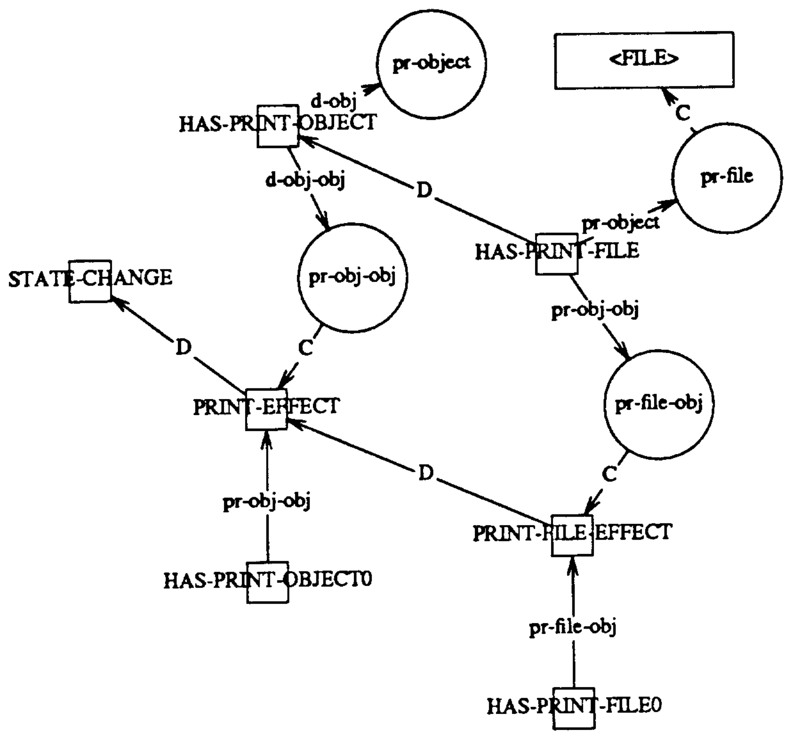


Figure 6
Relational constraint on **PRINT-FILE-EFFECT**.

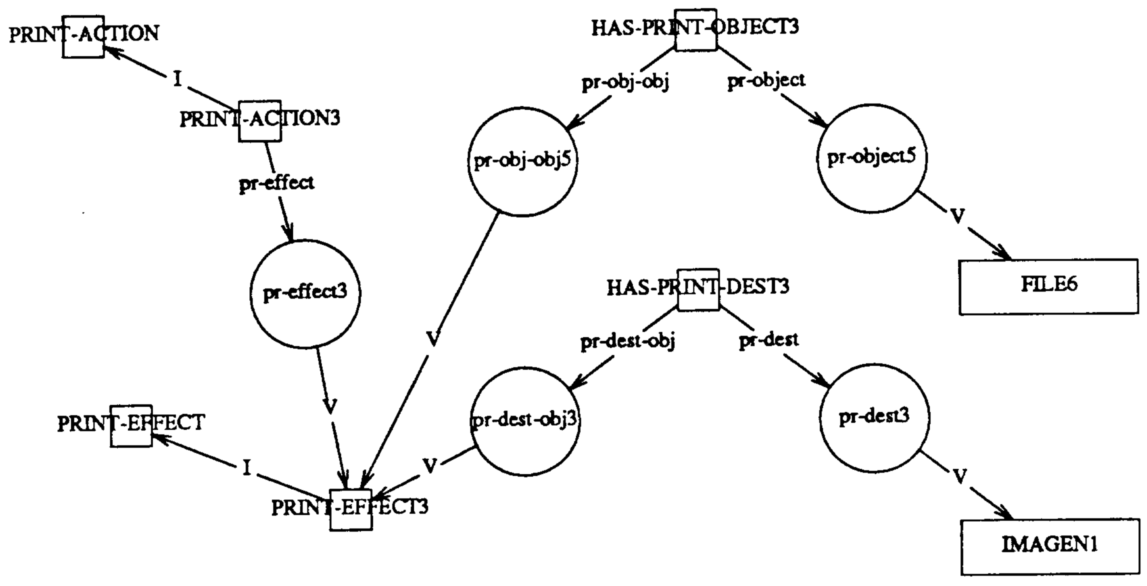


Figure 7
Representation of ''print a file on the imagen''

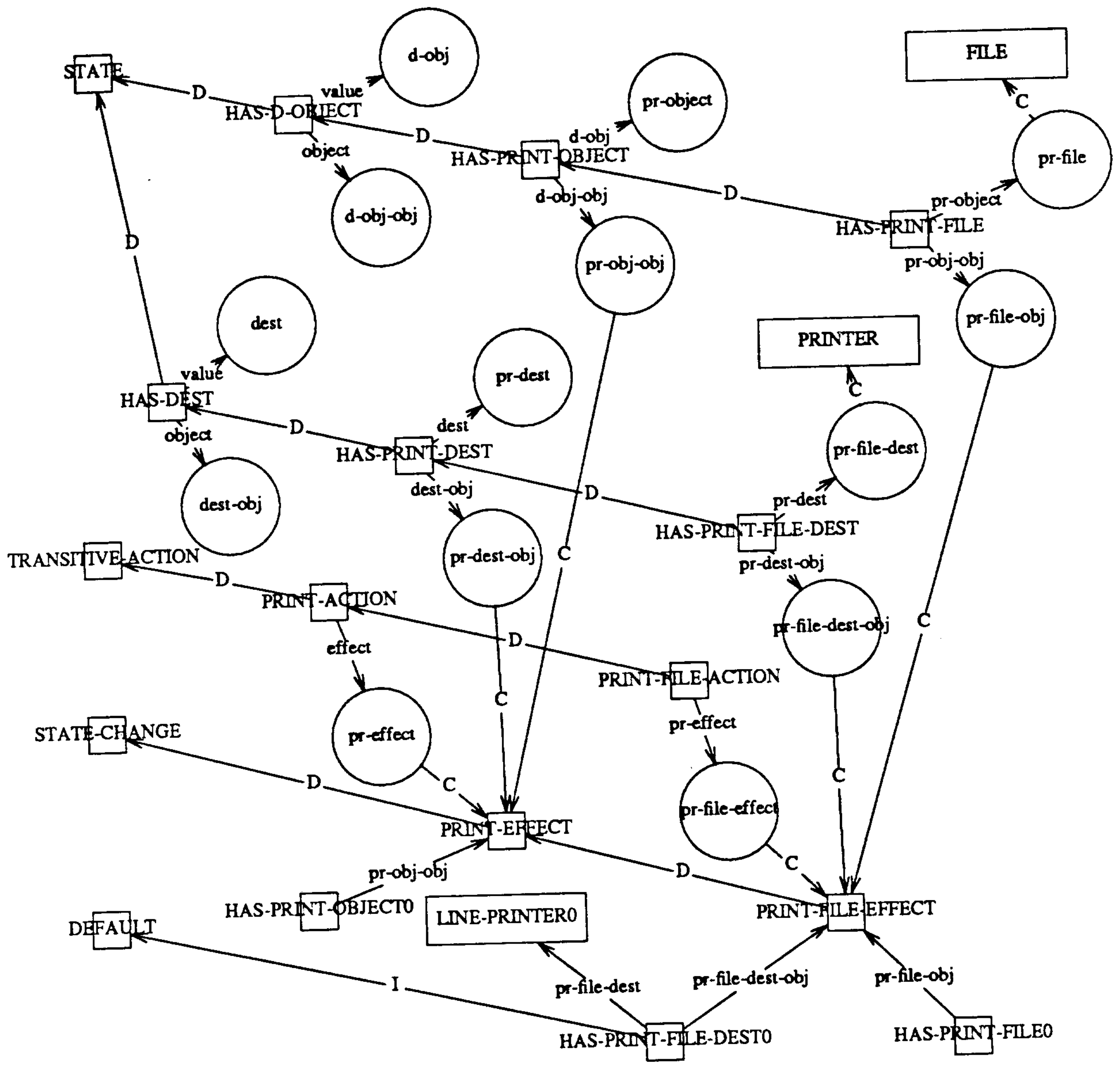


Figure 8
Representation of **PRINT-FILE-EFFECT**.

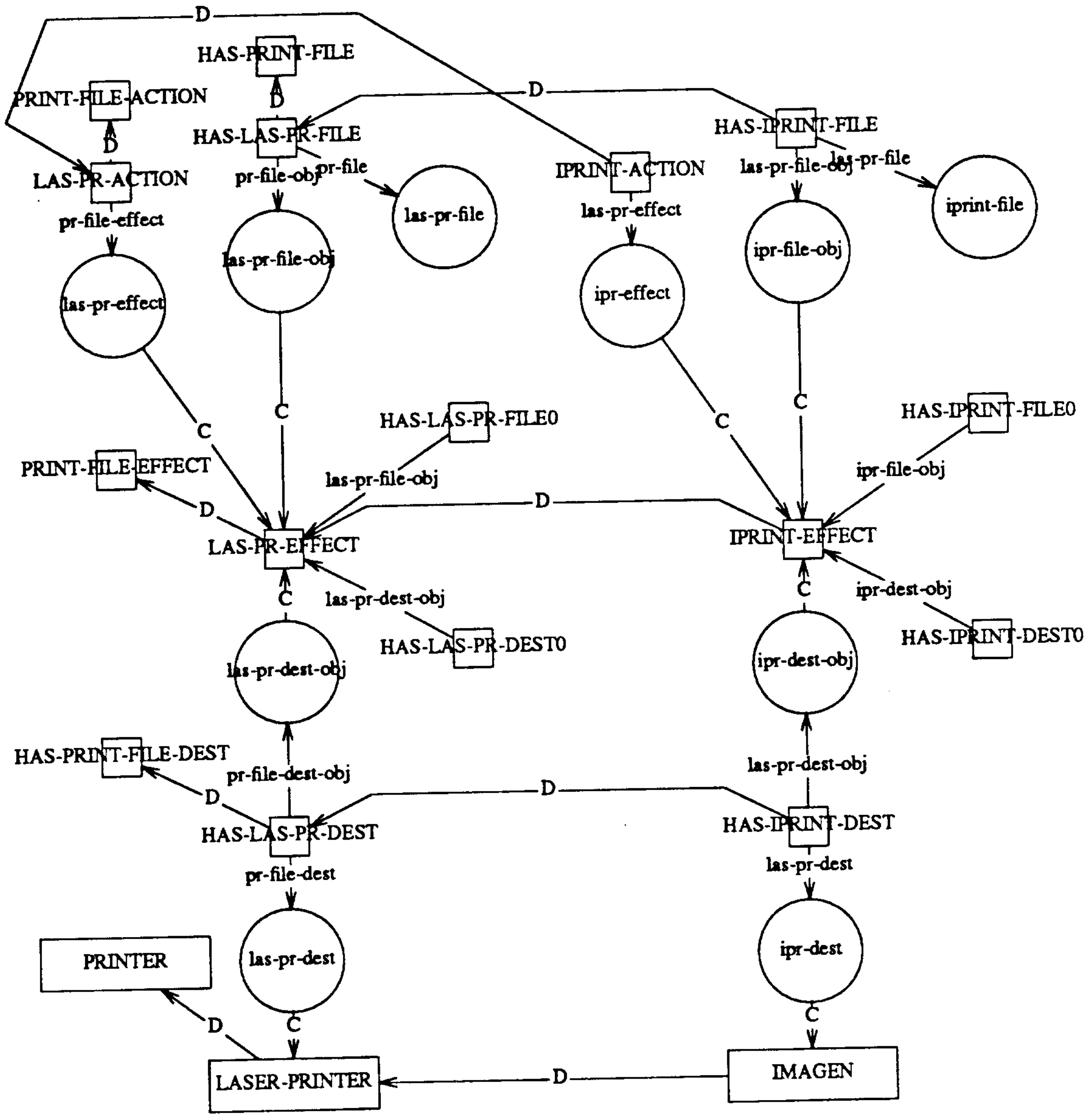


Figure 9
Representation of Imagen print effects, subordinate to **PRINT-FILE-EFFECT**.

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 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]. In speech act theory, an utterance is treated not as an abstract sentence but as an action performed by a speaker. A *direct speech act* is one whose intended meaning coincides with its literal interpretation; an *indirect speech act* is one whose intended meaning and literal interpretation differ. Thus, the use of

(2) How do I copy a file?

is a direct speech act, while the use of

(3) Do you know how to copy a file?

is likely to be an indirect speech act with the same intended meaning as that of (2).

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:

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

(5) Yes.

The ability to understand the full meaning of (5) is contingent on the realization that it relates directly and conventionally to (4). Thus PAGAN will require knowledge of such sequences to correctly determine the goal underlying utterances such as (5).

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 (2) 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

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 10. Figure 11 depicts an abstracted form of PLANFOR3 in its KODIAK form.

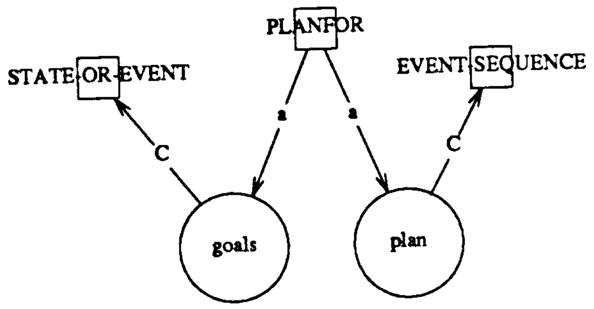


Figure 10
Definition of a **PLANFOR**

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

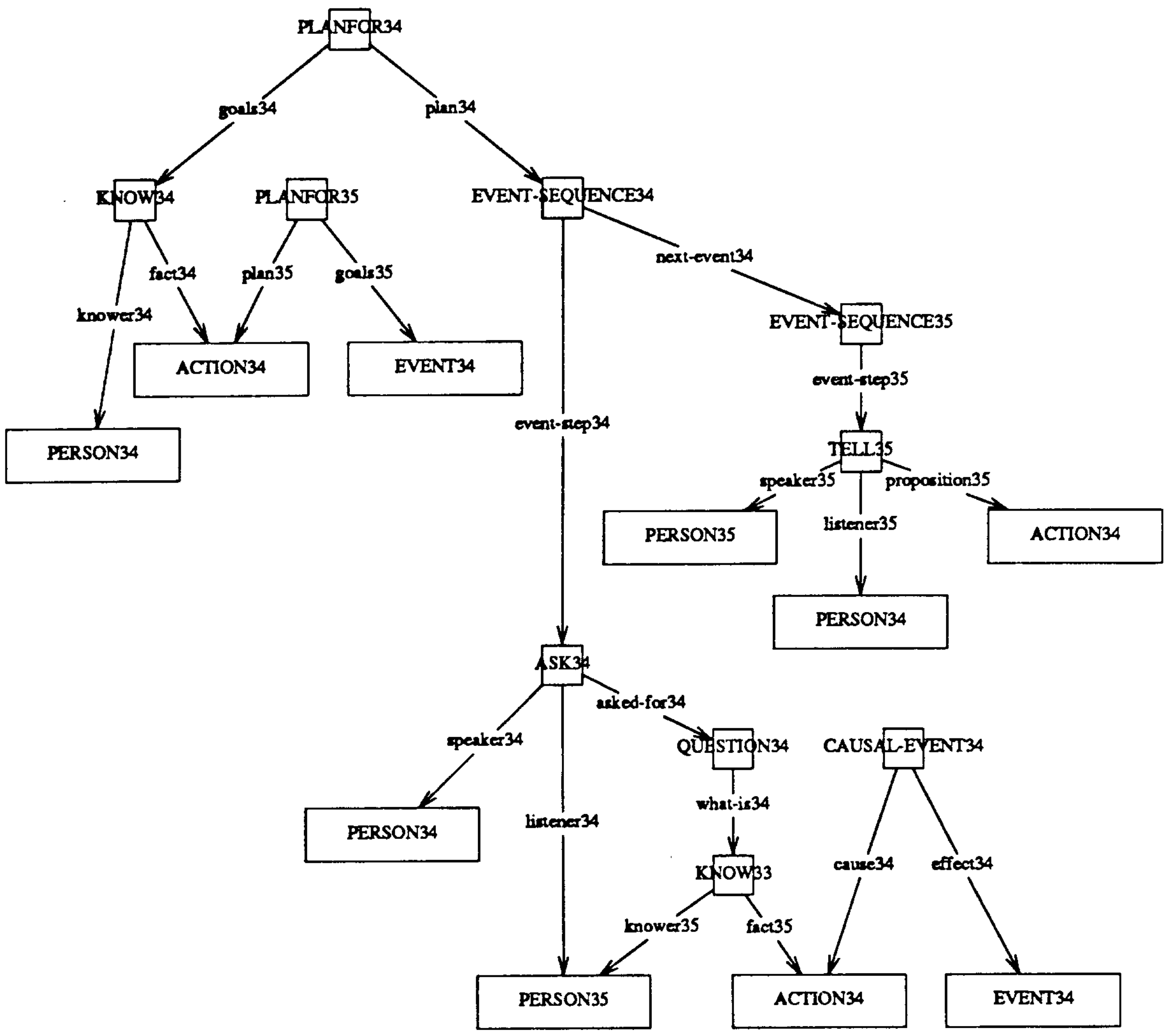


Figure 11
A plan for knowing is to ask if the hearer knows.

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 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, 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, and Litman 1982] is unnecessary, since the only difference between the two is the type of actions 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 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.

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, (2) is a plan for the goal of knowing how to copy a file. Achieving this goal may in turn be part of a plan for actually copying 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, (5) cannot be readily interpreted when viewed in isolation. However, if it is used in response to a question such as (4), 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 (5) would be matched against the second step of the question and answer plan started by (4) to determine if it is a response to the question. Second, when a match with a new planfor is found, PAGAN may also need to infer that the speaker has adopted the matched planfor. Suppose UC says (4) 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

(6) 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, PAGAN 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, PAGAN tries to reject an alternative as implausible. A planfor is implausible if PAGAN 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 PAGAN believes that the user believes that one of its plan steps could not be achieved.

5.3. Processing Overview

At PAGAN's core is a matching program that matches two KODIAK structures against one another. Two structures are said to match if they are isomorphic, 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, goat would match mammal or goat, and mammal would match goat, but goat would not match horse.

PAGAN 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, PAGAN 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 PAGAN to handle utterance (1). The input to PAGAN is the structure built by the analyzer from this utterance and refined by the concretion mechanism. A trace of PAGAN as it processes this structure is shown in Figure 12.

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

```
{1} This utterance wasn't expected.
{2} This utterance wasn't an adopted plan.
{3} Matching ASK0 pattern against ASK5.
{4} Could not match KNOW23 pattern to ACTION1 because of category KNOW.
{5} Match failed -- try the next one.
{6} Matching ASK1 pattern against ASK5.
{7} Match found.
{8} Matching ASK2 pattern against ASK5.
{9} Match found.
{10}Attempting to resolve ambiguity in the interpretation of ASK5.
{11}The alternatives are: KNOW5 KNOW6.
{12}Trying to determine whether KNOW5 was expected.
{13}KNOW5 was not expected.
{14}Trying to determine whether KNOW6 was expected.
{15}KNOW6 was not expected.
{16}The goal KNOW5 is implausible,
    since the speaker probably believes that it already holds.
{17}ASK5 is explained by the goal KNOW6.
{18}Creating new HAS-GOAL node: HAS-GOAL-ga0.
{19}Returning goal KNOW-ga0.
```

Figure 12
Trace of PAGAN's processing of
"Do you know how to print a file on the imagen?"

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 10. 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 know how to print a file on the imagen {16}. Thus, the planfor representing the indirect interpretation is selected {17}.

Once the utterance has been matched to the first step of this planfor, the remainder of the planfor is duplicated. In addition, a new HAS-GOAL relation is built {18}. The planner of this relation is the user, and the goal is the goal of the planfor. This HAS-GOAL represents the goal that the user had in mind in making the utterance, and is returned by **PAGAN** as its result {19}. It is shown in Figure 13.

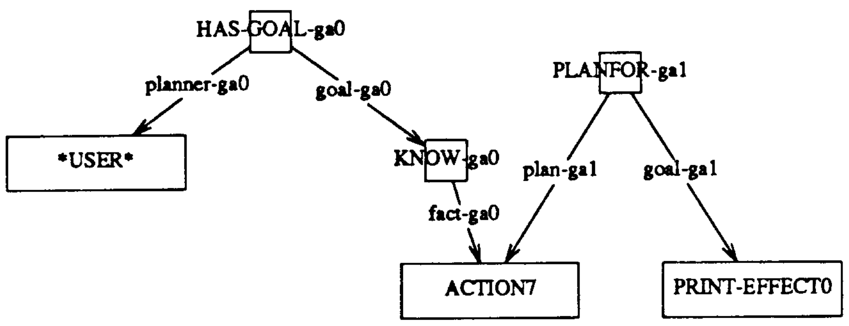


Figure 13
PAGAN output

6. The Ego Mechanism

6.1. Introduction

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 analyzer and concretion mechanism, and the user's goals and plans as inferred by UC's goal analyzer, PAGAN. UCEgo draws on the UNIX planner component of UC, UCPlanner, 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.

6.1.1. Themes and goals

In UCEgo, goal detection is implemented by *if-detected demons*. If-detected demons contain two subparts, a detection net and an addition net. Both of these nets 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 matched the detection net are used in place of the corresponding concepts in the addition net. Using this unification process, UCEgo avoids the need for the explicit variables found in other production 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 14 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 14 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 children nodes. The addition net consists of PLANFOR3 plus all its children nodes. 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.

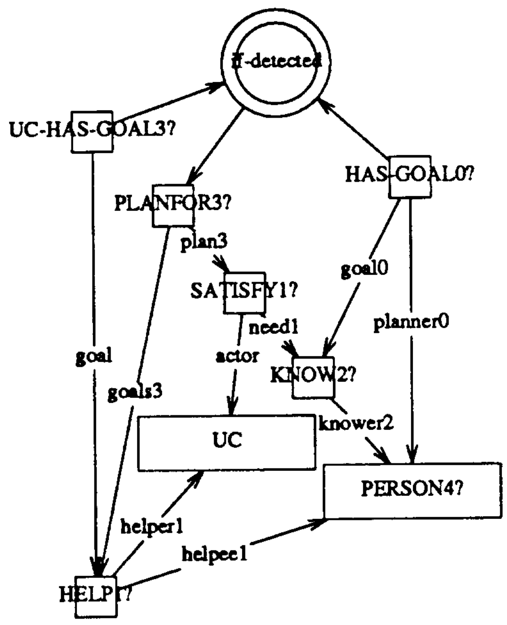


Figure 14
If-detected demon for adopting the user's goal of knowing.

The question-marks in the diagrams are significant to the demon interpreter both during matching and during 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.

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 one 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. Figure 15 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.

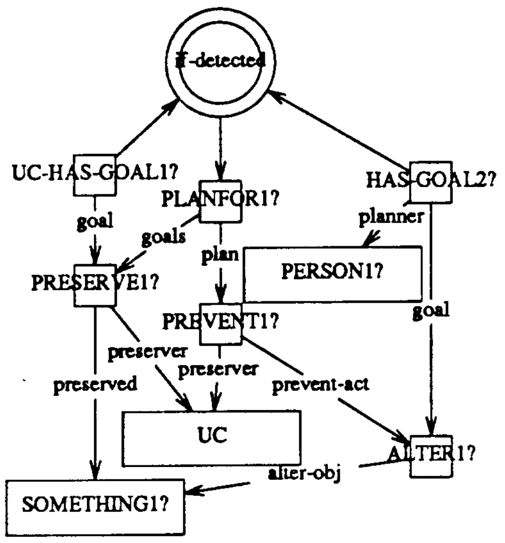


Figure 15
If-detected demon for detecting preserve/alter type goal conflicts.

Next, the goal of preventing the user from crashing the system, with the information (inferred by UC's User Model, q. v.) that the user does not know how to crash the system and the information that the user wants to know how (inferred by PAGAN), causes a new goal for UCEgo, namely preventing the user from knowing how to crash the system. Figure 16 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.

On 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, so as to help the user. Figure 17 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 goal of resolving the goal conflict. This *meta-goal* [Wilensky, 1983] has greater precedence than either of the conflicting goals, so the plan selection subcomponent of UCEgo handles the meta-goal first. In the scenario of preventing the user from knowing how to crash the system, the conflict is resolved by abandoning the less important of the two conflicting goals. Here, the

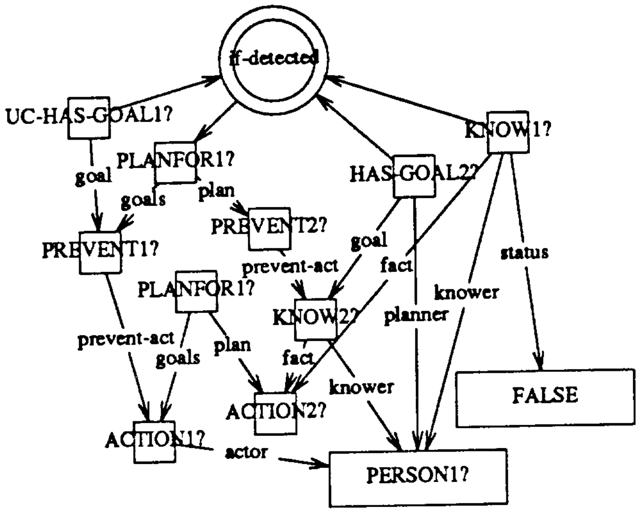


Figure 16
If-detected demon for preventing someone from knowing how to do something.

precedence relation between the two conflicting goals is inherited from their motivating themes. Since UCEgo's staying alive life theme has greater precedence than its consultant role theme, the goal of preventing the user from knowing wins out 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 *planfor* s. A planfor is a relation between a plan and the goal that the plan is designed to satisfy. 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 example, Figure 18 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 18. 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.

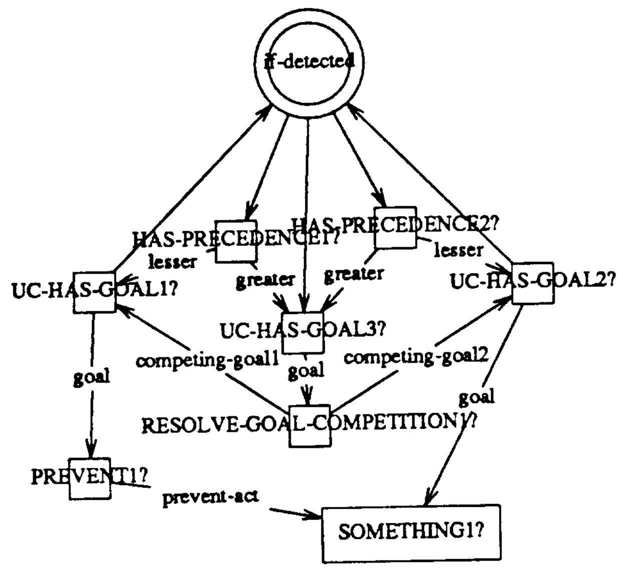


Figure 17
If-detected demon for detecting goal conflicts.

After selecting the right plan, UCEgo proceeds to execute the plan. Figure 19 shows the basic knowledge used by UCEgo to adopt a plan for execution.

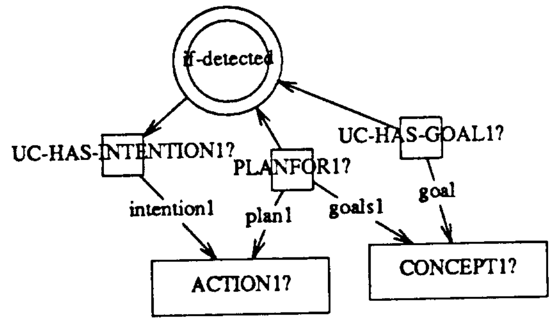


Figure 19
Principal if-detected demon used to adopt a plan.

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. After UCEgo has adopted an intention to perform some action, a specialized interpreter calls the proper subcomponent to perform the action. An example of this is when UCEgo calls UCExpress

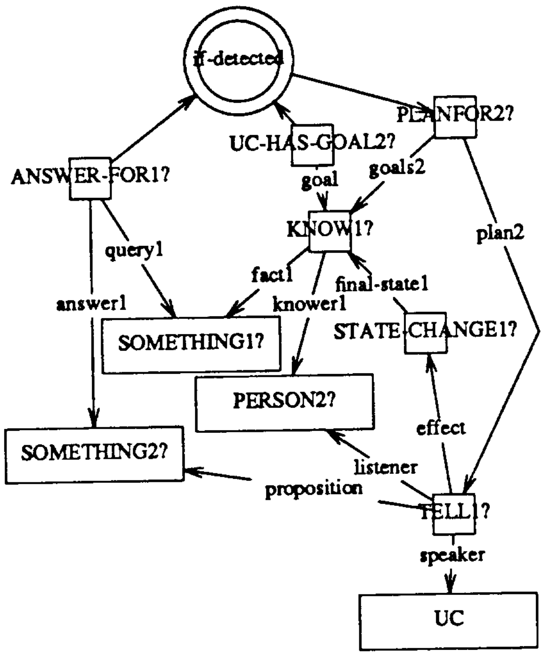


Figure 18
If-detected demon for suggesting the plan of telling the user.

to perform a TELL action.

6.2. User Model

The **User Model** in **UC** is used by the goal detection phase of **UCEgo**'s processing and by **UCExpress**. The **User Model** encodes the user's knowledge state instead of other user attributes such as personality traits [Rich, 1979], user preferences [Morik and Rollinger, 1985], or user performance as it departs from an expert [Brown and Burton, 1976].

In **UC**'s **User Model**, users are separated 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 [Rosch, 1978] for inference.

Besides separating users according to expertise, **UC**'s **User Model** 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 UC's **User Model** can be summarized using the five statements shown in Figure 20.

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 20
Summary of UC's **User Model**.

6.2.1. Detecting Misconceptions

One of many uses of the **User Model** in **UCEgo** is to detect user misconceptions. The user model analyzes the user's statements to make inferences about what the user believes and knows (the user knows some fact if and only if the user believes the fact and it is true). A misconception is when the user believes something that is false. An example of a user misconception is 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.

The **User Model** 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 and 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. To handle examples such as ls -v, UC's knowledge base would have to encode many facts indicating that particular command-option combinations are illegal. This would be an inefficient use of space.

What UC's **User Model** does instead is to augment an open world hypothesis with *meta-knowledge*. Meta-knowledge is knowledge that the **User Model** has about what UC itself does or does not know. For example, the **User Model** 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, the **User Model** is able to infer that -v is not an option of ls, hence the user has a misconception. This fact is passed on to **UCEgo**, which adopts the goal of correcting the user's misconception.

6.2.2. Inferring the User's Level

During a session, the **User Model** builds a profile of the user and infers the user's level of expertise. This proceeds in a two step process. First, the **User Model** infers particular facts about what the user does or does not know from the dialogue. Next these facts are combined to 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 $\rightarrow$ the user does not know ?x

This rule is implemented using the if-detected demon shown in Figure 21.

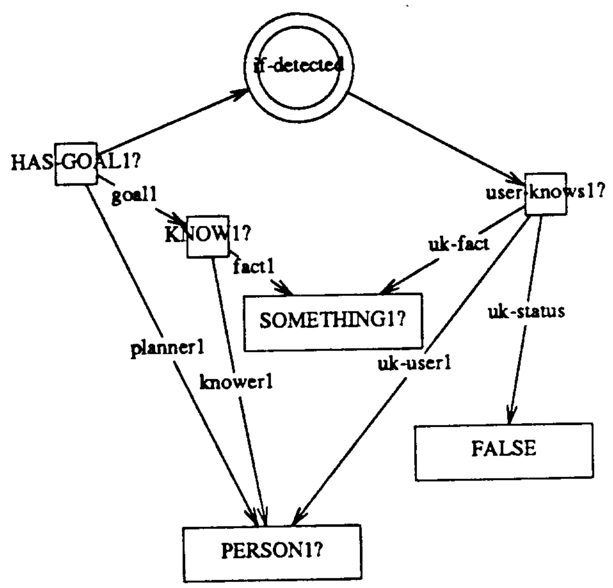


Figure 21
If-detected demon used for inferring that the user does not know something.

Based on such facts about what the user does or does not know, the **User Model** 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 the user's perceived level of expertise. For more details on this and other issues addressed in **UC**'s **User Model** (e.g. dealing with the inherent uncertainty of information encoded in the model, representing individual users, etc.), see [Chin, 1986].

6.3. Example

To see how **UCEgo** works in more detail, consider the example, "Do you know how to print a file on the imagen?" After **UC** has parsed and understood the question, **UC**'s goal analyzer, **PAGAN**, asserts that the user has the goal of knowing a plan for printing a file on the imagen. At this point, the interesting part of **UCEgo**'s processing begins. The following annotated trace 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 (goal19 = (HELP2 (helpee2 = *USER*) (helper2 = UC))))
(HAS-GOAL-ga0 (planner-ga0 = *USER*)
              (goal-ga0 = (KNOW-ga0 (knower-ga0 = *USER*)
                                    (fact-ga0 = (ACTION7 (actor7 = *USER*))))))
and asserts the following concept into the database:
(PLANFOR29 (goals29 = (HELP2 (helpee2 = *USER*) (helper2 = UC)))
           (plan29 = (SATISFY2 (need2 = (KNOW-ga0 &))
                               (actor2-0 = UC))))
```

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

HAS-GOAL-ga0, which is also shown in Figure 13, 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 (goal19 = (HELP2 (helpee2 = *USER*) (helper2 = UC))))
(PLANFOR29 (goals29 = (HELP2 (helpee2 = *USER*) (helper2 = UC)))
           (plan29 = (SATISFY2 (need2 = (KNOW-ga0 &))
                               (actor2-0 = UC))))
and asserts the following concept into the database:
(UC-HAS-INTENTION6 (intention6 = (SATISFY2 (need2 = (KNOW-ga0 &))
                                           (actor2-0 = UC))))
```

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

```
UCego detects the following concepts:
(UC-HAS-INTENTION6 (intention6 = (SATISFY2 (need2 = (KNOW-ga0 &))
                                           (actor2-0 = UC))))
and asserts the following concept into the database:
(UC-HAS-GOAL20 (goal20 = (KNOW-ga0 (knower-ga0 = *USER*)
                                   (fact-ga0 = (ACTION7 (actor7 = *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) to activate the detection net of the if-detected demon shown in Figure 14. 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 planfor 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 is a result of UCEgo's if-detected demon for plan selection, which is shown in Figure 19. 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-ga1 = PRINT-EFFECT0)
             (plan-ga1 = (ACTION7 (actor7 = *USER*))))
(UC-HAS-GOAL20
   (goal20 = (KNOW-ga0 (knower-ga0 = *USER*)
                       (fact-ga0 = (ACTION7 (actor7 = *USER*))))))
and asserts the following concept into the database:
(UNIX-planner1 (user-goals1-0 = 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 UCPlanner.

```
The planner is passed:
PRINT-EFFECT0
```

The planner produces:

```
(PLANFOR70 (goals70 = PRINT-EFFECT0)
           (plan70 = (UNIX-IPR-COMMAND0 (ipr-file0 = FILE0)
                                         (UNIX-IPR-COMMAND-effect0 =
                                            PRINT-EFFECT0))))
```

PLANFOR70 says that a plan for achieving the goal of PRINT-EFFECT0 is to use UNIX-IPR-COMMAND0, which has the name of lpr -Pip (not shown).

UCego detects the following concepts:

```
(ANSWER-FOR1
   (answer1-0 =
      (PLANFOR70 (goals70 = PRINT-EFFECT0)
                 (plan70 = (UNIX-IPR-COMMAND0 (ipr-file0 = FILE0)
                                               (UNIX-IPR-COMMAND-effect0 =
                                                  PRINT-EFFECT0)))))
   (query1-0 = (ACTION7 (actor7 = *USER*))))
(UC-HAS-GOAL20 (goal20 = (KNOW-ga0 &)))
```

and asserts the following concept into the database:

```
(PLANFOR30 (goals30 = (KNOW-ga0 &))
           (plan30 = (TELL4 (listener4-0 = *USER*)
                            (speaker4-0 = UC)
                            (proposition4 = (PLANFOR70 &))
                            (effect4 = (STATE-CHANGE1
                                          (final-state1-0 = (KNOW-ga0 &)))))))
```

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 (goal20 = (KNOW-ga0 &)))
(PLANFOR30 (goals30 = (KNOW-ga0 &))
           (plan30 = (TELL4 &)))
```

and asserts the following concept into the database:

```
(UC-HAS-INTENTION7 (intention7 = (TELL4 &)))
```

UC-HAS-INTENTION7 represents UC'*s intention of telling the user (TELL4).*

UCego detects the following concepts:

```
(UC-HAS-INTENTION7 (intention7 = (TELL4 &)))
```

and asserts the following concept into the database:

```
(UCexpress1
   (gen-prop1-0 =
      (TELL4 (listener4-0 = *USER*)
             (speaker4-0 = UC)
             (proposition4 =
                (PLANFOR70 (goals70 = PRINT-EFFECT0)
                    (plan70 = (UNIX-IPR-COMMAND0
                                  (ipr-file0 = FILE0)
                                  (UNIX-IPR-COMMAND-effect0 =
                                     PRINT-EFFECT0)))))
          (effect4 = (STATE-CHANGE1
                        (final-state1-0 =
                          (KNOW-ga0 (knower-ga0 = *USER*)
                                     (fact-ga0 = (ACTION7 (actor7 = *USER*)))))))))
```

UCexpress1 represents a call to UCExpress *to execute TELL4.*

Annotated trace of UCEgo's plan selection process.

The first step of the plan is to call the UNIX planner component of UC, UCPlanner. Figure 22 shows the if-detected demon that calls UCPlanner.

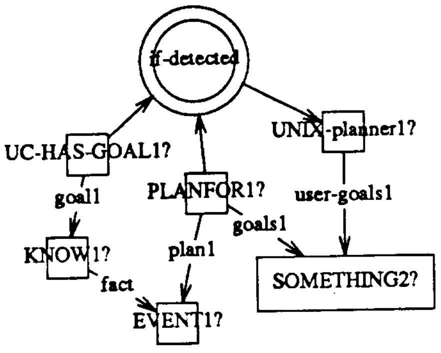


Figure 22
If-detected demon for calling the UNIX planner component of UC, UCPlanner.

The UNIX Planner, UCPlanner, is called whenever UC has the goal of knowing a plan for something (SOMETHING2 in Figure 22). In the trace, UC-HAS-GOAL20 and PLANFOR-ga1 combine to cause a call to UCPlanner (UNIX-planner1 in the trace). UCPlanner 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 18 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 19. 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.

7. The Planner

7.1. Introduction

This section describes **UCPlanner** [Luria 1985], a knowledge based commonsense planner [Wilensky 1983]. **UCPlanner** includes:

- A knowledge representation scheme to represent plans.
- A planning component that uses this knowledge to
  - Find potential plans for problem situations
  - Notice potential problems with these plans
  - Use meta-planning knowledge (knowledge about plans) to determine which plans are suggested.

7.2. Planning Process in UCPlanner

The following are the steps of the iterative process that **UCPlanner** uses:

(1)Goal detection
- input goals from **UCEgo**
- detection of background goals

(2)Plan selection - select a possible plan among all the known plans in the system
- Use a stored plan that is related to the user's goals
- Propose a new plan based on knowledge in the system

(3)Projection --test whether plan would be executed successfully
- Check all conditions
- Notice bad side effects
- Detect new goals -- and find plans to solve them

(4)Plan Evaluation
- Determine whether plan is impossible
- Determine whether plan is undesirable

The iterative structure described here is actually 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

UCPlanner's main goals are the domain goals that are passed to it by **UCEgo**. **UCPlanner** does not deal with any of UC's goals - it only creates plans for the user's UNIX goals. The user's goal may be a complex goal that can be decomposed into several simple goals. For example, if the user asks:

How do I move the file named file1 from machine1 to machine2?

the complex goal of moving a file to another machine is passed to **UCPlanner** by **UCEgo**. This complex goal is decomposed by **UCPlanner** into the simple goals:

a) file1 exists on machine2 (destination) with same contents and name
b) file1 no longer exists on machine1 (source)

In addition, other goals may be detected automatically by **UCPlanner**. For example, new goals are detected during the projection of possible plans. This might occur when **UCPlanner** notices that a selected plan would fail unless a condition is satisfied. The satisfaction of this condition becomes a new goal for **UCPlanner** and is added to its list of goals for this planning situation.

Another kind of goal that arises automatically is called a background goal. In UNIX, examples of background goals include having access to files and maintaining a low system load average. When dealing with a particular file the goal detector must detect the individual background goals that are associated with the file. **UCPlanner** checks only those background goals that are pertinent to the particular planning situation. This is done by indexing the pertinent background goals under the appropriate planning situation in the knowledge base.

Finally, the goal detector must notice goal conflicts. These goal conflicts will cause yet another goal to be instantiated: the resolution of these goal conflicts. In **UCPlanner**, there are often goal conflicts between user goals and background goals. For example, if the user wants to delete a file, this conflicts with the background goal of having access to the that file. The goal of resolving this conflict is added to the list of goals for this planning situation.

7.2.2. Plan Selection

Plan Selection is the process of selecting a potential plan to satisfy the user's goals, from among the plans known to the system. This potential plan is then tested during the rest of the planning process. If the plan passes these tests, 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 solving the goals to be achieved. In other words, if this specific goal has been encountered before and there is a specific stored plan that is a plan for this goal, then suggest that specific plan. Saved plans encode information such as the following:

- Use the rm command to delete a file
- Use the mv command to move a file
- Use the lpr command to print out a file on the lineprinter
- Use the who command to see who is on the system

Many important questions can be answered with stored plans. However, to answer more interesting problems it is necessary to be able to build new plans from existing plans. It would be impossible and undesirable to index an appropriate plan for each of the possible queries that a user might have.

7.2.2.1. New Plans

When **UCPlanner** has no stored plan for a particular goal, it selects a plan for a goal similar to the user's goal. **UCPlanner** 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, or GSMA.

For example, when selecting a plan for the goal of moving a file to another machine, there is no stored plan for this goal. Therefore, UCPlanner 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, UCPlanner searches for plans of goals that are dominated by these two goals. Figure 23 shows that one command, rcp, falls in this category. This command is used to copy a file from one machine to another.

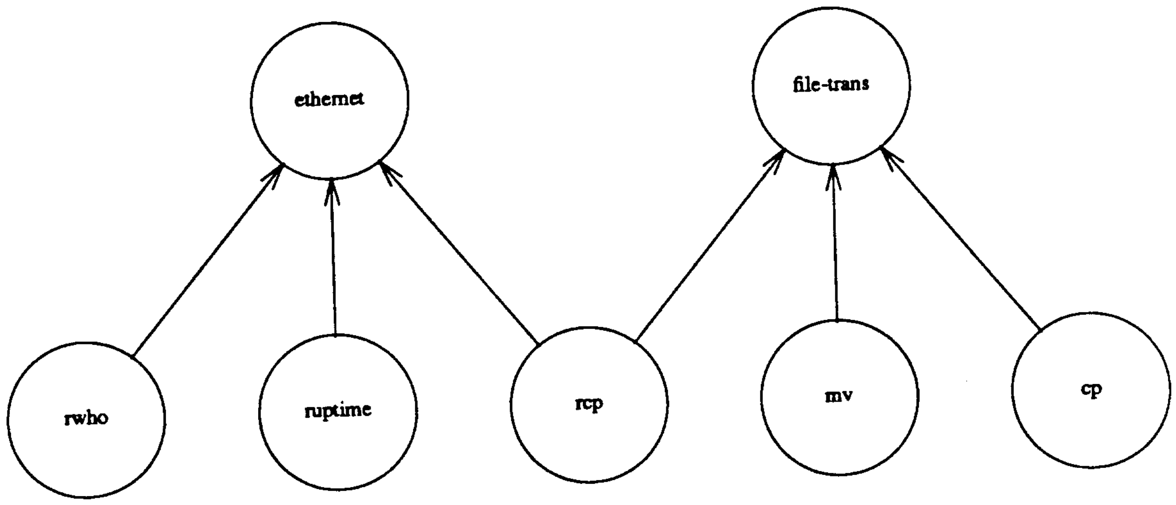


Figure 23
Hierarchy of Ethernet and File Transfer Commands

UCPlanner selects the USE-RCP-COMMAND plan as a *potential* plan to move a file to another machine. This plan is then tested during the rest of the planning process. If the plan does not satisfy all the goals of the user, the plan is fixed by determining which simple goals of the user's complex goal are left unsatisfied. Plans for these simple goals will be selected by the same algorithm, i.e., using a stored plan or looking at plans of goals similar to these goal parts. In this way, UCPlanner progressively refines the potential plan until as many goals as possible are satisfied by using plans for similar goals.

It is informative to contrast GSMA with the more traditional way of selecting a potential plan, namely *means-end* analysis [Newell and Simon 1972]. Means-end analysis was used by planners like GPS [Ernst and Newell, 1969] and STRIPS [Fikes and Nilsson 1971]. It entails examining all the plans in the database and selecting the plan that reduces the greatest difference between the present state and the goal state. STRIPS was given a well formed formula describing the goal state and the present state, and a set of formal descriptions of available operations, and attempts to prove the goal state true. If an individual subgoal of the goal state cannot be "proved" from the present state, STRIPS selects an operator that will allow the proof attempt to continue. For example, if

STRIPS were asked to find a plan for the goal of moving a file from one machine to another, it would look for a plan that reduces the greatest difference between the present state of file1 existing on machine1 and not on machine2, and the goal state of file1 existing on machine2 and not on machine1. The difference between the goal state and the present state is: 1) file1 exists on machine2 in the goal state and file1 does not exist on machine2 in the present state, and 2) file1 does not exist on machine1 in the goal state and file1 does exist on machine1 in the present state. STRIPS would look through all its plans and find two pertinent plans, USE-RCP-COMMAND to copy the file and reduce difference (1), and USE-RM-COMMAND to delete file1 and reduce difference (2). Since these two plans reduce the same amount of difference, according to the formal criteria of STRIPS, it might arbitrarily choose to use the USE-RM-COMMAND plan first and then look for another plan to reduce the difference that is left. However, since once the file is deleted it is impossible to use the USE-RCP-COMMAND, this plan will fail.

ABSTRIPS [Sacerdoti 1974], which modified STRIPS to avoid interacting subgoal problems, might actually do worse than STRIPS in this particular example. ABSTRIPS used GPS's idea of reducing the *important* differences in the problem first, by assigning criticality levels to differences and reducing those differences with the highest criticality first. Criticality levels were assigned by the program itself according to how hard it was to satisfy the preconditions for plans to reduce a particular difference. In the cross-machine move example, however, removing a file on machine1 might have a higher criticality level than copying a file from machine1 since copying a file only requires read permission, whereas deleting a file requires write permission on the parent directory, which is more critical than read permission.

In GPS, operators are selected according to which operator reduced the greatest difference by using a precomputed difference table. Differences are reduced in order of difficulty according to a predetermined ordering called a DIFF-ORDERING. The DIFF-ORDERING is assigned by the GPS implementor based on how easy tasks are to accomplish. For example, a human expert might determine that deleting a file is more difficult than creating a file, and assign a DIFF-ORDERING accordingly.

Both ABSTRIPS and GPS could conceivably choose to reduce difference (2) first since it is at a higher criticality level, or higher up in the DIFF-ORDERING. Since they would solve difference (2) first, they would have to deal with the interacting subgoal of the file being deleted before it is copied.

Each of the three planners described above uses means-end analysis, but technical problems prevent them from efficiently determining the proper difference to reduce. STRIPS' formal criteria would be unable to choose which subgoals to reduce. Both GPS and ABSTRIPS reduce differences in order of their importance. However, since they both use difficulty as a metric for importance, they sometimes erroneously deal with the *difficult* parts of problem before the *important* parts of a problem. However, the real problem with using these planners for knowledge intensive problems, is that they deal with knowledge about goals on too low a level. Rather than select a plan based on the complex goal of a cross-machine move, these planners are forced to individually deal with the simple goals that comprise this complex goal.

UCPlanner addresses this problem by applying knowledge about the higher level complex goal during plan selection using the GSMA algorithm. Rather than use only knowledge about the lower level simple goals, UCPlanner uses the conceptual hierarchy of the complex goal to find plans for goals that are similar. Since these goals are close in the conceptual hierarchy, they are similar in important ways, not just on the surface. GSMA can thus be viewed as a way to find the plan that reduces the difference between the user's goal and present state by the greatest amount, where similarity determines the important difference to reduce. For example, in the cross machine move example, **UCPlanner** chooses USE-RCP-COMMAND over USE-RM-COMMAND, even though this may not reduce the most difficult difference between the present state and the goal state.

Our experience with this Goal Similarity Matching Algorithm suggests that problems of interacting subgoals do not occur often when using this algorithm on knowledge intensive problems. In addition, using GSMA reduces search time, since **UCPlanner** only needs to consider plans for the few goals that are similar to the user's goals, rather than considering all the possible plans in the database.

7.2.2.2. Plan Specification

Once the system has determined that it wants to test a plan, it is necessary to specify the values of certain parts of the plan. This is necessary because the plans are described in long term memory in general terms. For example, if 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 24.

But **UCPlanner** must return a specific plan for this planning situation. In the general plan shown in Figure 24, there is no value for the del-file, the file to be deleted, or the rm-file, the file on which the rm command is applied. These must both have the same value in order for the plan to work successfully. In addition, there is no value for the rm-file-arg, the argument of the rm command. During plan specification, **UCPlanner** creates a new instance of the UNIX-RM-COMMAND and fills in the appropriate specific values for these arguments by looking at the general plan. This specific plan shown in Figure 25, specifies that the value of the del-file (the file to be deleted) is FILE1 (whose filename is *junk*), and the value of the rm-file-arg, the argument to the rm command is the string "junk". In other words, this specific plan states that when the user has a goal of deleting a file whose name is junk, use the rm command with the argument "junk". It is this specific plan that is tested during the rest of the planning process.

7.2.3. Projection

It is next necessary to test whether the plan as developed so far would actually execute successfully. Detailed descriptions of all UNIX commands are in the knowledge base, so most potential caveats are noticed during Plan Projection. 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.

7.2.3.1. Defaults

Defaults are important to the projection mechanism since much of the information that is necessary to simulate a plan may not be supplied by the user. The user might be unaware of some conditions, and certain conditions like the machine being up and the file

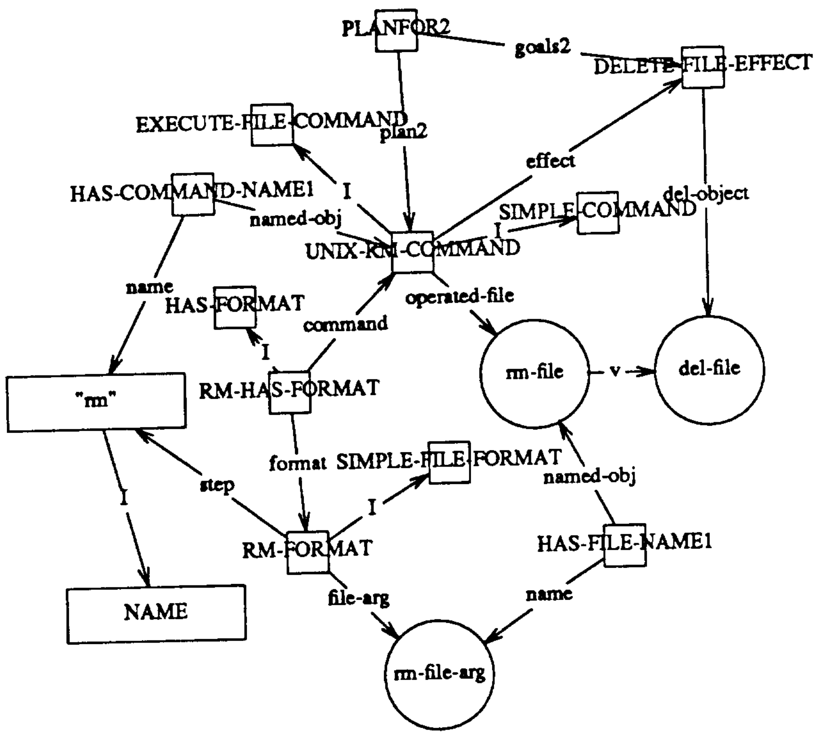


Figure 24
Representation of plan to delete a file by using the rm command

existing could be assumed. It would be undesirable to prompt the user for all such information, but this could be done if there was not enough information to make such an assumption. Defaults can also be useful in indexing, since when defaults are violated, certain plans may be more desirable. For example, when printing out a file, if the file is not the default text file, but a program, use the line printer instead of the laser printer.

7.2.3.2. Condition checking

To determine whether a potential plan will work, UCPlanner must determine whether the conditions specified in the plan are true in this planning situation. In the example of moving a file to another machine, conditions that are checked for the rcp command include: the user has an account on both machines; the user can read the file he wants to copy; and the user can write the file he wants to copy it to. These are defaults that are assumed to be true unless otherwise specified.

7.2.3.3. New Goal Detection

If the plan under consideration has effects that are not part of the user's goals, UCPlanner must decide if these effects conflict with any explicit goals of the user or background goals. If such a goal conflict exist, resolution of the goal conflict becomes a new

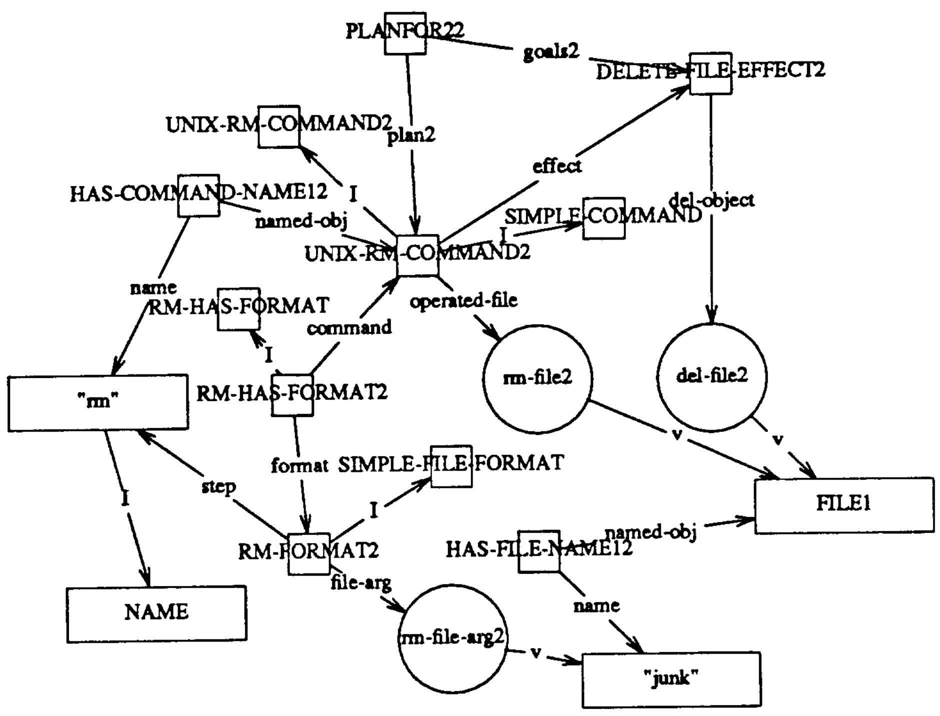


Figure 25
Representation of specific plan to delete a file named junk by using the rm command on the file named junk.

goal for **UCPlanner**.

For example, when **UCPlanner** is trying to find a plan to move a file to another machine, one effect of the rcp command is that a file of the same name on the destination machine is deleted. This deletion effect is matched against the background goals that **UCPlanner** knows about. Deleting the file conflicts with the background goal of keeping access to files. **UCPlanner** notices this conflict because it has stored the knowledge that the goal of deleting a file subverts the goal of accessing the file. In the implementation described in this paper, this fact is stored by creating a HAS-OPPOSITE relation between DELETE-FILE-EFFECT and FILE-ACCESS. (In future implementations of **UCPlanner**, we expect to represent more complex relationships between conflicting goals).

UCPlanner next determines how to resolve the goal-conflict. In this example, since there is no information to suggest that the user might have a file with the same name on the destination machine, the background goal is considered unlikely and therefore no resolution is necessary. Knowledge about common potential background goal conflicts and their resolutions is incorporated in the representation of the plan in long term memory. For example, if **UCPlanner** knows that the user has a file of the same name on the destination machine, executing the rcp command would conflict with the goal of preserving access to that file. **UCPlanner** has a stored resolution for this goal conflict, namely, to first rename this destination file using the mv command.

Finally, if any goals remained to be solved, **UCPlanner** needs to go back to the Plan Selection phase of the algorithm. In the cross-machine move example, the only problem with the rcp plan is that it leaves the original file on the source machine. The goal of the original file no longer existing on the source machine was one of the simple goals that **UCPlanner** inferred from the complex goal of moving a file from one machine to another. Since the other parts of the complex goal have been satisfied, the simple goal of the file not existing on the source machine becomes the new goal for **UCPlanner** to solve.

UCPlanner thus returns to the Plan Selection phase of the algorithm to plan for the goal of removing the source file. There is a stored plan for this goal, i.e., use the rm command. This plan is then specified during Plan Specification, and the entire plan is tested during Plan Projection. Here, the plan of copying the source file to the destination file using rcp, and then deleting the source file using rm, satisfies all the goals of the user.

7.3. Example and Trace Output of UCPlanner

This example does not require much work by **UCPlanner** because there is already a known plan for the goal of printing a file on the imagen. The input to the planner is shown in Figure 26.

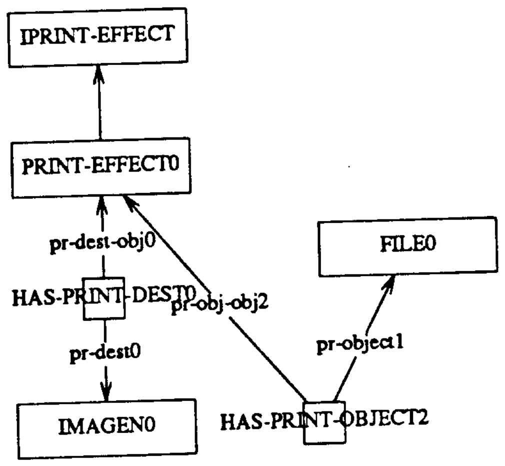


Figure 26
Input to **UCPlanner**

7.3.1. Goal Detection

UCPlanner is called by **UCEgo** with one argument, which is a list of goals. In this example, this is a list of only one goal. **UCPlanner** is passed the goal of printing a file (file1) on the imagen. **UCPlanner** then tries to detect background goals that were not part of the input. In this example, no background goals are detected. The following is a trace of **UCPlanner** during the goal detection phase.

```
Planner is passed:

(PRINT-EFFECT0)

(HAS-PRINT-OBJECT2 (pr-object1 = FILE0) (pr-obj-obj2 = PRINT-EFFECT0))

(HAS-PRINT-DEST0 (pr-dest0 = IMAGEN0) (pr-dest-obj0 = PRINT-EFFECT0))

(PRINT-ACTION0 (pr-effect0 = PRINT-EFFECT0)
               (cause0-0 = (ACTION8 (actor10 = *USER*)))
               (actor5 = *USER*))
```

7.3.2. Plan Selection

During the plan selection phase, **UCPlanner** first looks for stored plans associated with the user's individual goal. Here, the user's individual goal, is an individual goal of printing out file1 on the imagen.

The first thing that **UCPlanner** does during plan selection is to determine if the system already has a plan for this individual goal. In this particular case there is no associated plan for the individual goal of printing out file0. The system has a stored plan for a specific individual goal only in those cases where 1) UC has recently seen an instance of the same goal (e.g. in the context of a conversation) or 2) UC has specific information about a particular instance of a goal (e.g. a specific plan for printing a particular system file).

UCPlanner next looks at other goals in the hierarchy and determines if they have corresponding plans. The individual goal of printing out file1 is an instance of the goal of printing out any file on the imagen. This has the corresponding plan of using the lpr -Pip command with the file argument of the command being the name of the file to be printed. Here there is a specific goal and associated plan because the designer of the planning knowledge base determined that expert UNIX users had a stored plan for this individual goal. In other cases **UCPlanner** might have to search further up the hierarchy to find an appropriate stored plan.

If **UCPlanner** had searched further in the hierarchy above the goal of printing files on the imagen, it would have found the general goal of printing a file on any printer. The stored plan for this goal is to use the lpr command. Furthermore, the description of the plan indicates which arguments of the lpr command should be used, depending on the destination of the file. During the plan specification process, **UCPlanner** would specify that the lpr command be used with the argument of -Pip and the appropriate file argument. This is the same plan that was selected by looking at the stored plan of the more specific goal. However, to arrive at the plan using a more general goal, **UCPlanner** must do more work. The following is a trace of **UCPlanner** during the plan selection phase.

```
Looking for stored plan for PRINT-EFFECT0
Looking for specific stored-plan for this individual goal
Looking for plan for PRINT-EFFECT0
No stored plan for PRINT-EFFECT0 was found
Try all the parents to see if they have a plan
Looking for plan for IPRINT-EFFECT
UNIX-IPR-COMMAND is a plan for IPRINT-EFFECT
Making a new instance of UNIX-IPR-COMMAND
```

7.3.2.1. Plan Specification

During the Plan Specification phase, **UCPlanner** creates all the necessary **KODIAK** knowledge structures for the individual plan that will be communicated to the user. **UCPlanner** creates a new instance of the lpr command. The command argument is -Pip. The file argument is the name of the file to be printed. The individual file to be printed does not have a name, since that was not specified in the problem description by the user. Therefore, the file argument's value cannot point at the name of the file. Instead, **UCPlanner** creates a new HAS-NAME relation, sets the value of the named-object aspectual to the file object, and uses the name aspectual as a placeholder for the name of the file. The file argument's value is set to this aspectual, which represents the name of the file. This demonstrates an advantage of **KODIAK**: it allows the program to make assertions about the name of the file without knowing what the name is. The following is a trace of **UCPlanner** during the plan specification phase.

```
Making a new instance of UNIX-IPR-COMMAND
Filling in PRINT-EFFECT0 relation
Filling in aspectual ipr-file0 with value FILE0
Filling in aspectual UNIX-IPR-COMMAND-effect0 with value
PRINT-EFFECT0

Making a new instance of HAS-COMMAND-NAME3
Filling in the HAS-COMMAND-NAME-named-obj30 with
HAS-COMMAND-NAME30
Making a new instance of IPR-HAS-FORMAT
Making a new instance of IPR-FORMAT

Filling in PRINT-EFFECT0 relation
Making a new instance of HAS-FILE-NAME
Filling in the named-file15 with HAS-FILE-NAME15
Filling in aspectual ipr-file-arg0 with value file-name15
Filling in aspectual IPR-FORMAT-UNIX-COMMAND-FORMAT-step0
with value lpr -Pip
Making a new instance of PLANFOR3
Filling in the goals30 with PLANFOR30
Filling in the plan30 with PLANFOR30
```

7.3.3. Plan Projection

UCPlanner has specified a plan that is an instance of a stored plan, i.e., use the lpr -Pip command. Since the plan is a stored plan, and potential bugs in the plan are already known and identified, little plan projection is necessary. The only condition that is checked is whether the user has read permission on the file. No information has been supplied by the user about this permission so the default read permission is used. This default is that the user does have read permission.

7.3.4. Plan Evaluation

Since the plan is a stored plan, plan evaluation is also relatively easy. It has not been noted that the stored plan of using lpr -Pip has any bad side effects, or any other properties that would make this plan undesirable. Therefore, the plan evaluation mechanism returns this plan as the suggested plan, and **UCPlanner** passes this following information to the expression mechanism.

```
Planner returns:
PLANFOR30
UCPlanner produces:
(PLANFOR30 (goals30 = PRINT-EFFECT0)
           (plan30 =
                   (UNIX-IPR-COMMAND0
                    (ipr-file0 = FILE0)
                    (UNIX-IPR-COMMAND-effect0 =
                                               PRINT-EFFECT0))))
```

The planner's output is shown in Figure 27.

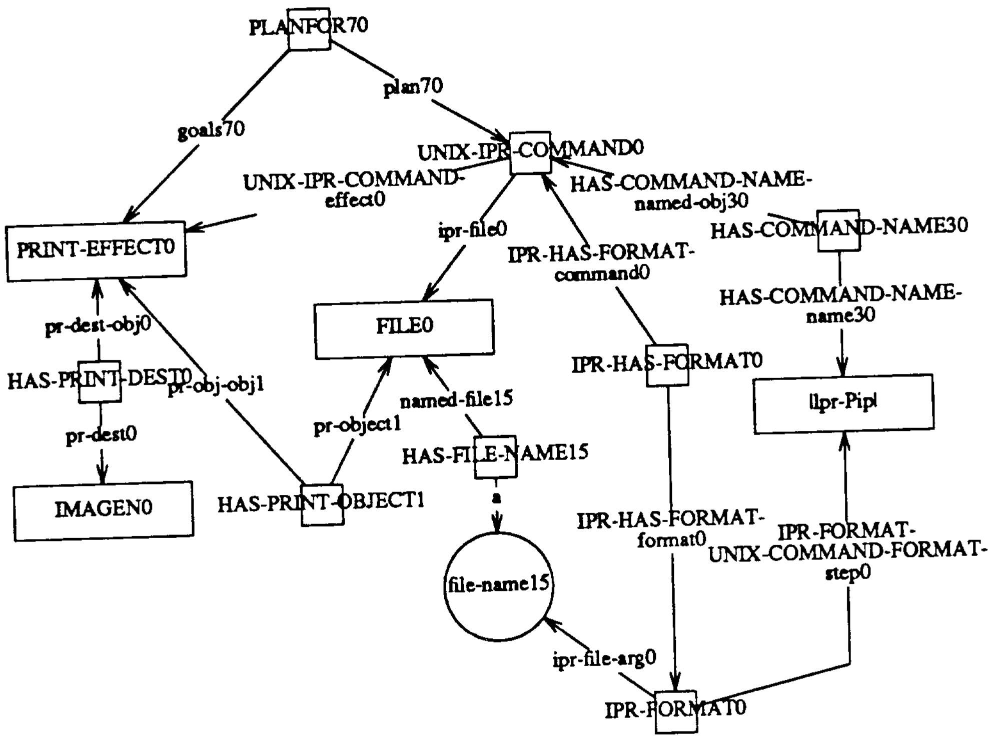


Figure 27
Output from UCPlanner

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 to answer 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, the concepts to be communicated to the user are edited to avoid telling the user something that the user already knows. 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 is ready for generation into natural language.

When **UCExpress** is passed a set of concepts to communicate to the user, the first stage of processing prunes them by marking extraneous concepts. The guiding principle is to avoid telling the user anything that the user already knows. Currently UC models two classes of information that the user may already know. The first is semantic information about the user's knowledge of UNIX-related facts. Such knowledge is modeled by UC's **User Model**, which was described in the section on **UCEgo**. More details can also be found in [Chin, 1986]. 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 UC's **User Model** deduces that the user already knows is marked and not communicated to the user.

Consider the following scenario from a UC session:

User:	How can I compact a file?
UC:	Use compact.

The conceptual answer passed to **UCExpress** is more complex than UC's answer above. If the conceptual answer were to be paraphrased into English, the result might 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.

UCExpress prunes the concepts corresponding to "compacting a file," since they are already in the conversational context. Also, the format of the compact command is pruned from UC's answer, since UC's **User Model** deduced that the user already knew the SIMPLE-FILE-FORMAT (the name of the command followed by the name of the file to be operated on). If the user were a novice, then UC could not assume that the user already knew this format, and would have provided the following answer, which includes an example of the format:

UC to novice: Use compact. For example, to compact the file foo, type 'compact foo'.

After pruning, UCExpress enters the formatting phase where it tries to apply different *expository formats* to express the concepts in a clearer manner. One such 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 for expressing general knowledge about complex (i.e., multi-step) procedures such as 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 procedure into an example. This involves stepping through the general procedure and transforming general information into specific instances. So UCExpress arbitrarily chooses specific values to replace general referents. Consider the following UC dialogue:

User: How can I change the read permission of a file?

UC: Use chmod.
For example, to remove group read permission from the file 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." These specific values must also be introduced to the user before their use. This can be done by using expressions like "the file named foo" before using "foo" by itself. Also care must be taken to ensure that any newly created values do not conflict (e.g. give different files different names), and that the new values are consistent (e.g. make sure the same file has the same name throughout the example). The key principle in producing examples is to be explicit.

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 has determined the proper answer, it calls UCExpress to express this to the user. The input to UCExpress is shown in Figure 28. This input conceptual network is the same as the plan for printing a file on the imagen that the '*(Pl produced, except with the addition of some information about who should say this to whom. This plan is in a form that is optimized for internal manipulation by UC components such as the UCPlanner, so it is not suitable for direct generation into natural language. 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" followed by concatenating "-P" with "ip" and followed by the name of the file to be printed on the imagen printer.

Such a direct translation is verbose and hard to understand. UCExpress is called to select a better format. The following trace illustrates the processing of UCExpress:

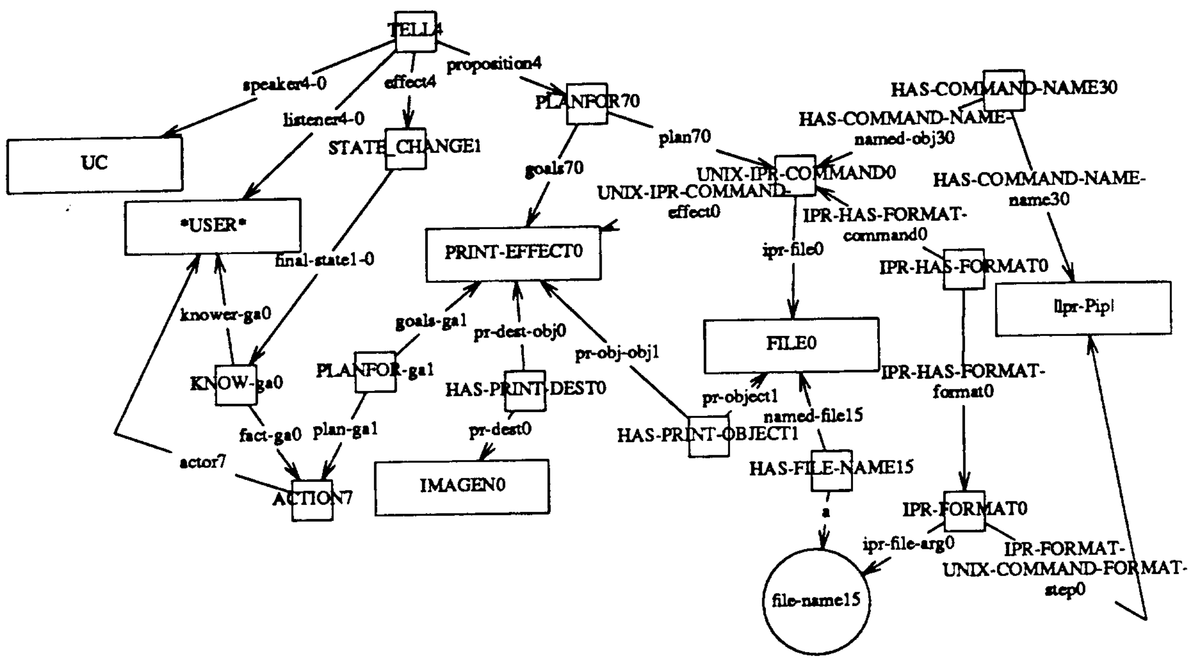


Figure 28
The input to UCExpress.

```
Express:  now expressing the PLANFOR,
(PLANFOR70 (goals70 = PRINT-EFFECT0)
        (plan70 = (UNIX-IPR-COMMAND0 (ipr-file0 = FILE0)
                                      (UNIX-IPR-COMMAND-effect0 =
                                        PRINT-EFFECT0))))

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 (speaker5-0 = UC)
           (listener5-0 = *USER*)
           (proposition5 =
              (EXAMPLE0
                  (example0 =
                     (PLANFOR31 (goals31 = PRINT-EFFECT1)
                                   (plan31 =
                                      (TYPE-ACTION0
                                          (actor0-0 = *USER*)
(type-string0 =
   (IPR-FORMAT0
      (ipr-file-arg0 =
         (aspectual= filename15 of
                        HAS-FILE-NAME15))
      (IPR-FORMAT-UNIX-COMMAND-FORMAT-step0 =
         llpr -Pipl))))))))))

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

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 traversing the input conceptual network, UCExpress runs into the command-format (IPR-FORMAT0), which is not pruned. Command-formats are checked to see if they are completely specified. Here, it is incomplete, so UCExpress creates an example to explain the format. Creating the example is part of UCExpress's formatting phase, which here is interleaved with the pruning phase.

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. Also, to introduce the name before its use, the phrase "to print the file named foo on the imagen printer," was reinstated in the example part of the answer. The complete example is then turned into the proposition part of a TELL (TELL5 in the trace).

Continuing with the pruning phase, UCExpress prunes the concept for "printing a file on the imagen printer" (PRINT-EFFECT0) since that exact concept is already in the context. The rest of the conceptual network passes without any pruning. Figure 29 shows the conceptual network after pruning and the addition of an example. The pruned and augmented conceptual network is next passed to the generator, which produces the following English output:

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

If the user was judged to be at least a beginner in experience, then the command-format would also have been pruned. This is because UC's User Model indicates that users at least beginner level can be assumed to know that part of the command format. Without a need to express the command-format, UCExpress would not have created an example and the final conceptual network passed to the generator would lack the example.

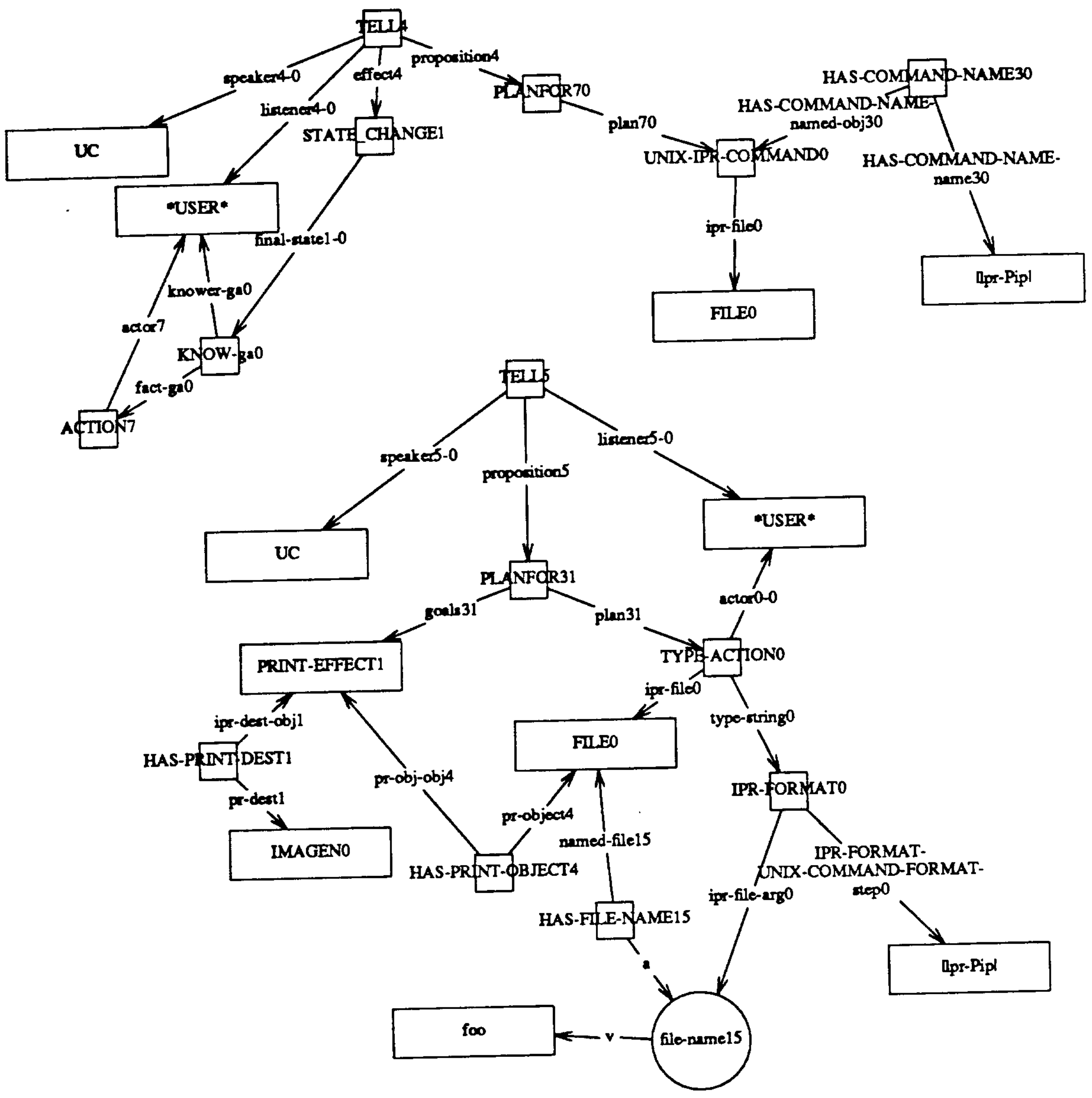


Figure 29
The output of UCExpress.

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 a 'pattern' that represents some linguistic output to a concept. 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, with the pattern '(gen plan)' for the case when 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, 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 the relevant knowledge structure is given in Figure 30. A trace produced while generating this output is given in Figure 31.

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.' PLANFOR is a category for PLANFOR70, so the pattern for PLANFOR is retrieved. Since goals70 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 plan aspectual for PLANFOR70 is plan70. Its value, UNIX-IPR-COMMAND0, is sent to 'gen.' The pattern for this concept is found under EXECUTE-FILE-COMMAND and is 'use (name-of-con).' This pattern causes the generator to look for a name for the concept being generated, here UNIX-IPR-COMMAND0. It does this by looking for a relation that has the concept as a value for an aspectual dominated by 'named-obj.' It then finds the name as the value of the 'name' aspectual of the relation. Here, HAS-COMMAND-NAME15 is the relation, and the value found 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 example).' This sets up a recursive call to gen with the value of example0 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 'goals' is PRINT-EFFECT1. The pattern is found under LAS-PR-EFFECT, and is 'print (rel-obj HAS-D-OBJECT) on the (rel HAS-PRINT-DEST).'

The rel-obj causes the generator first to find a relation that 1) is below HAS-D-OBJECT in the hierarchy, and 2) has an aspectual whose value is the concept being generated. The concept being generated is PRINT-EFFECT1, and the relation found is HAS-PRINT-OBJECT4. Then rel-obj causes the generator to generate the value of the other aspectual of this relation, and generate it as an object. The value here is FILE0. In

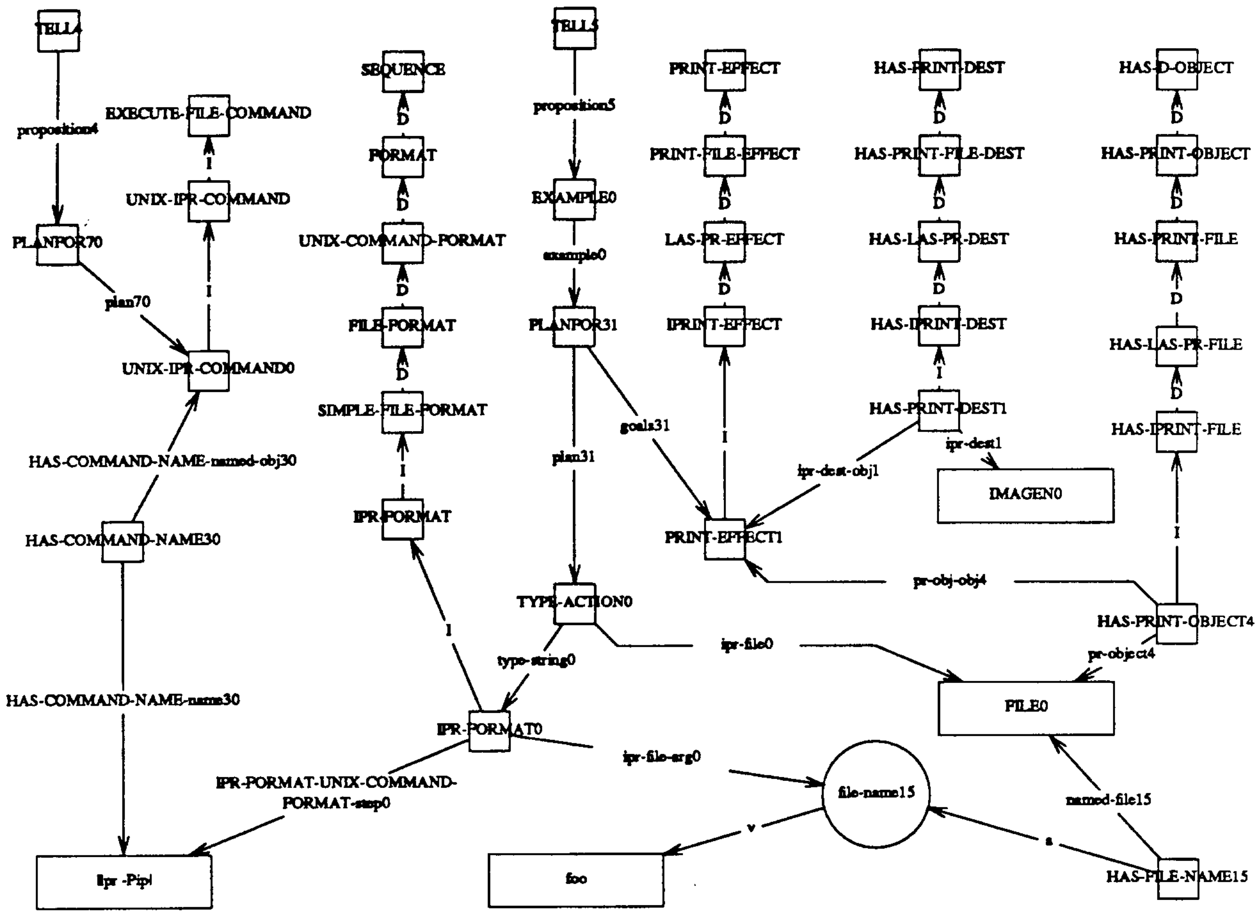


Figure 30
Knowledge state for example question.

generating a concept as an object, it is determined if the concept has a name; if it does, something of the form 'the x named y' is output. Otherwise, something of the form 'an x' is output. Here, FILE0 has the name 'foo.'

The other part of the 'goals' to be output is from the second half of the pattern: 'on the (rel HAS-PRINT-DEST).' Rel is like rel-obj except the concept to be output is not treated as an object. Here, the relation found is HAS-PRINT-DEST1 and the value is IMAGEN0. The pattern for IMAGEN0 is just 'imagen.'

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 IPR-FORMAT0. The pattern is from SEQUENCE, and is '(gen step) (gen next).' Here, the step is 'lpr -Pip,' and the next is the name of the file, 'foo.' 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 (name-of-con))
Phrase derived from (name-of-con) is: (lpr -Pip)
Phrase derived from (gen plan) is: (use lpr -Pip)

Use lpr -Pip.

Pattern for EXAMPLE0 is: (for example comma (gen example))
Value for example is: PLANFOR31
Pattern for PLANFOR31 is: (to (gen goals) comma (gen plan))
Value for goals is: PRINT-EFFECT1
Pattern for PRINT-EFFECT1 is:
(print (rel-obj HAS-D-OBJECT) on the (rel HAS-PRINT-DEST))
Value for HAS-D-OBJECT is: FILE0
Pattern for FILE0 is: (file)
Phrase derived from (rel-obj HAS-D-OBJECT) is: (the file named foo)
Value for HAS-PRINT-DEST is: IMAGEN0
Pattern for IMAGEN0 is: (imagen)
Phrase derived from (rel HAS-PRINT-DEST) is: (imagen)
Phrase derived from (gen goals) is:
(print the file named foo on the imagen)
Value for plan is: TYPE-ACTION0
Pattern for TYPE-ACTION0 is: (type (gen type-string))
Value for type-string is: IPR-FORMAT0
Pattern for IPR-FORMAT0 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 lpr -Pip foo)
Phrase derived from (gen example) is:
(to print the file named foo on the imagen comma type lpr -Pip foo)

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

Figure 31

UCGen trace for the example question.

For example, to print the file named foo on the imagen, 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 implementors 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. Our 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 and Hendrix, 1980]. **UCTeacher** [Martin, 1985] is the system being built to provide this facility for UC. In this report we will focus exclusively on **UCTeacher**'s acquisition of UNIX domain knowledge. The UNIX expert can engage **UCTeacher** in a mixed-initiative dialogue to add more information to the UC's knowledge base. The following are some working examples from **UCTeacher**:

1)

User:	There is a new command.
UCT:	What is its name?
User:	sort
UCT:	What is it used for?
User:	It is used to sort files.
UCT:	You mean that typing 'sort filename' will sort the file filename.
User:	Yes.

2)

User:	Use the rm command to delete files.
UCT:	You mean that typing 'rm filename' will delete the file named filename.
User:	Yes.

3)

User: Use compact to compress a file.

UCT: You mean that typing 'compact filename' will compress the file named filename.

User: Yes

The above examples illustrate that **UCTeacher** has access to and is using much knowledge to guide the acquisition dialogue. **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 meta-knowledge will be described in more detail in the next section. This use of knowledge plus meta-knowledge makes the knowledge acquisition process a knowledge-based task. This approach to knowledge acquisition was first used by Davis [1976] in his TEIRESIAS system.

The system 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 things. Examples (2) and (3) show the system making full use of the expert's utterance to avoid asking redundant questions.

10.3. A Detailed Example

Consider the following sequence of interactions:

4)

User: How can I print a file?

UC: 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. In the following dialogue an expert enters **UCTeacher** and provides the necessary information.

User: There is a new command.

UCT: What is its name?

User: lpr

UCT: What is it used for?

User: It is used to print files.

UCT: Where does the output go?

User: the lineprinter

UCT: You mean that typing 'lpr filename' will print the file filename on the lineprinter.

User: Yes.

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

User: How can I print a file?

UC: To print a file use lpr. For example, to print the file named foo, type lpr foo.

ALANA is used to parse the initial utterance by the expert. An instantiation of the concept EXECUTE-UNIX-COMMAND is then passed to UCTeacher. The main operation that UCTeacher performs is the instantiation of relations connected to the concept being learned. Therefore, the first step that must be performed is to collect from the inheritance hierarchy of UC's knowledge base all the relations that EXECUTE-UNIX-COMMAND participates in. Here, the following relations are found: HAS-COMMAND-NAME, EFFECT, OPERATED-FILE and HAS-FORMAT. Figure 32 illustrates some of the knowledge used by UCTeacher to learn about commands.

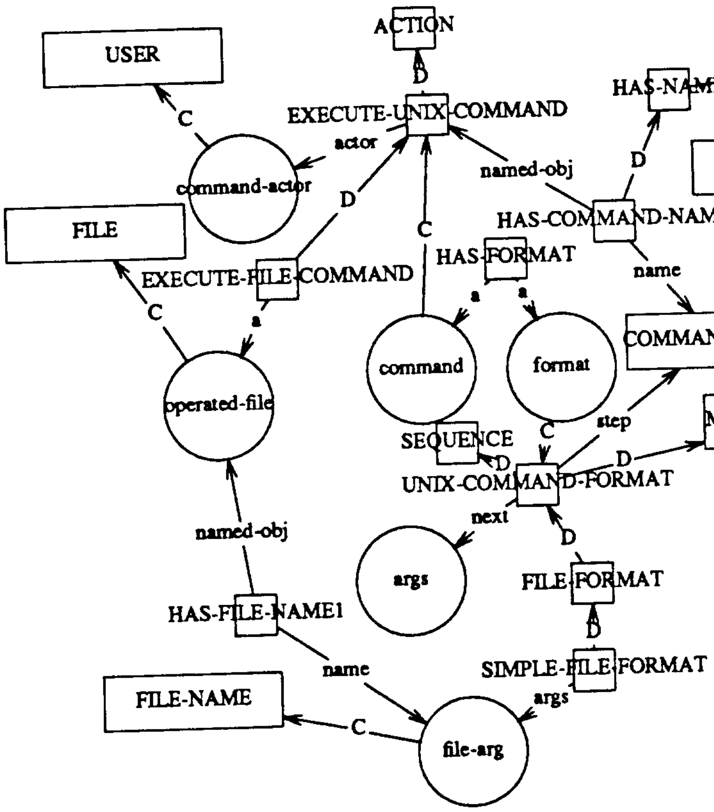


Figure 32
Knowledge for learning about commands

Because of the proliferation of concepts and relations in KODIAK not all the necessary relations are directly related to the concept being learned. Some important facts are related only indirectly to the main concept. An example of this is the HAS-FILE-NAME relation connected to the OPERATED-FILE. The possibility of wide distribution of concepts in the network led to the development of the *Minimal Aspectual Set* (MAS). The MAS constitutes 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.

The system's first question in this example is an attempt to find a value for the relation HAS-COMMAND-NAME. This is an illustration of the use of meta-knowledge. The piece of meta-knowledge used here is that the only way to fill in the value of a EXECUTE-UNIX-COMMAND-NAME is to ask the user. This fact can not be inferred from anything else about the command. Therefore in order for **UCTeacher** to be able to fill in a concept it must have an acquisition method associated with it. Asking the user is one such method; others are using defaults and inferring values from other concepts. These will be illustrated below. In the case where asking the user is the method, a partial phrase that is appropriate is included as part of the meta-knowledge of that concept. Here the name is read, instantiated and connected to the HAS-COMMAND-NAME relation. At this point **UCTeacher** calls the concretion mechanism to see if it is possible to concrete based on the information just acquired. An attempt to concrete is made after each new relation is instantiated. Here no concretion is made.

The second question from **UCTeacher** is an attempt to instantiate the EFFECT relation from the EXECUTE-UNIX-COMMAND concept. Again the user is queried and the relation instantiated and given the value PRINT-FILE-EFFECT. Again a concretion is attempted. Here the concretion is successful; EXECUTE-UNIX-COMMAND is concreted to EXECUTE-PRINT-FILE-COMMAND. 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. This leads to the attempt to fill in the HAS-PRINT-FILE-DEST relation by asking the last question.

The next relation that gets instantiated is the HAS-FORMAT relation. Note that **UCTeacher** does not query the expert for the argument format. This is because the meta-knowledge about this relation specifies that the relation can be defaulted to SIMPLE-FILE-FORMAT based on the value of the EFFECT relation. This reflects the fact that the argument format of a command can in general be inferred from its effect, which is already known. The rest of the relations are inferred in a similar manner.

The next operation **UCTeacher** performs is to generate a statement reflecting what it has just learned. This gives the expert the opportunity to correct any incorrect inferences made by the system. An example of this might be a simple file command that did not take the usual argument format. **UCTeacher** would default the format. At this point the expert could correct it.

The final phase of **UCTeacher**'s processing is to make the changes permanent. Each absolute, relation and aspectual 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.**

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. Most likely, 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 kind of marker-passing algorithm 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 question:

What does ls -v do?

This question is problematic because a "literal" response to it might be "It lists the contents of the current directory." This response is possible because there is no "-v" option to the "ls" command, and it is a characteristic of this command that it ignores options it does not recognize.

A much better response would be "There is no -v option to the ls command." To produce this 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.

This 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.

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. Appendix: UC KODIAK Diagrams

This appendix contains a small sample of the KODIAK diagrams used to define representations in UC. Some of these diagrams define very general concepts such as EVENTs and STATEs; other diagrams describe concepts which are very specific to UC, such as file protection, and specific UNIX commands.

Each of the diagrams includes a short description of what it represents. Most of the concept names in UC's version of KODIAK are reasonably mnemonic, so the reader should be able to decipher the diagrams that describe them.

Note that the meaning of the arcs in the diagrams is context sensitive. This ambiguity is not a problem in the implementation of UC since the interpretation of an arc is clearly determined by its context. In fact, the KODIAK representations in UC are entered as diagrams using a graphic editor, which translates them into an internal form.

The meanings of nodes and arcs in the diagrams that follow is summarized in the translation legend below.

NODES

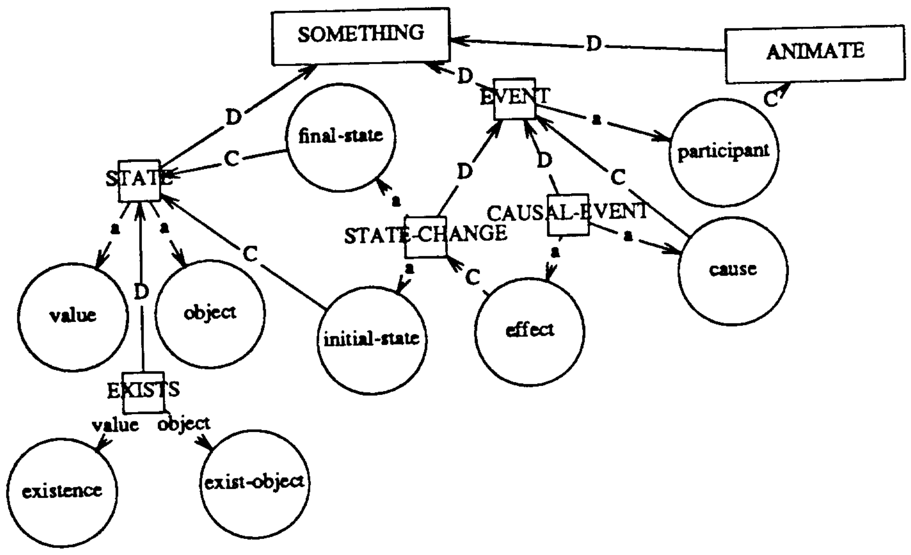
REPRESENTATION	MEANING
rectangle	an ABSOLUTE
small box	a RELATION
circle	an aspectual
double-circle	special meanings
$\boxed{\text{conceptN}}$, N an integer	$\boxed{\text{conceptN}}$ is an instance of $\boxed{\text{concept}}$

ARCS

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

$\square \xrightarrow{a} \bigcirc$	$\bigcirc$ is an aspectual of $\square$
$\boxed{\text{dominated}} \xrightarrow{D} \boxed{\text{dominator}}$	$\boxed{\text{dominator}}$ dominates $\boxed{\text{dominated}}$
$\boxed{\text{instance}} \xrightarrow{I} \boxed{\text{category}}$	$\boxed{\text{instance}}$ is an instance of $\boxed{\text{category}}$
$\bigcirc \xrightarrow{C} \boxed{\text{constraint}}$	$\bigcirc$ is constrained to be of type $\boxed{\text{constraint}}$
$\bigcirc \xrightarrow{v} \boxed{\text{value}}$	the value of aspectual $\bigcirc$ is $\boxed{\text{value}}$
$\boxed{\text{instanceN}} \xrightarrow{\mathit{aspectualN}} \boxed{\text{value}}$	*aspectualN* is an aspectual of $\boxed{\text{instanceN}}$; the value of *aspectualN* is $\boxed{\text{value}}$; and *aspectualN* plays the role of *aspectual* which must be defined as an aspectual of some relation dominating $\boxed{\text{instanceN}}$
$\boxed{\text{instanceN}} \xrightarrow{\mathit{aspectual}} \boxed{\text{value}}$	where *aspectual* is an aspectual of some relation dominating $\boxed{\text{instanceN}}$, the corresponding aspectual of $\boxed{\text{instanceN}}$ has the value $\boxed{\text{value}}$
$\square \xrightarrow{\mathit{aspectual}} \bigcirc$	$\bigcirc$ is an aspectual of $\square$, and $\bigcirc$ plays the role of *aspectual* which must be defined as an aspectual of some relation dominating $\square$
$\square \xrightarrow{\mathit{aspectual}} \boxed{\text{value}}$	where *aspectual* is an aspectual of some relation dominating $\square$, the corresponding aspectual of $\square$ has the value $\boxed{\text{value}}$

Translation Legend



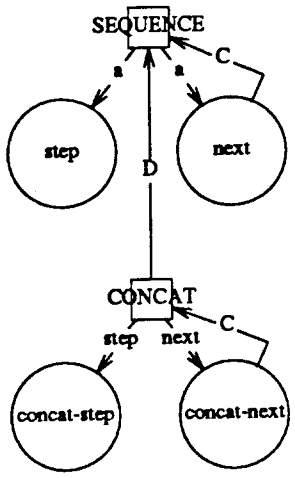
This diagram defines the concepts EVENT, STATE, STATE-CHANGE, and CAUSAL-EVENT.

A STATE has aspectuals object and value.

An EVENT has a participant aspectual which is constrained to be filled by an ANIMATE.

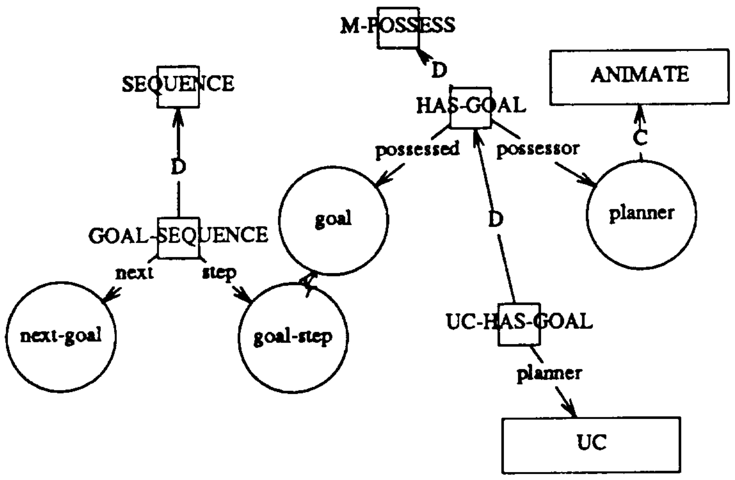
A STATE-CHANGE is an EVENT with an initial-state and a final-state. A subtype of EVENT is CAUSAL-EVENT which has a cause and an effect.

An example of STATE is EXISTS in which the object is an exist-object and the value is existence.



The notion of a SEQUENCE is defined here.

A SEQUENCE has two aspectuals, step and next. The next must be a SEQUENCE in its own right. A specialization of SEQUENCE is a CONCAT which is used to represent concatenations.

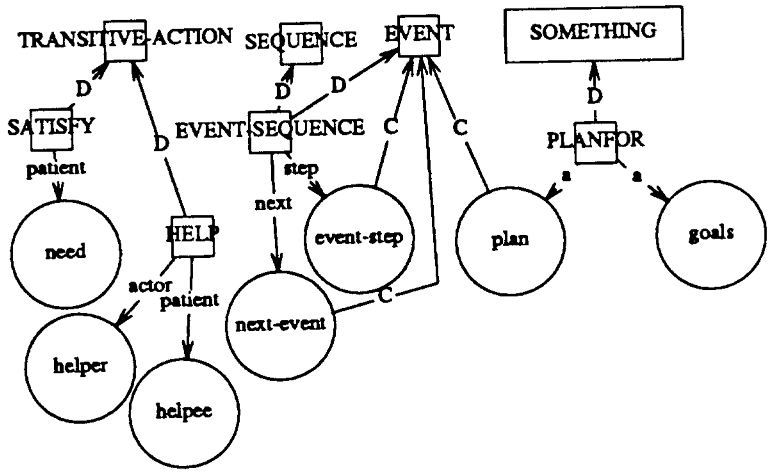


This diagram defines goals and GOAL-SEQUENCEs.

HAS-GOAL is a type of mental possession where the possessor is the planner and the possessed is the goal. A goal is an aspectual of the HAS-GOAL relation since a goal cannot exist independently of the ANIMATE that has the goal.

A GOAL-SEQUENCE is a type of SEQUENCE where the goal-step is constrained to be something which plays the role of a goal.

UC-HAS-GOAL is a type of HAS-GOAL in which the planner is UC.



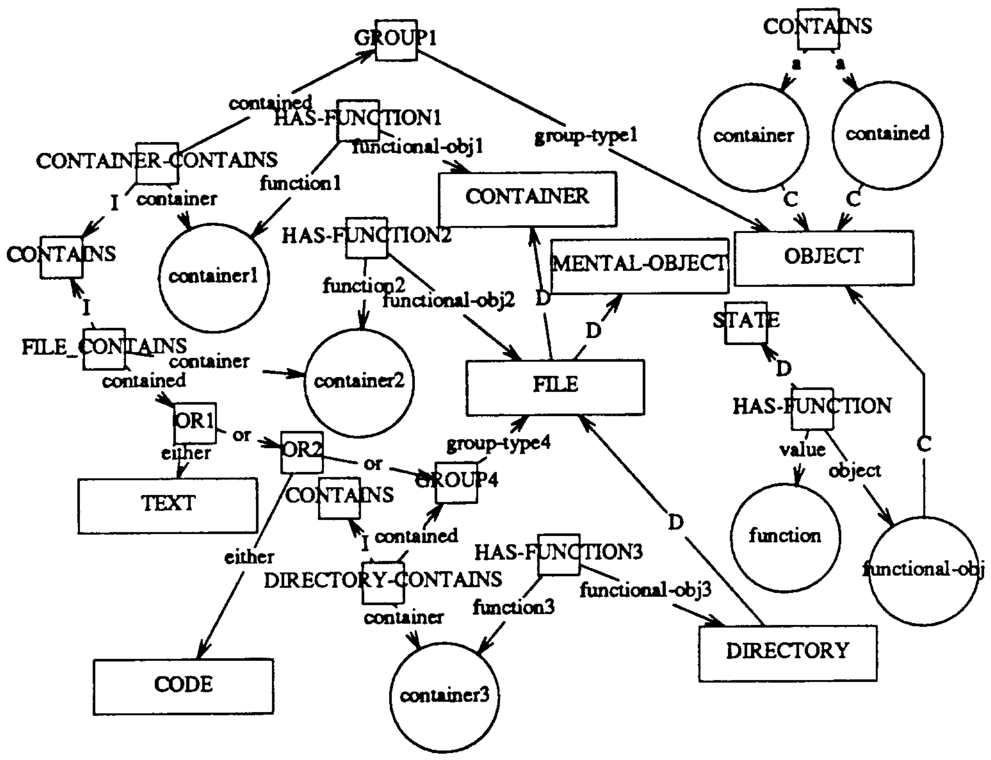
This diagram describes the concepts EVENT-SEQUENCE, PLANFOR, SATISFY, and HELP.

A PLANFOR is a relation with the aspectuals plan and goals.

A plan is constrained to be an EVENT (actually a HYPOTHETICAL EVENT since the EVENTs in plans generally have not occurred in actuality).

An EVENT-SEQUENCE is both a SEQUENCE and an EVENT where both the event step and the next-event are constrained to also be EVENTs.

HELP and SATISFY are types of TRANSITIVE-ACTIONs, which are defined in a later diagram.

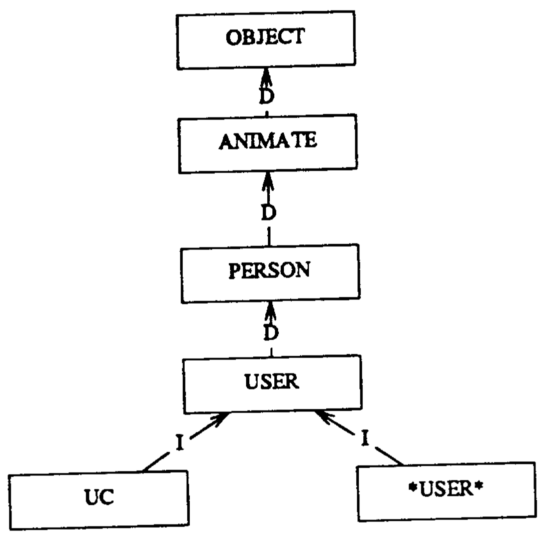


This diagram defines the ideas of FILE, DIRECTORY, CONTAINS, and HAS-FUNCTION.

A FILE is a CONTAINER and a MENTAL-OBJECT.

The function of a CONTAINER is to serve as the container of a CONTAINS. The function of a FILE is as the container for TEXT, CODE, or GROUPs of FILEs.

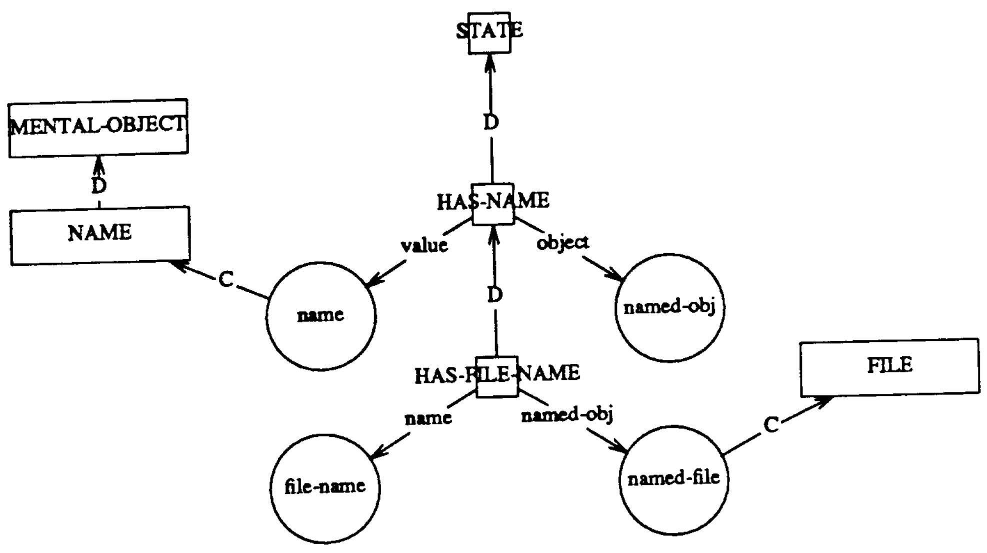
A DIRECTORY is type of FILE which has the function of only containing GROUPs of FILEs.



This diagram describes what a USER is.

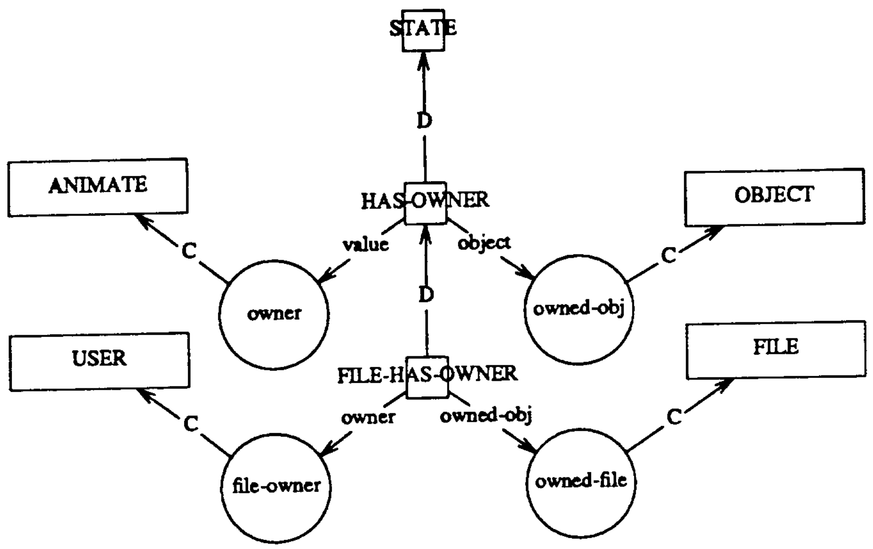
A USER is a type of PERSON, for example UC and *USER*.

A PERSON is a type of ANIMATE which is a type of OBJECT.



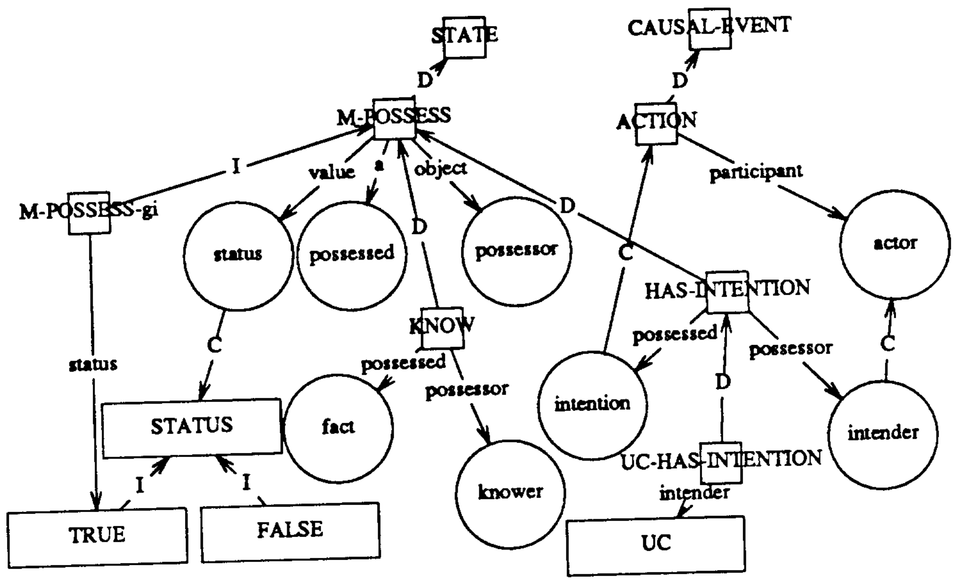
This diagram describes the relation between objects and names.

HAS-NAME is a type of STATE in which the value is name and the object is a named-object. The name is constrained to be a NAME which is a type of MENTAL-OBJECT. A subtype of HAS-NAME is HAS-FILE-NAME where the named-object is a named-file and the name is a file-name which can only be filled by a FILE.



This diagram defines ownership.

HAS-OWNER is a type of STATE where the owner is constrained to be an ANIMATE and the owned-obj is constrained to be an OBJECT. Note that owner is only an aspectual since an owner does not make sense except in relation to an object that the owner owns. A subtype of HAS-OWNER is FILE-HAS-OWNER in which the owner is constrained to be a USER and the owned-file is constrained to be a FILE.

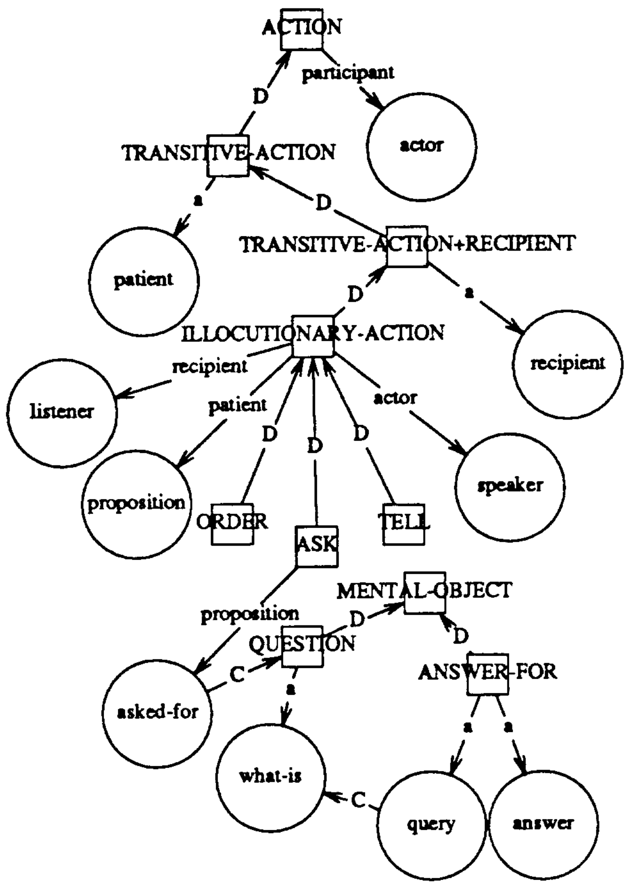


Herein is described the ideas of mental-possession, action, intention, and knowing.

M-POSSESS is a STATE which describes the possession by a possessor of the possessed object with value (the status aspectual) TRUE or FALSE. The generic M-POSSESS, M-POSSESS-gi, has the status TRUE. A subtype of M-POSSESS is KNOW where the possessed is a fact and the possessor is the knower.

An ACTION is a type of CAUSAL-EVENT where the participant is an actor.

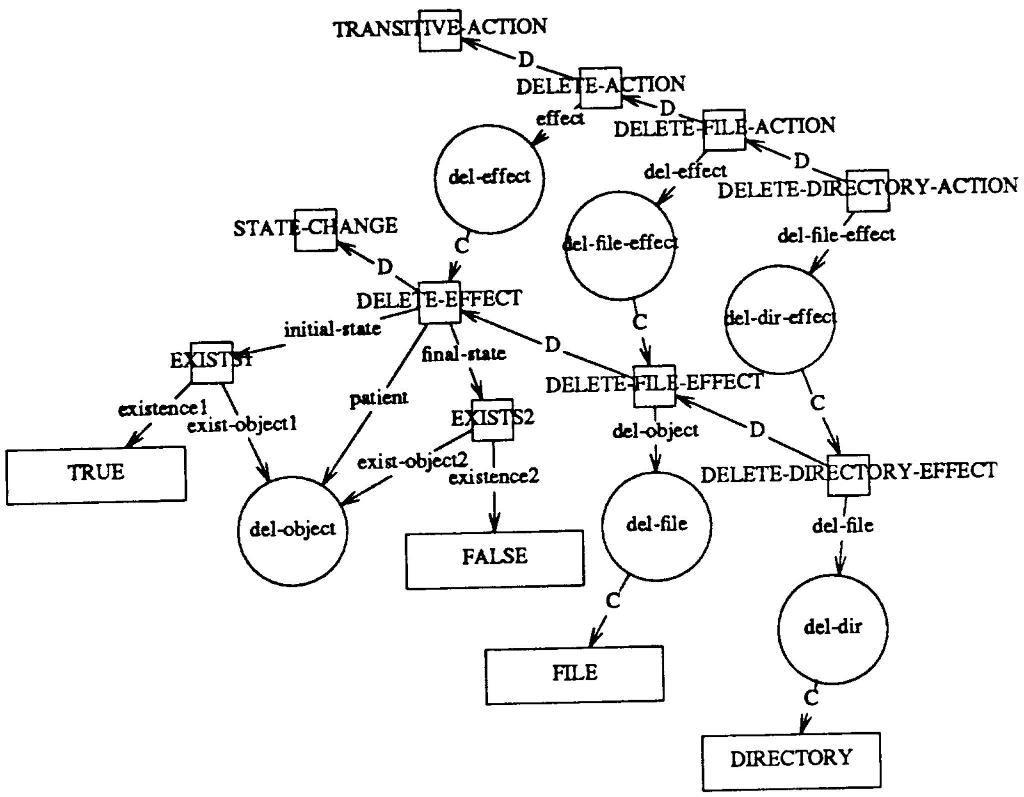
HAS-INTENTION is a type of M-POSSESS where the intender is constrained to be an actor and the intention is constrained to be an ACTION. A subtype of HAS-INTENTION is UC-HAS-INTENTION where the intender is UC.



This describes different types of ACTIONs including TELLs.

A TRANSITIVE-ACTION is an ACTION with a patient. A subtype of TRANSITIVE-ACTION is a TRANSITIVE-ACTION+RECIPIENT which also has a recipient.

An ILLOCUTIONARY-ACTION is a kind of TRANSITIVE-ACTION+RECIPIENT where the recipient is a listener, the patient is a proposition, and the actor is a speaker. Examples of ILLOCUTIONARY-ACTIONs include TELLs, ASKs, and ORDERs. In the ASK, the proposition is an asked-for which is constrained to be a QUESTION (a type of MENTAL-OBJECT).

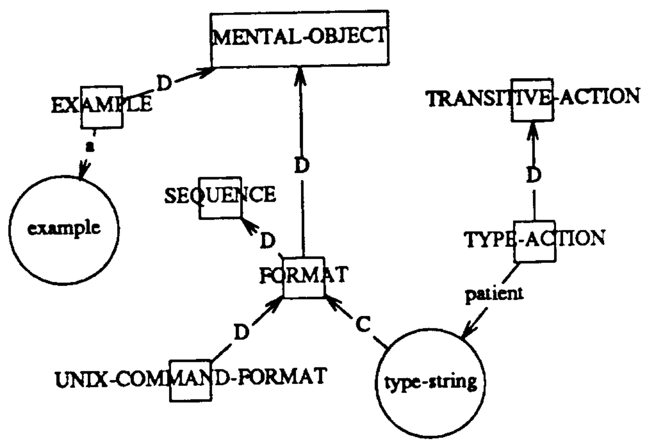


This defines deleting.

DELETE-ACTION is a kind of TRANSITIVE-ACTION where the del-effect must be filled by a DELETE-EFFECT. A type of DELETE-ACTION is DELETE-FILE-ACTION where the del-effect is constrained to be a DELETE-FILE-EFFECT.

A DELETE-EFFECT is a STATE-CHANGE where the initial-state is an EXISTS and the final-state is a does not EXISTS. The exist-objects of the EXISTS are required to be the patient of the DELETE-EFFECT. A type of DELETE-EFFECT is DELETE-FILE-EFFECT with the del-object constrained to be a FILE.

DELETE-DIRECTORY-ACTION and DELETE-DIRECTORY-EFFECT are analogous concepts for DIRECTORYs instead of FILEs.

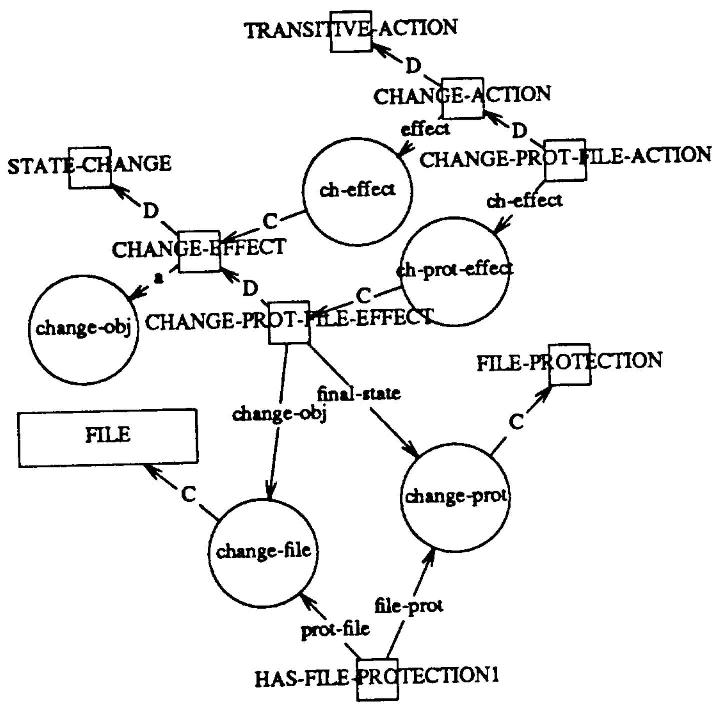


This diagram has the definitions for EXAMPLE, FORMAT and TYPE-ACTION.

An EXAMPLE is a MENTAL-OBJECT that has an example.

A FORMAT is both a type of MENTAL-OBJECT and a type of SEQUENCE. A sub-type of FORMAT is UNIX-COMMAND-FORMAT.

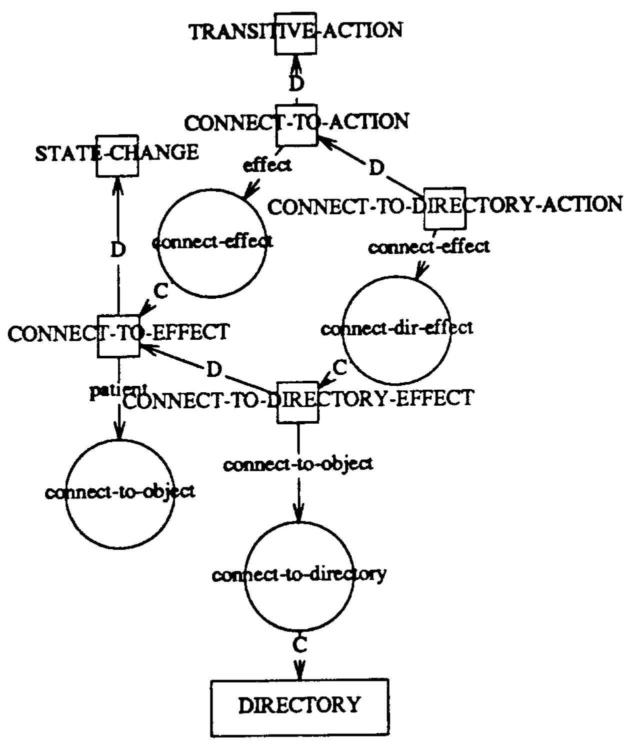
TYPE-ACTION is a TRANSITIVE-ACTION where the patient is a type-string that is constrained to be a FORMAT.



This diagram describes changing file protections.

A CHANGE-ACTION is a TRANSITIVE-ACTION with the ch-effect constrained to be a CHANGE-EFFECT. A subtype of CHANGE-ACTION is CHANGE-PROT-ACTION where the ch-prot-effect must be a CHANGE-PROT-EFFECT.

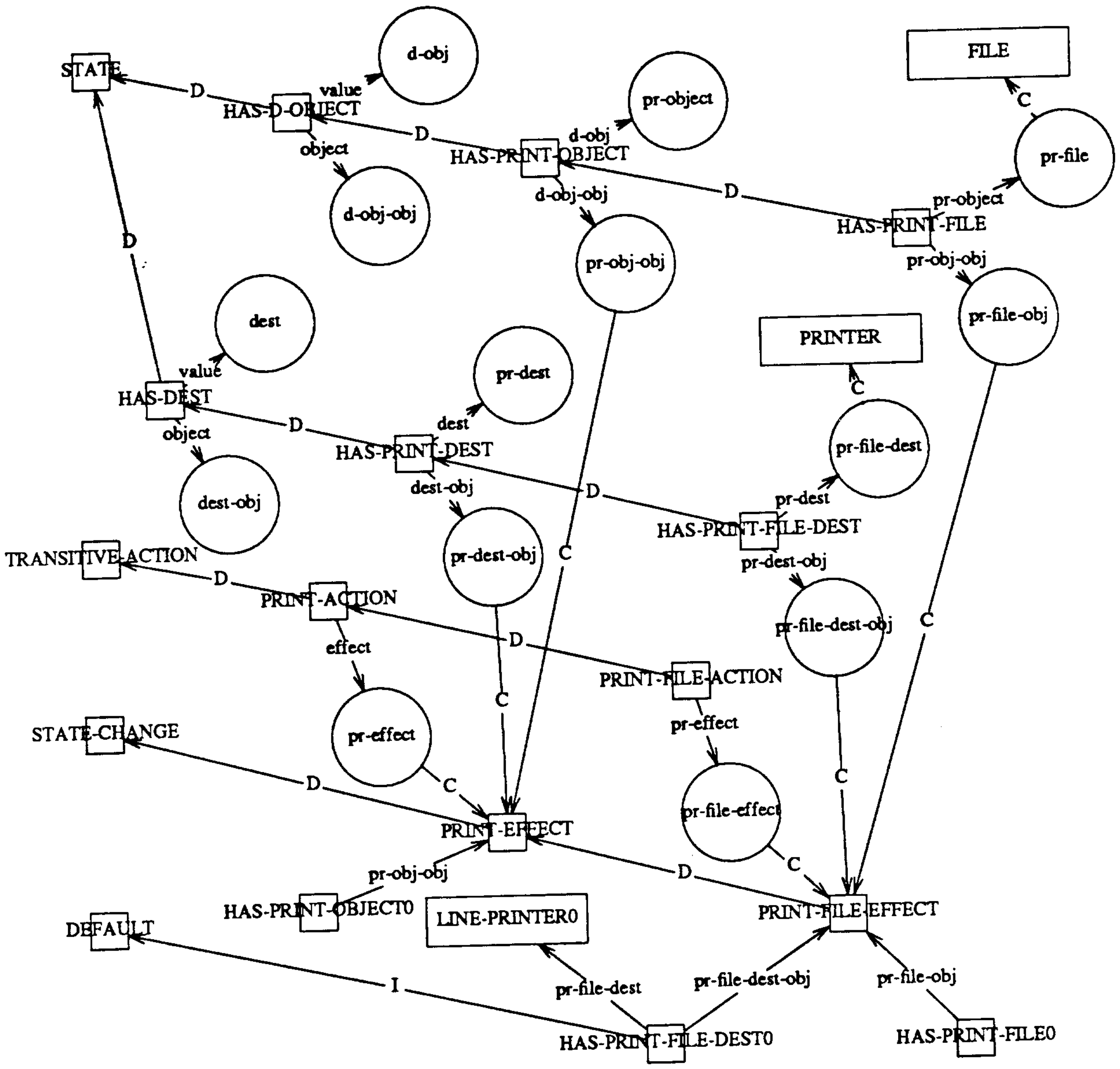
A CHANGE-EFFECT is a type of STATE-CHANGE with a change-obj aspectual. A subtype of CHANGE-EFFECT is CHANGE-PROT-EFFECT where the change-file must be a FILE and the change-prot must be a FILE-PROTECTION. Also, the change-file and the change-prot in the CHANGE-PROT-EFFECT are required to be related by a HAS-FILE-PROTECTION relation.



Here is defined the idea of connecting to something.

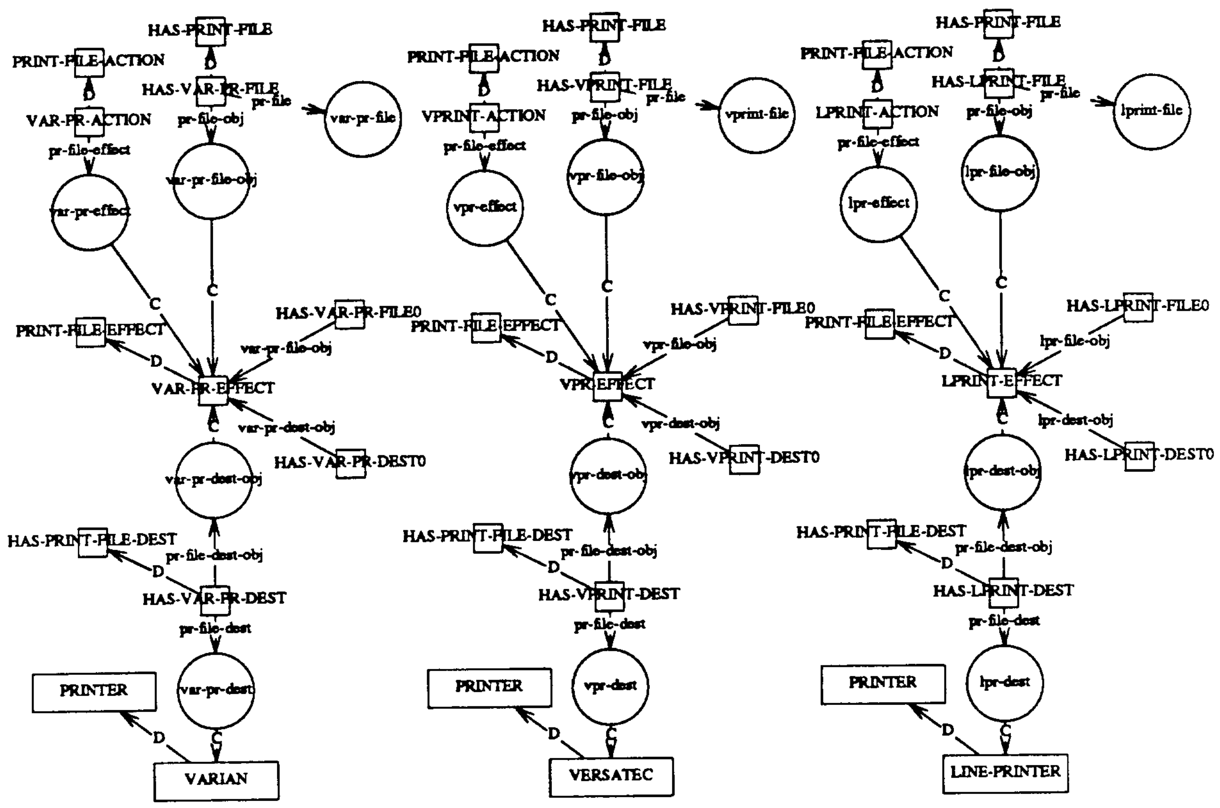
A CONNECT-TO-ACTION is a type of TRANSITIVE-ACTION with the connect-effect constrained to be a CONNECT-TO-EFFECT. A subtype of CONNECT-TO-ACTION is a CONNECT-TO-DIRECTORY-ACTION where the connect-dir-effect is constrained to be a CONNECT-TO-DIRECTORY-EFFECT.

A CONNECT-TO-EFFECT is a STATE-CHANGE with the patient being a connect-to-object. A subtype of CONNECT-TO-EFFECT is CONNECT-TO-DIRECTORY-EFFECT where the connect-to-directory is constrained to be a DIRECTORY.

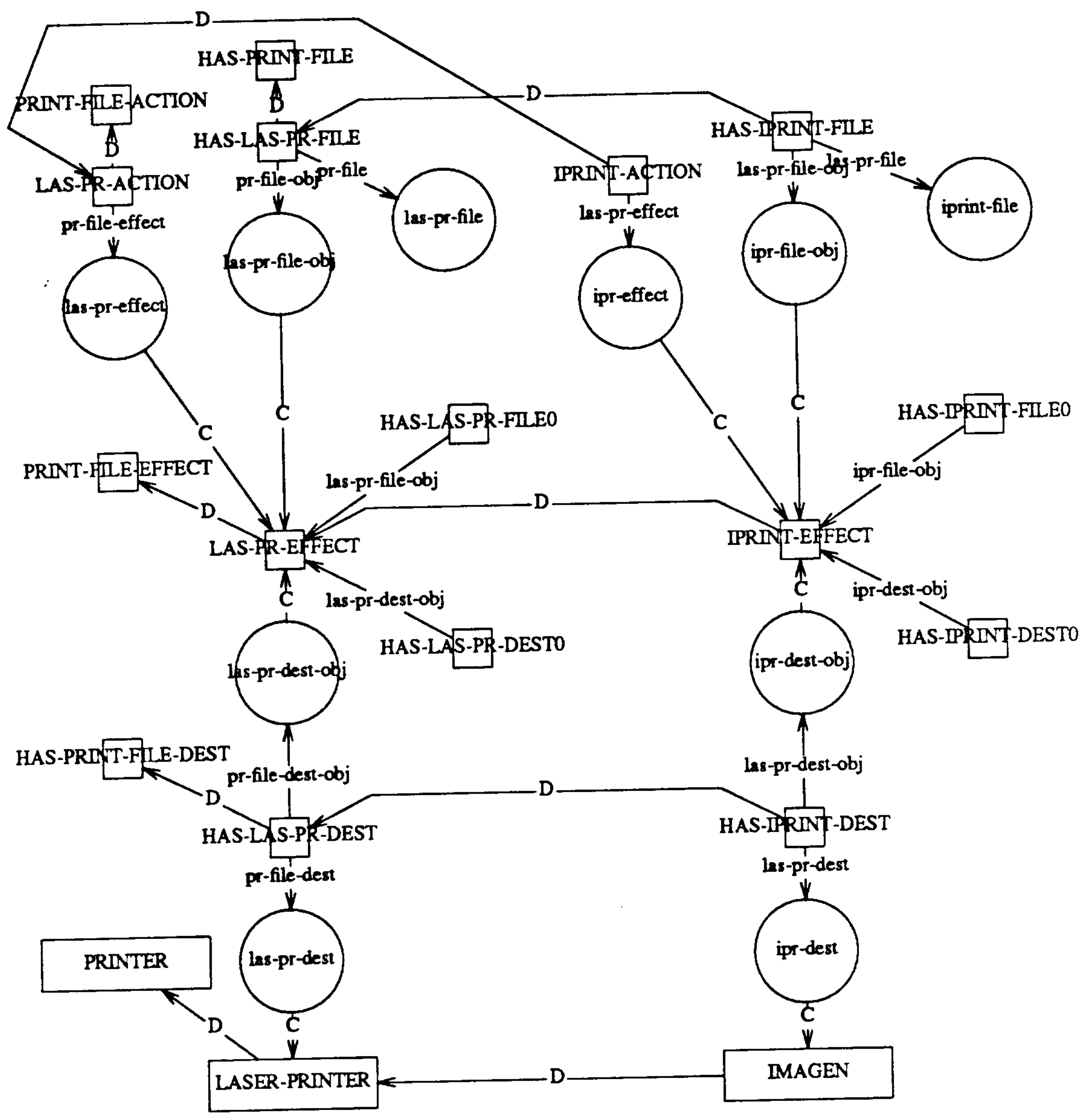


The next three diagrams describe printing.

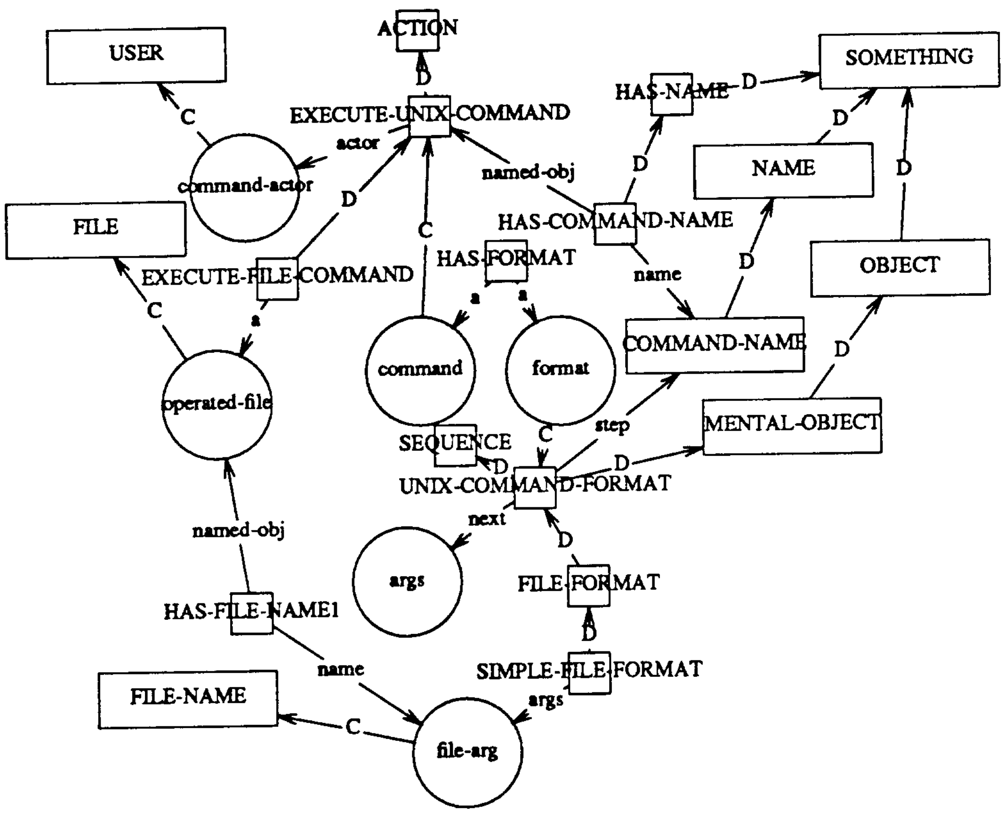
PRINT-ACTION, PRINT-FILE-ACTION, PRINT-EFFECT, and PRINT-FILE-EFFECT are defined here. Every PRINT-EFFECT is related to a pr-object by a HAS-PRINT-OBJECT relation. In the case of a PRINT-FILE-EFFECT, this pr-file must be a FILE.



This diagram describes the different subtypes of HAS-PRINT-FILEs, PRINT-FILE-EFFECTs, and HAS-PRINT-FILE-DESTs for different kinds of printers: VARIAN, VERSATEC, and LINE-PRINTER.



This diagram describes printing on a LASER-PRINTER and in particular on the IMAGEN laser-printer. See the previous diagram for definitions involving other types of printers.



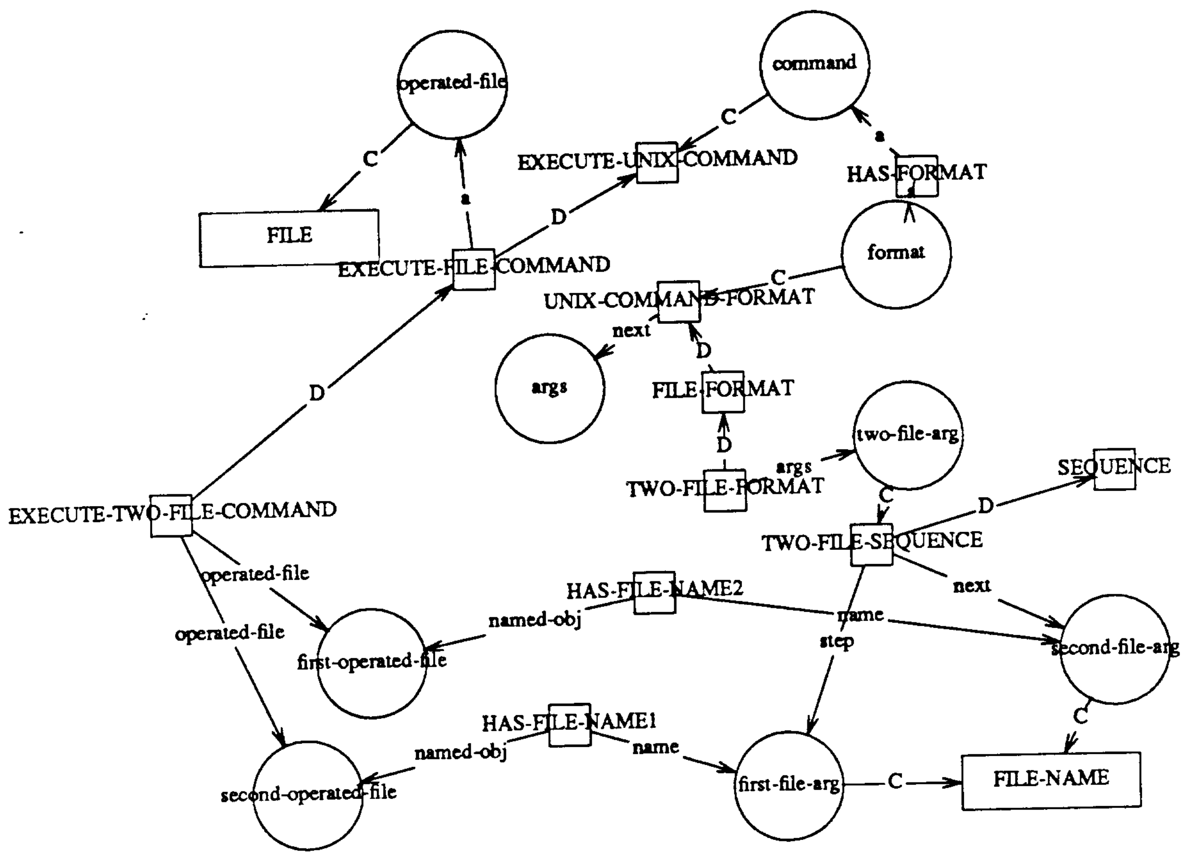
UNIX commands and simple command formats are defined here.

EXECUTE-UNIX-COMMAND is the ACTION of using a command. EXECUTE-FILE-COMMAND is a subtype which deals with FILEs.

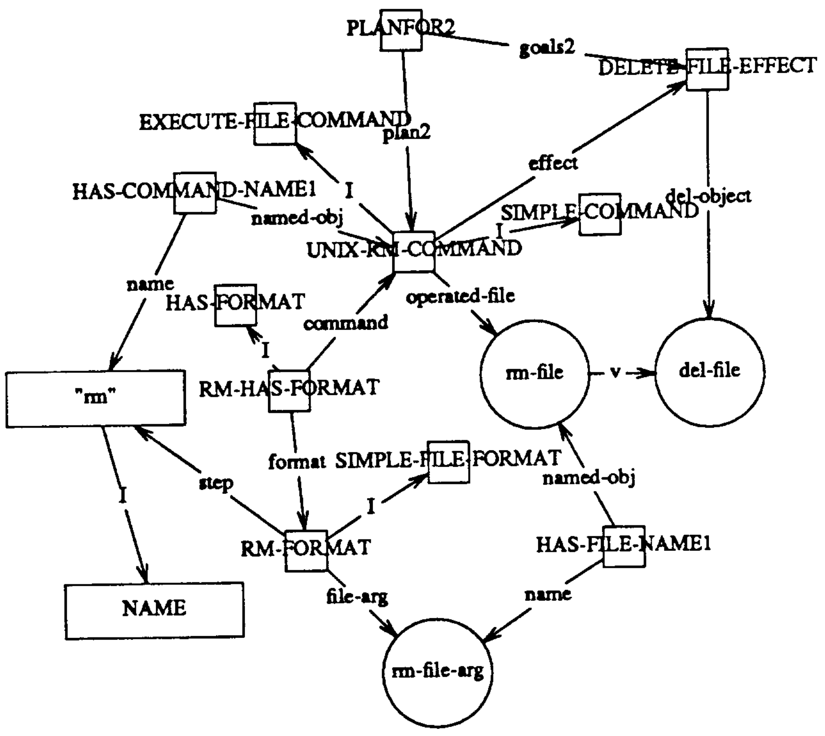
HAS-COMMAND-NAME relates an EXECUTE-UNIX-COMMAND to its COMMAND-NAME. HAS-FORMAT relates an EXECUTE-UNIX-COMMAND to its UNIX-COMMAND-FORMAT.

The UNIX-COMMAND-FORMAT is a SEQUENCE where the step is the COMMAND-NAME and next are the args.

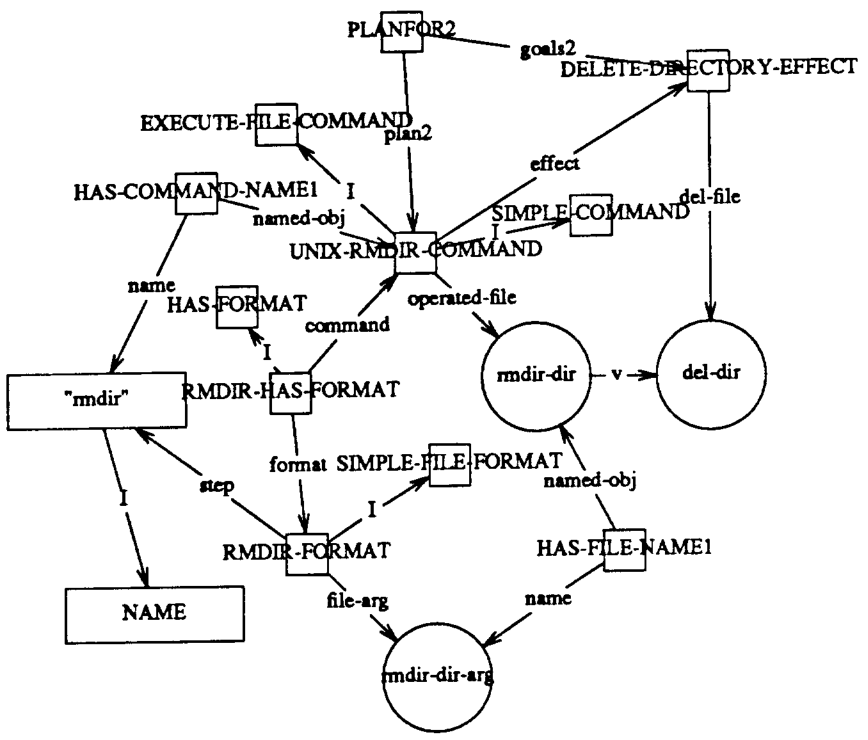
Specializations of UNIX-COMMAND-FORMATs include FILE-FORMATs and SIMPLE-FILE-FORMATs. The SIMPLE-FILE-FORMAT has a single argument, the file-arg, which is constrained to be a FILE-NAME.



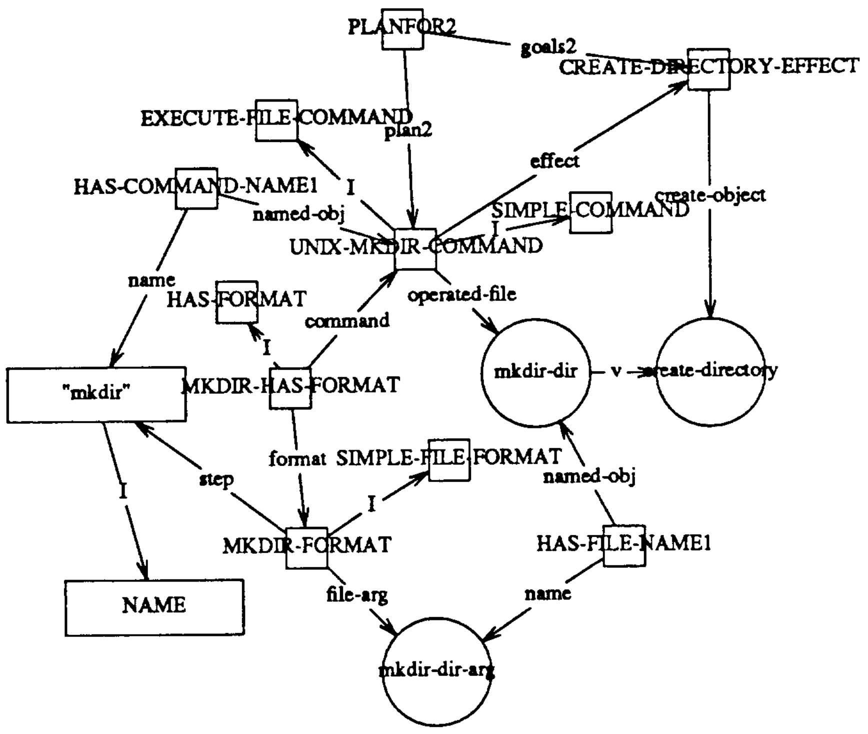
This diagram describes commands with two file arguments.



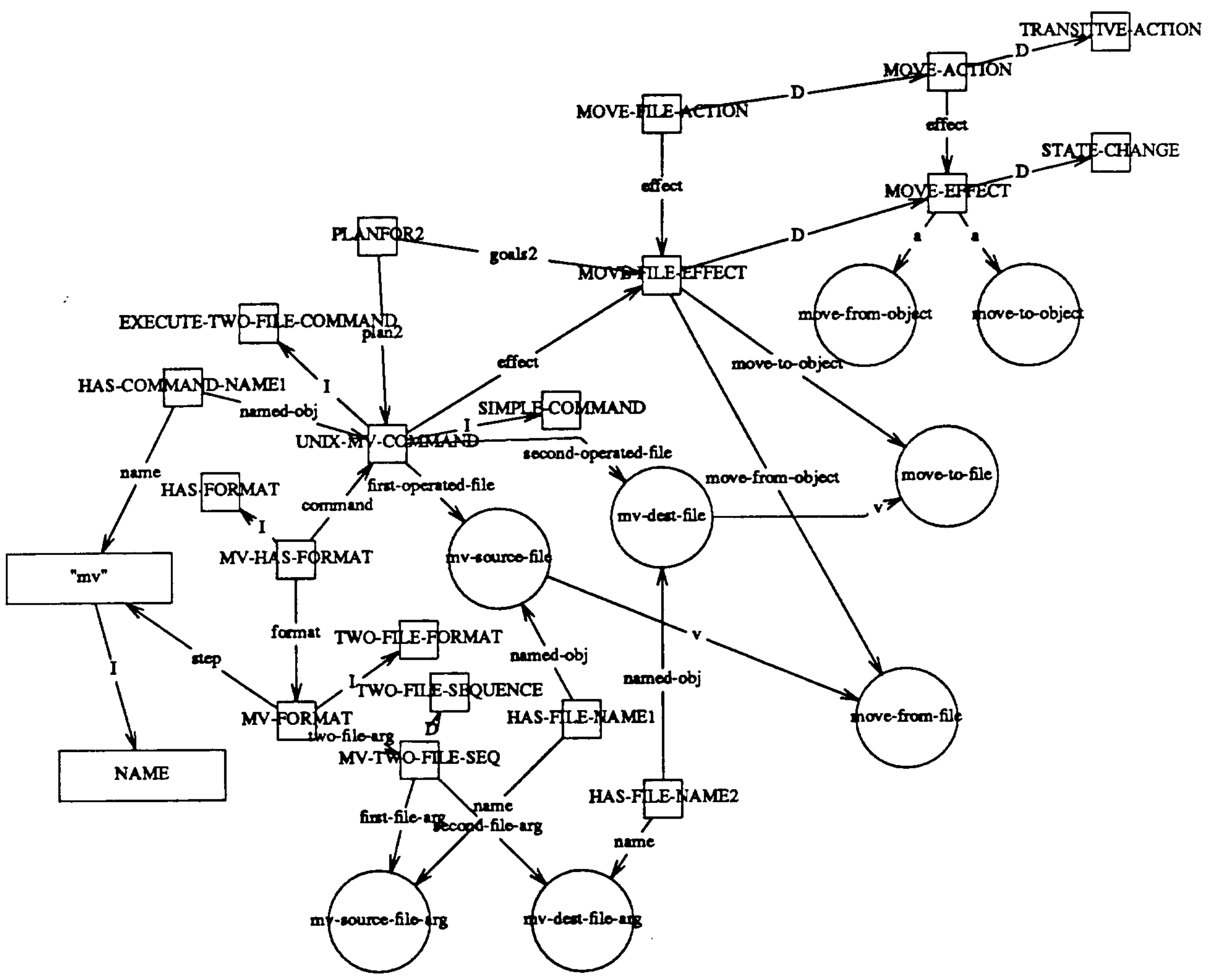
This diagram describes the rm command which is the plan for deleting a file.



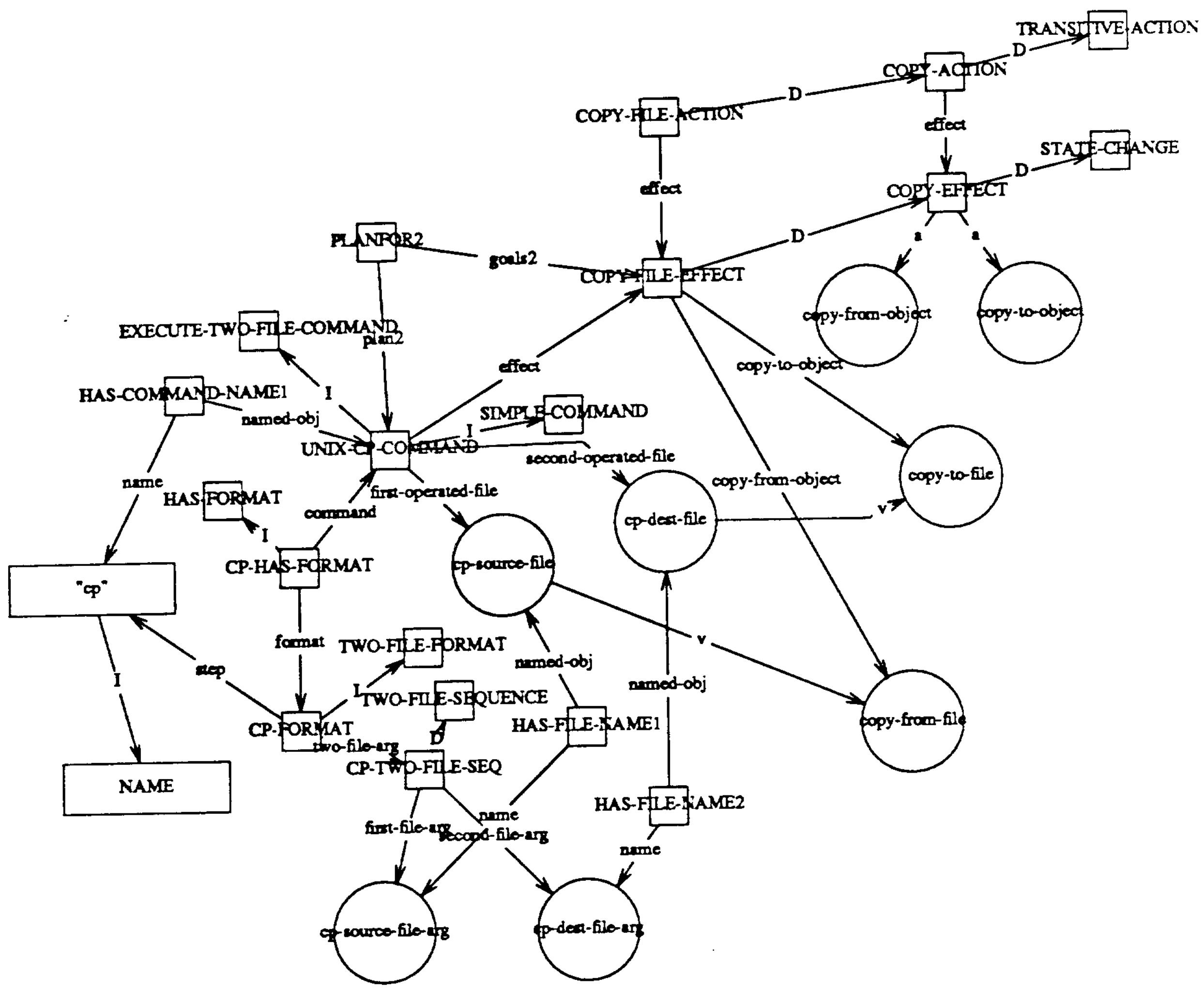
This diagram describes the rmdir command which is the plan for deleting a directory.



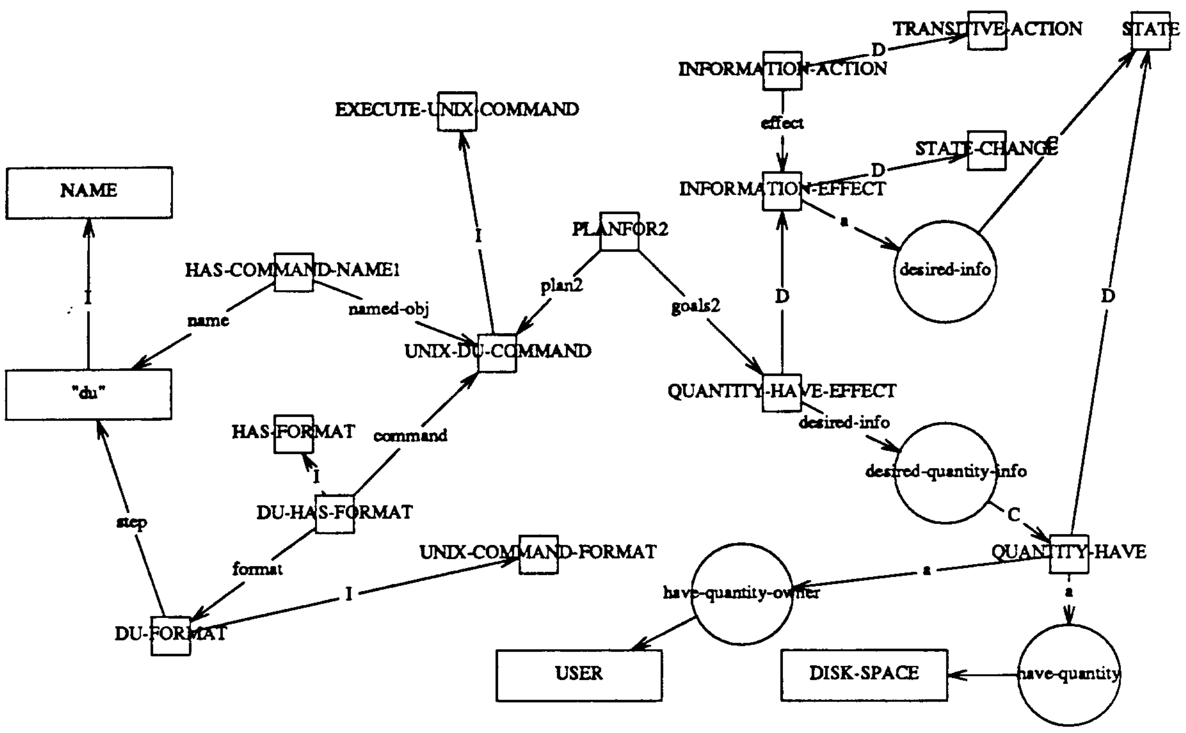
This diagram describes the mkdir command which is the plan for creating a directory.



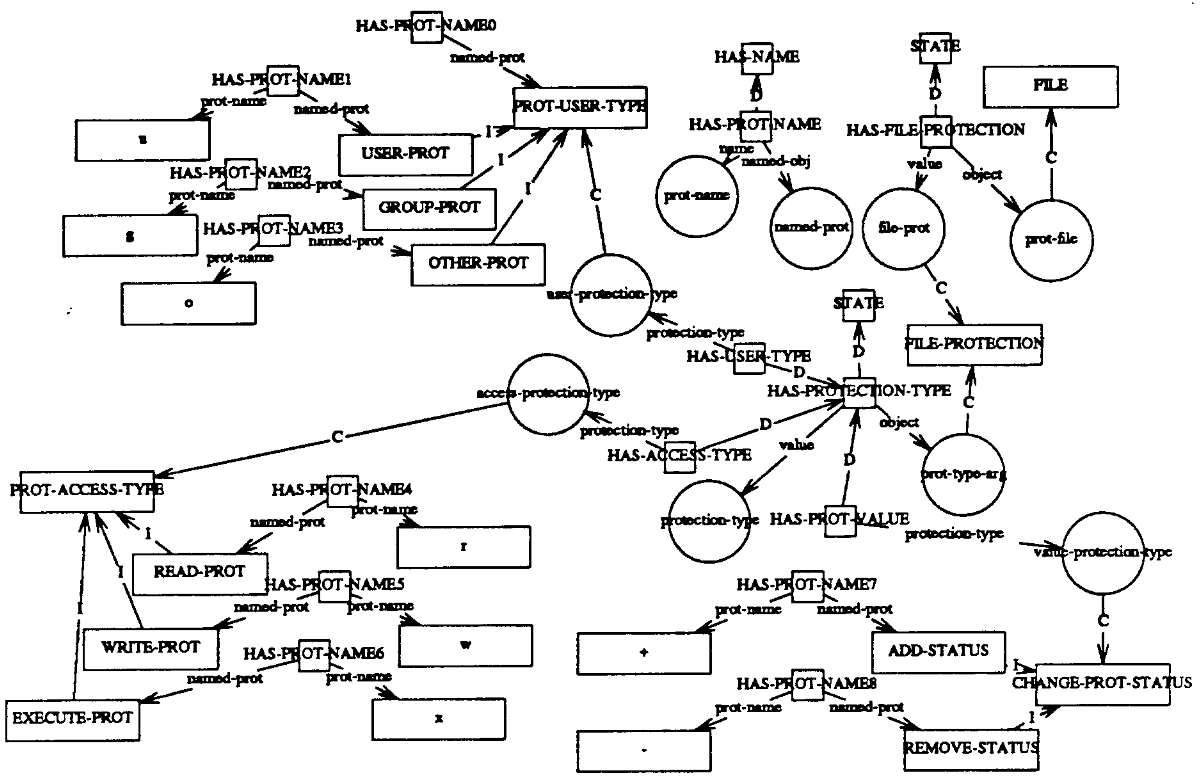
This diagram describes the mv command which is the plan for moving a file.



This diagram describes the cp command which is the plan for copying a file.

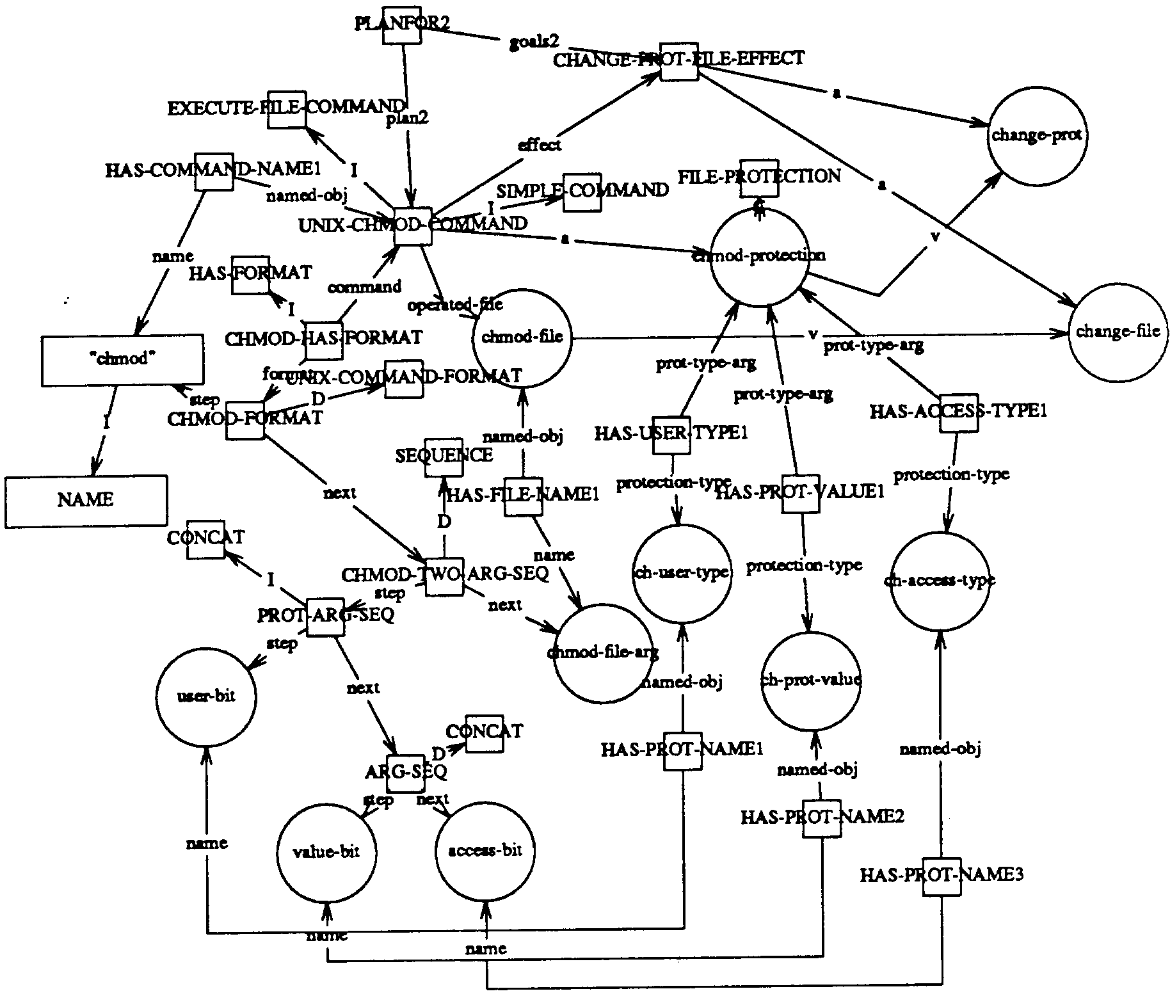


This diagram describes the du command which is the plan for finding out the quantity of disk space that a user has.

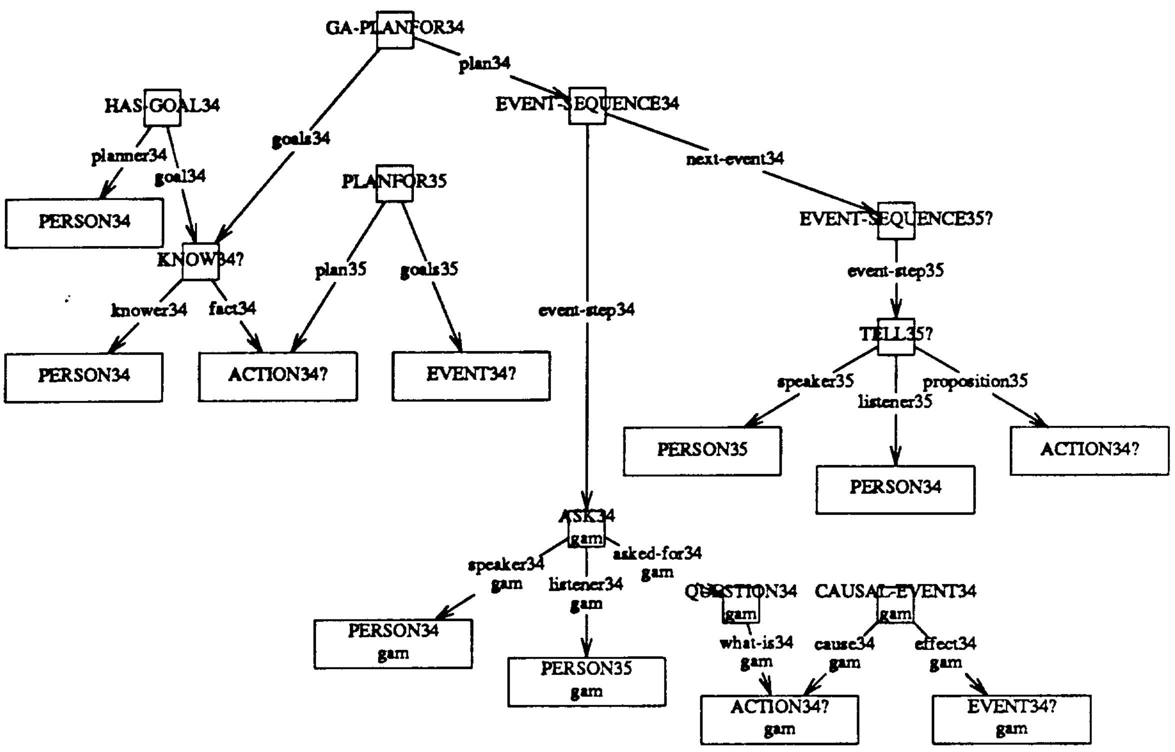


This describes UNIX file protection modes.

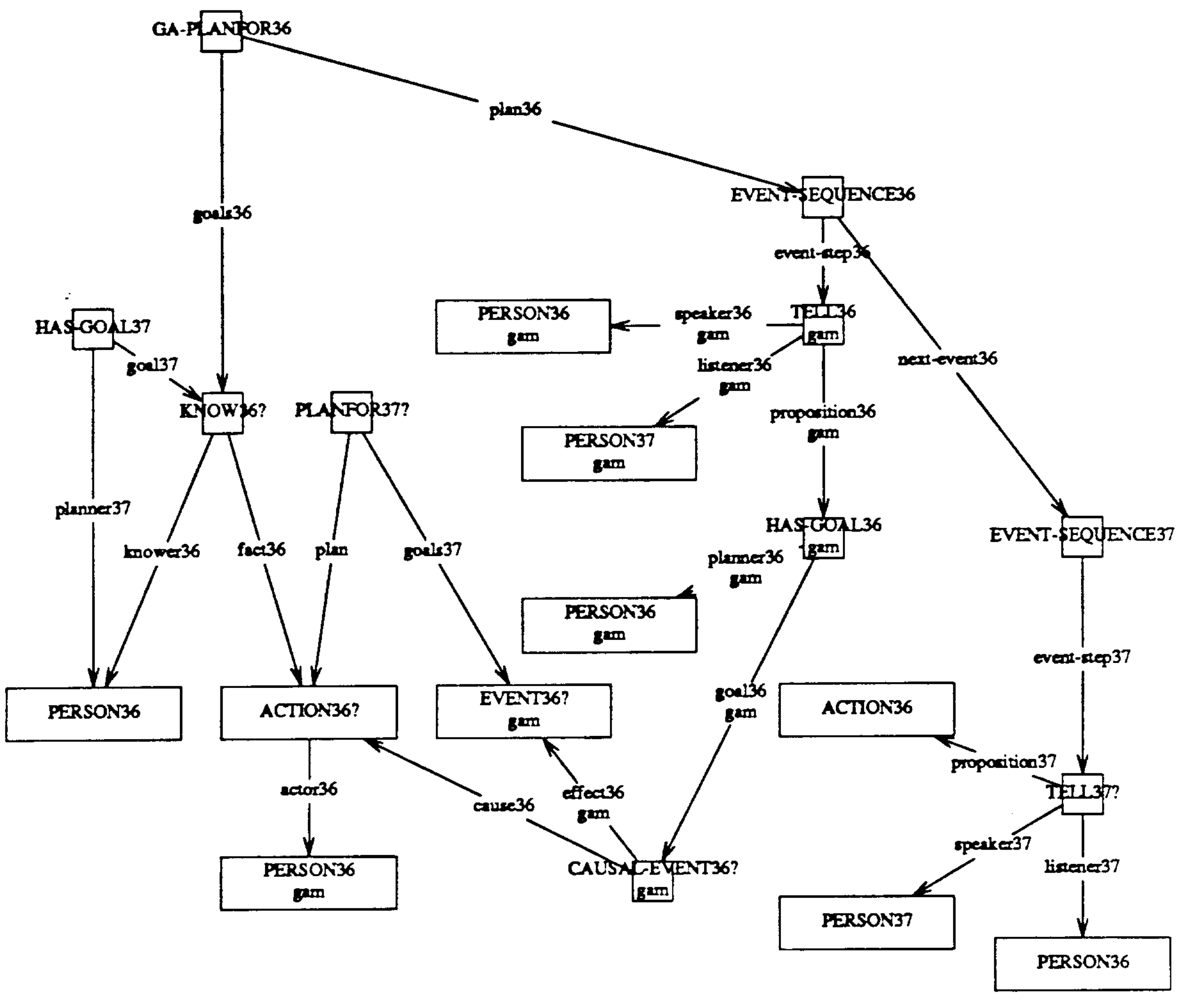
HAS-FILE-PROTECTION is a relation between FILEs and FILE-PROTECTIONs. A FILE-PROTECTION is described by three kinds of HAS-PROTECTION-TYPE relations, HAS-USER-TYPE, HAS-ACCESS-TYPE, and HAS-PROT-VALUE.



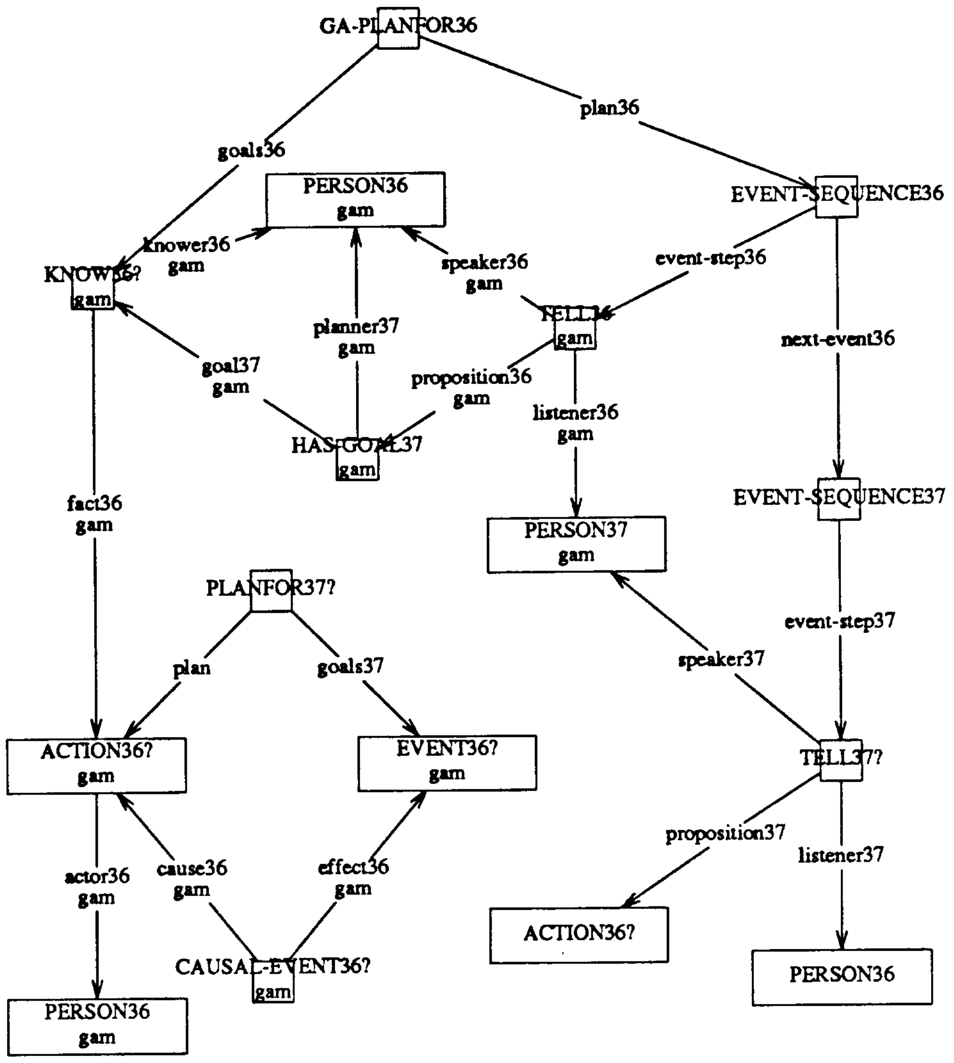
This diagram describes the chmod command which is the plan for changing the protection of a file.



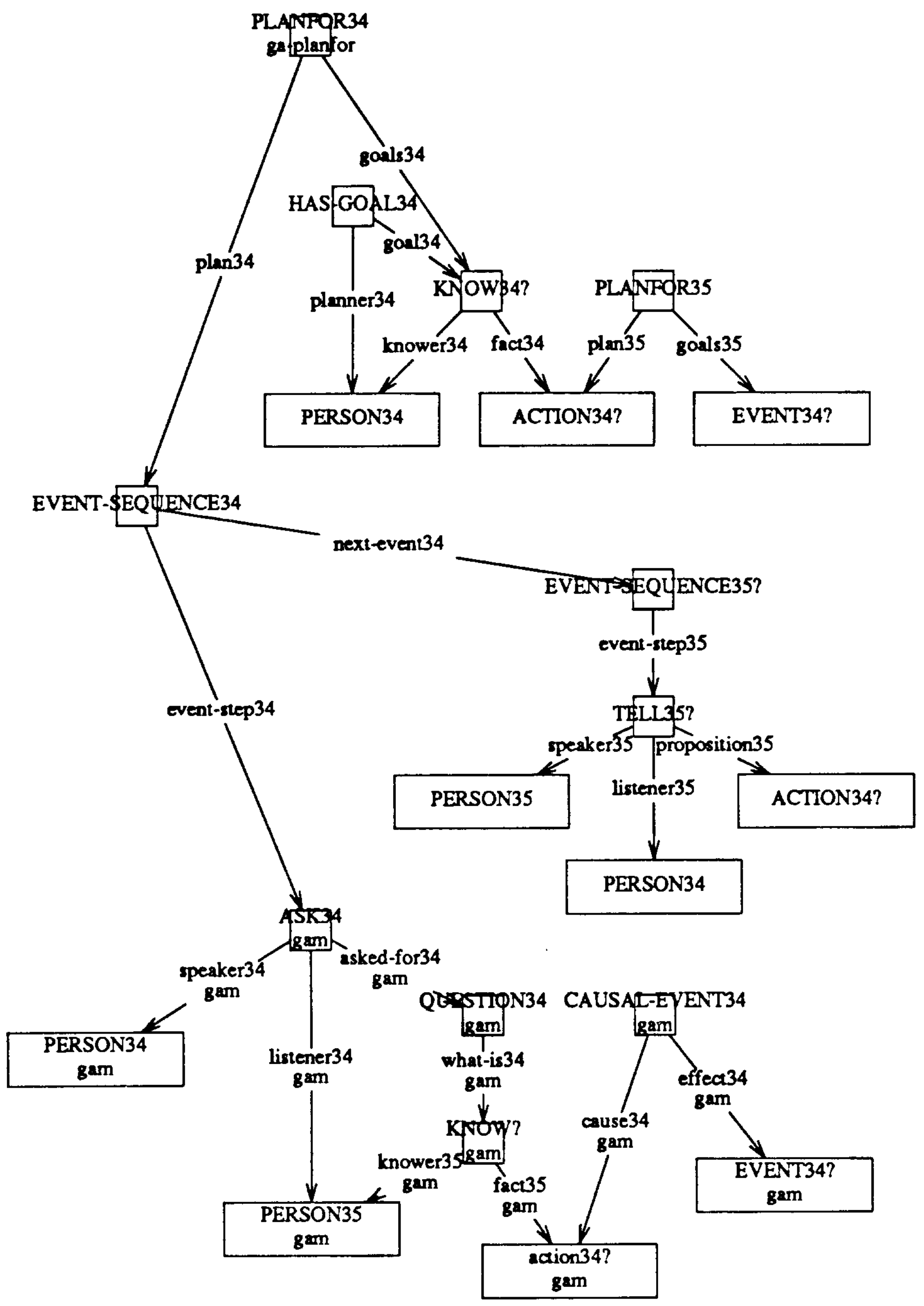
A plan for the goal of knowing how to perform some action is to ask how to do it, then be told how to do it. This planfor is used by **PAGAN**.



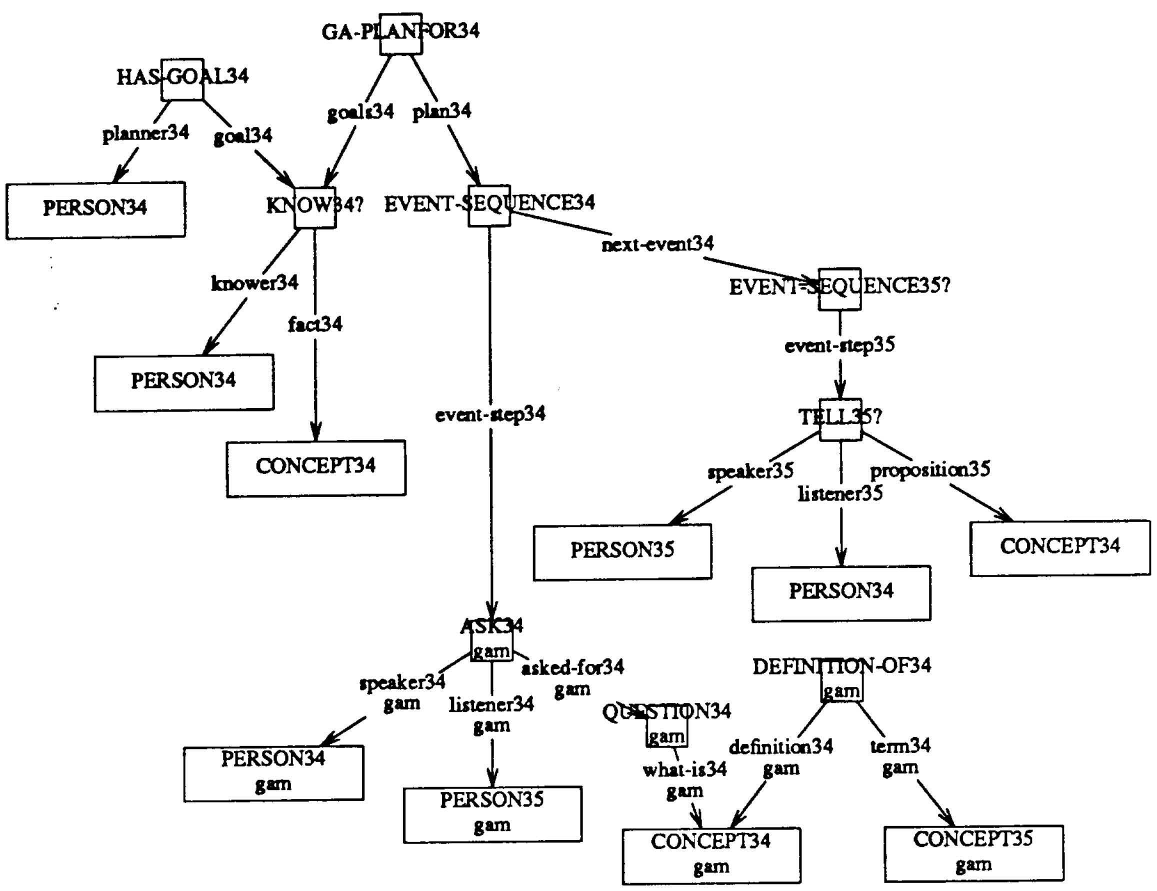
A plan for the goal of knowing how to perform some action is to indicate to the hearer that you want to do it, then be told how to do it. This planfor is used by PAGAN.



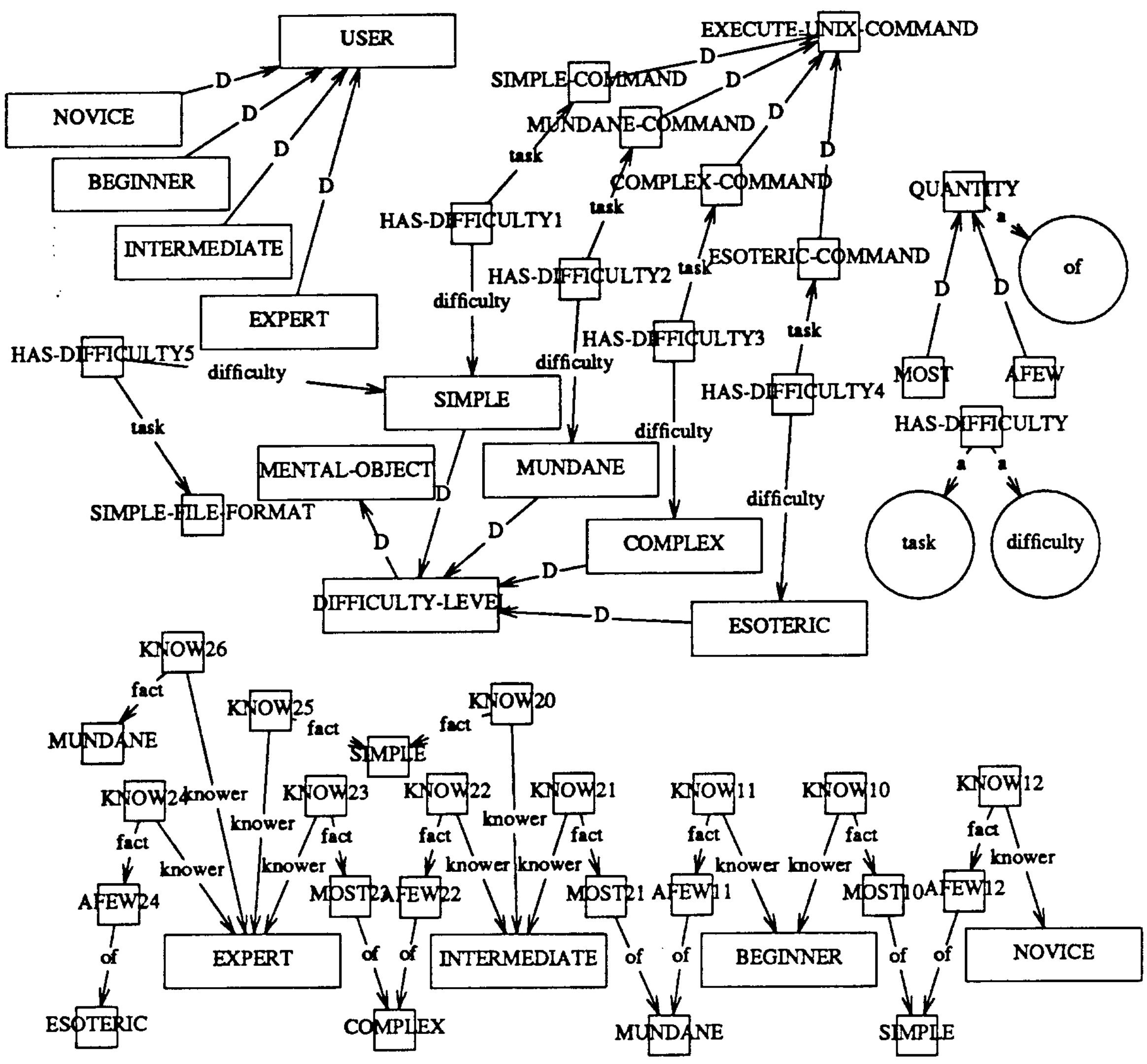
A plan for the goal of knowing how to perform some action is to indicate to the hearer that you want to know how to do it, then be told how to do it. This planfor is used by **PAGAN**.



A plan for the goal of knowing how to perform some action is to ask if the hearer knows how to do it, then be told how to do it. This planfor is used by PAGAN.



A plan for the goal of knowing a definition of some concept is to ask what it is, then be told what it is. This planfor is used by PAGAN.



This shows the main definitions for UC's User Model.

13. References

Allen, J. F., Frisch, A. M. and Litman, D. J. 1982. ARGOT: the Rochester dialogue system. In *Proceedings of the National Conference on Artificial Intelligence.* Pittsburgh, PA.

Allen, J. F. and Perrault, C. R. 1980. Analyzing intention in utterances. *Artificial Intelligence* Volume 15, pp. 143-178.

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

Brachman, R. and Schmolze, J. 1985 An overview of the KL-ONE knowledge representation system. In *Cognitive Science*, Volume 9, pp. 171-216.

Brown, J. S. and R. R. Burton 1976. A tutoring and student modelling paradigm for gaming environments. In the *Symposium on Computer Science and Education,* pp 236-246. Anaheim, CA.

Carberry, S. 1983. Tracking user goals in an information-seeking environment. In *Proceedings of the National Conference on Artificial Intelligence.* Washington, DC.

Chin, D. N. 1986. User modeling in UC, the UNIX consultant. In *Proceedings of the CHI-86 Conference,* Boston, MA, April.

Cox, C. A. 1986 *ALANA: Augmentable LANguage Analyzer* Computer Science Division, University of California, Berkeley, Report No. UCB/CSD 86/283.

Davis, R. 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 1976

Deering, M., Faletti, J., and Wilensky, R. 1982 *Using the PEARL AI Package* Computer Science Division, University of California, Berkeley, Memorandum No. UCB/ERL M82/19

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

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence,* Vol. 2, No. 3-4, pp. 189-208.

Grice, H. P. 1975. Logic and conversation. In *Studies in Syntax,* Vol. III, pp. 41-58, edited by P. Cole and J. L. Morgan. Seminar Press.

Grosz, B. and Sidner, C. L. 1985. *The Structures of Discourse Structure.* Center for the Study of Language and Information Report No. CSLI-85-39

Haas, N. and Hendrix, G. 1980. An approach to acquiring and applying knowledge, *Proceedings of the National Conference on Artificial Intelligence,* pp. 235-239, Stanford, Ca.

Jacobs, P. S. 1984. *PHRED: A Generator for Natural Language Interfaces* Computer Science Division, University of California, Berkeley, Report No. UCB/CSD 84/189.

Jacobs, P. S. 1985. *A Knowledge-Based Approach to Language Production.* Ph.D. thesis, University of California, Berkeley.

Kaplan, S. J. 1983. Cooperative Responses from a Portable Natural Language Database Query System. In *Computational Models of Discourse,* edited by Brady and Berwick. MIT Press, Cambridge, MA.

Litman, D. J. and Allen, J. F. 1984. A plan recognition model for clarification subdialogues. In *Proceedings of the Tenth International Conference on Computational Linguistics,* Palo Alto.

Luria, Marc. 1982. Dividing up the question answering process. In the *Proceedings of National Conference on Artificial Intelligence,* pp. 71-74, Pittsburgh, Pennsylvania.

Luria, M. 1985. Commonsense planning in a consultant system, *Proceedings, 1985 IEEE International Conference on Systems, Man, and Cybernetics.* pp. 602-606, Tucson, Arizona.

Martin, J., 1985. Knowledge acquisition through natural language dialogue, *Proceedings of the 2nd Conference on Artificial Intelligence Applications,* Miami, Florida.

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,* pp. 197-201. Boston, MA.

Morik, K. and C-R Rollinger. 1985. The real estate agent -- modeling the user by uncertain reasoning. In *AI Magazine,* Vol. 6, No. 2, pp. 44-52.

Newell, A., and Simon, H. A., 1972. *Human Problem Solving*. Prentice-Hall, Englewood Cliffs, N. J.

Norvig, P. 1983. Frame Activated Inferences in a Story Understanding Program. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence,* pp. 624-626.

Rich, Elaine. 1979. User modeling via stereotypes. In *Cognitive Science,* Vol. 3, pp. 329-354.

Rissland, E. L. 1983. Examples in Legal Reasoning: Legal Hypotheticals. In *Proceedings of the Eight International Joint Conference on Artificial Intelligence,* Vol 1., pp. 90-93.

Rissland, E. L., E. M. Valcarce, and K. D. Ashley. 1984. Explaining and Arguing with Examples. In *Proceedings of the National Conference on Artificial Intelligence,* pp. 288-294.

Rosch, Eleanor. 1978. Principles of categorization. In *Cognition and Categorization,* edited by E. Rosch and B. B. Lloyd. Lawrence Erlbaum. Hillsdale, N. J.

Sacerdoti, E., 1974. Planning in a hierarchy of abstraction spaces, *Artificial Intelligence* Vol. 5, pp. 115-135.

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.

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

Sidner, C. L. 1985. Plan parsing for intended response recognition in discourse. *Computational Intelligence 1,* pp. 1-10.

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

van Melle, W., 1980 *A Domain Independent System That Aids in Constructing Knowledge-Based Consultation Programs.* Heuristic Programming Project Report No. HPP-80-22, Computer Science Department, Stanford University, CA.

Webber, B. L. and E. Mays. 1983. Varieties of User Misconceptions: Detection and Correction. In the *Proceedings of the Eighth International Joint Conference on Artificial Intelligence,* Vol. 2, pp. 650-652. Karlsruhe, West Germany.

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

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

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

Wilensky, R., Arens, Y., and Chin, D. 1984. Talking to Unix in English: an overview of UC. *Communications of the Association for Computing Machinery,* June.