Talking to UNIX in English:

An Overview of an On-line UNIX Consultant*

Robert Wilensky

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

1. Introduction

UC (UNIX Consultant) is an intelligent natural language interface that allows naive users to communicate with the UNIX† operating system in ordinary English. The goal of UC is to provide a natural language help facility that allows new users to learn operating systems conventions in a relatively painless way. UC is not meant to be a substitute for a good operating system command interpreter, but rather, an additional tool at the disposal of the new user, to be used in conjunction with other operating system components.

UC allows the user to engage in natural language dialogues with the operating system. While there are a number of other natural language interfaces available today, these are mostly used as natural language front ends to particular data bases (for example, see Hayes & Carbonell 1981, Hendrix 1977, Robinson 1982, Waltz et al. 1978, and Woods 1970). In contrast, the user uses UC in order to learn how better to use the UNIX environment in which UC is embedded. UC can handle requests stated in a wide variety of forms, and has a number of features to enhance its function as a user interface. These include the following:

1) A robust language analyzer, which almost never has a "hard" failure and which has the ability to handle most elliptical constructions in context.

2) A context and memory mechanism that determines the focus of attention and helps with lexical and syntactic disambiguation, and with some aspects of pronominal reference.

3) Highly extensible knowledge bases both of facts about UNIX and about the English language.

*This research was sponsored in part by the Office of Naval Research under contract N00014-80-C-0732 and the National Science Foundation under grant MCS79-06543.
†UNIX is trademark of Bell Laboratories

4) A mechanism for trying to make sense out of ill-formed inputs.

5) An experimental planning component that provides commonsense reasoning in creating plans for the user.

6) A goal analysis mechanism, which performs functions related to interpreting the user's statements as speech acts.

7) A UC Teacher, which enables UC to learn new vocabulary and new facts about UNIX by being instructed in natural language.

While some of the components of the system are experimental in nature, the basic features of UC provide a usable device to obtain information about UNIX. In addition, it is straightforward to extend UC's knowledge base to cover areas of UNIX with which UC is not currently familiar.

2. UC Examples

The following examples are illustrative of the breadth of dialogues in which UC can participate. UC is capable of generating the answers shown below in response to most forms of the request users might supply. UC can participate in some dialogues that are considerably more involved that this one, and which will be discussed in more detail later on:

Simple Dialogue

User:How do i delete a file?
(or remove a file)
UC: Typing 'rm filename' will remove the file with name filename from your current directory.

User:a directory?
UC: Typing 'rm -r directoryname' will recursively delete the directory with name directoryname and all the files and directories contained in the directory.

User:How should one edit a file?
(or change a file)
UC: Typing 'ex filename' will get you into the ex editor; typing 'emacs filename' will get you into the emacs editor.

User:What is the best way to create a file? (or the right way or the proper way, or make a file)
UC: The best way to create a new file is to use the editor, i.e.- 'ex filename'.

User:a directory?

UC: Typing 'mkdir directoryname' will create a directory with name directoryname in your current directory.

User:Do you know how to read a file?
(or list a file)

UC: Typing 'pr filename' will print the file with name filename on your terminal.

User:How can I find out how full the disk is?

UC: Typing 'df' will tell you how full the disk is.

User:How can I find out who is on the system? (or on Unix or using Unix)

UC: Typing 'who' will tell you who is on Unix.

User:How can I compare two files?

UC: To compare two files, type 'diff file1 file2'.

User:How can I get more help?

UC: To get more information on Unix, try typing 'help'.

User:How can I sort a file?

UC: Type 'sort file1' to sort the file with name file1.

User:How can I find out who has access to my files?

UC: Typing 'ls -l filename' will tell what the protection on the file named filename is. (For more information on the ls command type 'man ls'.)

User:How can I change the write permission on my terminal? (or permission of or protection)

UC: To change the write permission on your terminal, type:
'mesg y' to allow writing to your terminal,
'mesg n' to disallow writing to your terminal.

User:What is a search path?

UC: A search path in Unix is a list of directories in which the operating system searches for programs to execute.

User:What do you know about Unix? (or What can you tell me about Unix?)

UC: I currently know about the Unix file system.

3. Rationale

The problem of naive user communication with an operating system provides an appealing domain for study and application of a number of Artificial Intelligence techniques. The domain is complex enough to provide substantial sub-problems, but not so unbounded that a useful working system must possess a hopelessly

large repetoire of knowledge. The task involves a quite realistic use of natural language, namely, participating in a purposive dialogue.

In addition, UC provides an application of natural language processing that people would actually be willing to use. Some common uses for natural language processing, for example, natural language front ends to data bases, may not have this property. Simple requests to a data base management system may be shorter in natural language than in typical data base accessing formalisms (see Hayes & Carbonell 1981 for some examples). However, this is less clearly true for more complex requests. Thus it may be the case that once a user has learned a somewhat more cryptic query language, he is apt to prefer it over natural language as it is likely to provide a more precise and less verbose mode of expression for many of his queries. Also, use of a natural language front end is probably not conducive to the learning of the underlying formalism.

In contrast, a naive user of a system would probably be willing to converse in English with a machine, since the alternatives are apt to be worse. Manuals and on-line help facilities are only marginally useful. A user usually needs to be fairly sophisticated in order to issue the right help command, and then these are successful only a fraction of the time. In the times that they do succeed, the information returned is apt to be cryptic and confusing.

Consider, for example, the problem of finding the answer to the question "How can I find out if two files are the same?" Since the user does not know the name of the appropriate command, retrieving by command name is not applicable (This is particularly true in UNIX, where many command names would appear to be unmotivated). Retrieval by a keyword is possible using 'file', but this is likely to return too much information, since operating systems generally have many commands related to files. In the current release of UNIX, for example, issuing an "apropros file" command returns five screenfuls of command names.

Also, there is no guarantee that the keyword with which the user states his request is appropriate for the particular operating system, or happens to index the appropriate information. For our previous example, in UNIX, retrieving by the keyword 'same' will find no associated commands. Nor will retrieval by the word 'difference'. However, using the keyword 'different' or 'compare' returns the lists "diff diff3" and "cmp", respectively.

In such situations, navigating through a maze of information is undesirable, and the user would probably prefer simply to pose the question to a colleague. However, people knowledgable enough to be helpful are not always available and usually do not have the time to offer. Under such circumstances, typing in the question in English, exactly as it occurred to the user, becomes an attractive alternative.

In addition, the domain is "soft" in the sense that a system that did not work all the time is still likely to be useful. For example, if UC failed to answer a question, the user is no worse off than he would be otherwise, and still has the normal repertoire of options at his disposal. The same is probably not the case if natural language is intended to be used as the primary means of communication. For example, a natural language interface to a data base that failed to successfully treat a user query would leave the user without recourse. However, UC need only succeed a high enough percentage of time for the user to benefit from using it. Failure to be helpful in any given query would be no more disasterous

for the user than would failure of the normal help facilities. Both would still retain some potential value for the user. Thus UC is an AI system that is useful even as it expands cover to a larger portion of its domain.

Lastly, the problem integrates a number of areas of concern for artificial intelligence, including natural language understanding, natural language production, planning and problem solving.

4. Problems with Natural Language Interfaces

A natural language interface must deliver a variety of services to the user. These involve a number of processes that are primarily linguistic in nature. The system must know enough about the primary linguistic conventions of the particular natural language in order to interpret the meaning of a user's request and to produce a reasonable natural language utterance of its own. A number of well known problems are associated with these tasks and have received a great deal of attention in the literature. For example, a natural language understander must be capable of resolving various kinds of ambiguities, determining the referents of pronouns and other noun phrases, interpreting an abbreviated utterance in the context of the previous dialogue, and making some sense out of ill-formed inputs. In addition, the interface will have to be continually extended, both to include new vocabulary and new facts about the domain of discourse. The constraints on the design of the interface imposed by the latter feature have been less well studied. Below we describe how these problems are handled in UC.

A useful natural language interface must also incorporate some processes that may be collectively referred to as commonsense reasoning. To demonstrate the importance of such extra-linguistic mechanisms, consider the following hypothetical dialogue with a naive interface (NI). We assume here that NI possesses knowledge about a language's syntactic and semantic conventions, but is not otherwise an intelligent system:

> User: I'm trying to get some more disk space.
> NI: Type 'rm *'

NI's suggestion, if executed by the user, would destroy all the user's files. This rather disturbing response might be generated by a naive interface because the response fulfills the user's literal request for more disk space. However, the answer lacks a certain cooperative spirit. A more felicitous answer might be "Delete all the files you don't need" or "Ask the systems manager for some more storage." However, in order to prefer these responses over the above, the interface must be able to infer that the user possesses some goals other than the one stated in the request, and that these background goals interact with the request to constrain the beneficial courses of action. Specifically, the interface would have to have realized that users generally like to preserve files they have already created, and that therefore a conflict exists between this goal and that of obtaining more space by deleting files. Then a method of reasoning about this situation would have to be employed.

These considerations fit more properly in the domain of planning and problem solving than language processing. However, an interface that was not designed

with them in mind would run the risk of innocently alienating its clientele. In the discussion that follows, we shall discuss the application of an experimental planning mechanism in UC to produce reasonable courses of action.

Actually, even the response shown in the previous example would most likely not have been generated by an interface as naive as the one we have been supposing. One further problem is that the user never explicitly made a request to the system. Rather, he simply stated something that he was trying to do, trusting that the interface would take it upon itself to offer some assistance. This is an instance of an *indirect speech act*, another aspect of natural language interaction that has received much attention (see for example Perrault, Allen, Cohen 1978). However, the class of problems that includes this one also includes some problems that have received less treatment. For example, consider answering the following question:

User: How can I prevent someone from reading by files?

The direct response to this question would be to use a protection command. However, an additional response in UNIX would be to use an encryption command, that is, a command which encodes the file so that one needs a key in order to decode it. The problem with offering the latter suggestion is that it does not literally fulfill the user's request. Encryption does not prevent someone from reading a file, but merely from understanding it when it is read. In order to decide to inform the user about encryption, then, the interface must assume that the user misstated his request. Presumably, the user is really interested in preventing others from learning the contents of his files. Having made this reinterpretation of the actual utterance, both methods of protection would be applicable.

In the next section we discuss how these and other problems are addressed by the design of UC.

5. The Structure of UC

UC runs on a VAX11/780 and 750 and is programmed in FRANZ LISP and in PEARL, an AI language developed at Berkeley (Deering et al., 1981). Although UC is itself a new system, it is built out of a number of components, some of which have been under development for some time. These components are outlined briefly in this section.

5.1. Linguistic Knowledge

The primary natural language processing in UC is done by an extended version of PHRAN (PHRasal ANalyzer) and PHRED (PHRasal English Diction) (Wilensky 1981b). PHRAN reads sentences in English and produces representations that denote their meanings; PHRED takes representations for ideas to be expressed, and produces natural language utterances that convey these ideas. These programs represent the very front and back ends of the interface, respectively. In addition, PHRAN and PHRED are relatively easily extended to process new linguistic forms and domains.

For UC, PHRAN has been extended to handle some forms of ellipsis. A component to process ill-formed constructions is under developement but has not yet been incorporated into UC. A more significant extention to PHRAN is the addition of a context mechanism. This is essentially an activation-based memory used by UC to keep track of the focus of the discourse (cf. Grosz 1977), handle some aspects of reference, and help in word disambiguation.

5.2. Goal Analysis

Once the meaning of an utterance has been extracted by PHRAN, it is passed to the Goal Analyzer. This module performs a form of plan recognition on the input, so that indirect requests may be interpreted as such. The Goal Analyzer also attempts to do the sort of "request correction" demonstrated in the example given previously. That is, it not only tries to interpret the goal underlying the action, but it may infer that the user wanted something somewhat different from what was actually requested.

5.3. Plan Formation

Responses are generated in UC by using a plan generation mechanism called PANDORA (Faletti 1982). PANDORA takes as input the goal produced by the Goal Analyzer, and tries to create a plan for the user that will achieve this goal. In doing so, PANDORA reasons in a manner that will produce commonsense results, thus preventing the sort of anomalies discussed previously. In the majority of cases, PANDORA will do nothing more complex that look up a stored plan associated with the goal in memory and return this as the answer. This appears to be adequate for most simple requests. Thus even though PANDORA is an experimental system, we can run UC in a useful mode without encountering most of the complexities this component was designed to contend with.

5.4. Expression Formation

Once a plan is selected by PANDORA, it is sent to the expression formation component to decide which aspects of the response should actually be mentioned. Currently, this component is quite sketchy, largely because most of our effort has gone into request understanding rather than answer generation. Ultimately, the goal is to produce answers that do not contain facts the user is likely to know, etc. Some of the principles involved in this mechanism are described in Luria (1982).

Once the answer is formulated, it is sent to PHRED, which attempts to express it in English. In addition, some of the facts stored in the UC knowledge base are associated with canned pieces of text. If the answer corresponds to such a simple fact, this text is output rather than using the PHRED generator. Since the most frequently asked questions are generally of this form, considerable time savings is accomplished by bypassing the more complex generation process.

5.5. Representation and Knowledge Base

Facts about UNIX, as well as other pieces of world knowledge, are represented declaratively in an associative knowledge base managed by PEARL (Package for Efficient Access to Representations in LISP). This is essentially a data base management package that supports labelled predicate-argument propositions and allows the implementation of frame-like structures. Having a data base management package is particularly important for our goal of scaling up, as it allows us to add and use new facts without creating new code.

PEARL incorporates such standard features as automatic inheritance and various demon facilities. In addition, PEARL has a flexible indexing structure which allows the user to advise it about how to store facts to take advantage of how they are likely to be used.

The theory of knowledge representation used in UC is beyond the scope of this report. We will give enough examples of the representation of individual facts to suggest to the reader the basic elements of our scheme. However, we have liberally simplified the actual representations used in UC for expositional purposes. Further details of PEARL are available in Deering, Faletti, and Wilensky (1981, 1982).

As an example, consider the following simplified version of the PEARL representation of the fact that using the 'rm' command is a way to delete a file:

```
(planfor
  (concept
    (causation
      (antecedent (do (actor ?X)))
      (consequent
              (state-change
                      (actor ?F/is-file)
                      (state-name exist)
                      (from T)
                      (to Nil)))))
  (is (use (actor ?X) (command (name rm) (arg ?F)))))
```

Briefly, *planfor* is a predicate used to denote that fact that a particular action can be used to achieve a particular effect. This predicate takes two arguments, labeled *concept* and *is* (Labeled arguments have the same syntax as predicates. They can be distinguished by their position in the formula). The semantics of this predication is that the *is* argument is a plan for achieving the *concept* argument. The arguments are PEARL structures of the same sort. For example, the *concept* argument is a *causation* predicate, denoting that an action causes a state change. This particular state change describes a file going from an existent to non-existent status. Items prefixed with question marks are variables. These can be constrained to match only certain kinds of items by following them with a (one argument) predicate name. Thus *?F/is-file* constrains ?F to match something that is a file (This form is an abbreviation for something which can be only be represented less conveniently in the regular PEARL notation).

A question such as "How can I delete a file?" is analyzed by PHRAN into a form similar to this, but with the *is* argument filled with a symbol indicating an unspecified value. UC attempts to answer this question by using this form to fetch from the PEARL knowledge base. Such a fetch will retrieve the above representation. The contents of the *is* argument can then be used for generation.

6. PHRAN

PHRAN was originally implemented by Yigal Arens, and applied to the UC domain by David Chin. In addition to analyzing sentences for UC, PHRAN serves as the natural language front end for several story understanding systems under development at Berkeley, and has been tested on a variety of other sentence sources as well. PHRAN is discussed in Wilensky and Arens (1980) and will only be described briefly here.

One of the primary design goals of PHRAN is that it be easily extended to cope with new language forms. Applying PHRAN to the domain of requests about UNIX was therefore as much a test of this design as it is a useful application, as most of the forms and vocabulary used by UC were new to PHRAN. As of this writing, the PHRAN component of UC understands requests covering about 35 different topics, each of which may be asked in many different linguistic realizations. To extend PHRAN to handle a new vocabulary item now requires only a few minutes of effort by someone familiar with PHRAN patterns, provided of course that the representation to be produced is understood. As the section on UC Teacher suggests, part of this process has already been automated to eliminate the need for trained personnel.

At the center of PHRAN is a knowledge base of *patter-concept pairs*. A *phrasal patterns*. is a description of an utterance that may be at many different levels of abstraction. For example, it may be a literal string such as "so's your old man"; it may be a pattern with some flexibility such as "<nationality> restaurant", or "<person> <kick> the bucket"; or it may be a very general phrase such as "<person> <give> <person> <object>".

Associated with each phrasal pattern is a *conceptual template*. A conceptual template is a piece of meaning representation with possible references to pieces of the associated phrasal pattern. Each phrasal pattern-conceptual template association encodes one piece of knowledge about the semantics of the language. For example, associated with the phrasal pattern "<nationality> restaurant" is the conceptual template denoting a restaurant that serves <nationality> type food; associated with the phrasal pattern "<person1> <give> <person2> <object>" is the conceptual template that denotes a transfer of possession by <person1> of <object> to <person2> from <person1>. The understanding process reads the input text and tries to find the phrasal patterns that apply to it. As it reads more of the text it may eliminate some possible patterns and suggest new ones. At some point it may recognize the completion of one or more patterns in the text. It may then have to chose among possible conflicting patterns. Finally, the conceptual template associated with the desired pattern is used to generate the structure denoting the meaning of the utterance.

The version of PHRAN used in UC takes from 1.5 to 8 seconds to analyze a sentence, with a typical sentence taking about 3.5 seconds of CPU time. This version

of PHRAN contains about 675 basic patterns of varying length and abstraction. About 450 of these patterns are individual words. Of these, about 60 are verbs. PHRAN knows both the roots of these verbs, as well as all the morphological variations in which each verb may be found. Of the 220 patterns containing more than one word, about 90 indicate the way a particular verb is used. This latter group of patterns can be used by the program, when the need arises, to generate approximately 800 additional patterns.

6.1. Ellipsis

For UC, PHRAN has been extended by Lisa Rau to include an ellipsis mechanism. This mechanism handles both intra-sentential forms, such as "I want to delete the small file and the large", and inter-sentence forms, as in "How do I delete a file? A directory?"

Ellipsis is handled by first letting the basic PHRAN analysis mechanism produce what it can from the input. This process leaves a history of the patterns used to arrive at that understanding. If the result of this process is incomplete (e. g., generally something that is not a sentence where a sentence is expected), then previously used PHRAN patterns are examined to see if they match the patterns used to understand the fragment. If so, then the words of the fragment are substituted for the words of the previous sentence that correspond to the common pattern. The resulting linguistic unit is then re-analyzed by PHRAN.

7. The Context Mechanism

PHRAN's knowledge is more or less confined to the sentence level. Thus PHRAN by itself is unable to deal with reference, and cannot disambiguate unless the linguistic patterns used require a particular semantic interpretation of the words. In addition, the same utterance may be interpreted differently in different contexts, and the mechanism described so far has no facility for accomplishing this.

To fulfill the need for processing on the discourse level, we have constructed a single mechanism which addresses many of these problems, called the *Context Model*. The Context Model contains a record of knowledge relevant to the interpretation of the discourse, with associated levels of activation. The Model is manipulated by a set of rules that govern how elements introduced into the Context Model are to influence it and the system's behavior.

PHRAN and the Context Model interact continually. PHRAN passes its limited interpretation of the input to the Context Model, and it in turn determines the focus of the conversation and uses it to resolve the meaning of ambiguous terms, of references, etc., and passes these back to PHRAN.

The Context Model groups related entries and arrives at a notion of the situation being discussed. Alternative situations in which a concept may appear can be ignored, thus enabling the system to direct the spreading of activation. The Context Model is similar to Grosz's scheme for determining focus of a task oriented dialog and using it to resolve references (Grosz, 1980). However, Grosz's system relies heavily on the inherent temporal structuring of the task,

whereas we are trying to develop an approach that is independent of the type of subject matter discussed.

The following UC example illustrates the system's ability to shift focus freely according to the user's input, including the ability to store and recall previous contexts into focus:

[1] User: How do I print a file on the line printer?
[2] UC: To print a file on the line printer
type 'lpr filename'.

.
. (intervening commands and questions)
.

[3] User: Has the file foo been printed yet?
[4] UC: The file foo is in the line printer queue.
[5] User: How can I cancel it?
[6] UC: To remove files from the line printer queue,
type 'lprm username'.

In order to reply to the last question, UC must find the referent of 'it'. The language used implies that this must be a command, but the command in question was issued some time ago. Since then, intervening commands have been issued, so that the chronologically more recent command is not the proper referent. The system is able to determine the correct referent because the context of [1] and [2] had been stored previously, and recalled upon encountering [3].

7.1. Structure and Manipulation of the Context Model

The Context Model is in a constant state of flux. Entries representing the state of the conversation and the system's related knowledge and intentions are continually being added, deleted, or are having their activation levels modified. As a result the same utterance may be interpreted in a different manner at different times. Following are short descriptions of the different elements of the system.

7.1.1. Entries

The Context Model consists of a collection of entries with associated levels of activation. These entries represent the system's interpretation of the ongoing conversation and its knowledge of related information. The activation level is an indication of the prominence of the information in the current conversational context, so that when interested in an entry of a certain type the system will prefer a more highly activated one among all those that are appropriate. There are various types of entries, and these are grouped into three general categories:

1) Assertions - statements of facts known to the system.
2) Objects - objects or events which the system has encountered and that may be referred to in the future.

3) Intentions - entries representing information the system intends to transmit to the user (i.e. output) or other components of an understanding system (e.g. goal tracker, planner), and entries representing information the system intends to determine from its knowledge base.

7.1.2. Clusters

The entries in the Context Model are grouped into *clusters* representing situations, or associated pieces of knowledge. If any one member of a cluster is reenforced it will cause the rest of the members of the cluster to be reenforced too. In this manner inputs concerning a certain situation will continue reenforcing the same cluster of entries -- those corresponding to that particular situation. Thus the system arrives at a notion of the topic of the conversation which it uses to help it choose the appropriate interpretation of further inputs.

7.1.3. Reenforcement

When the parse of a new input is received from PHRAN the system inserts an appropriate entry into the Context Model. If there already exists an entry matching the one the system is adding then the activation levels of all entries in its cluster(s) are increased. The level of activation decays over time without reenforcement, and when it falls below a given threshold the item is removed.

7.1.4. Stored Clusters

Upon inserting a new item in the Context Model the system retrieves from a database of clusters all those that are indexed by the new item. Unification is done during retrieval and the entries in the additional clusters are also inserted into the Model, following the same procedure described here except that they are given a lesser activation. We thus both avoid loops and accommodate the intuition that the more intermediate steps are needed to associate one piece of knowledge with another the less the mention of one will remind the system of the other.

The system begins operation with a given indexed database of clusters, but clusters representing various stages of the conversation are continually added to it. In principle, this should be performed automatically when the system is cued by the conversation as to the shifting of topic, but currently the system user must instruct it do so. Upon receiving such an instruction, then, all but the least activated entries in the Context Model are stored as a cluster indexed by the most highly activated among them. This enables the system to recall a situation later when presented with a related input.

7.1.5. Operations on Entries in the Context Model

After a new entry is made in the Context Model the process described above takes place and eventually the activation levels stabilize, with some of the items being deleted, perhaps. Then the system looks over each of the remaining entries and, if it is activated highly enough, performs the operation appropriate

for its type. The allowed operations consist of the following:

1) Deleting an entry.
2) Adding another entry.
3) Transmitting a message to another component of the system (i.e. output to the user or data to another program, e.g. PANDORA (Faletti, 1982), for more processing)
4) As part of the UC system, getting information from the UNIX system directly (and inserting an entry corresponding to the result).

7.2. Example

In [1] the user asks a simple question. PHRAN analyzes the question and sends the Context Model a stream of entries to be inserted. Among them are the fact that the user asked for a plan for printinga file on the line printer. The system records these facts in the Context Model. Indexed under the entry representing the user's desire to obtain a goal there is a cluster containing entries representing the system's intent to find a plan for the goal the user has and instructing the system to tell the user of this plan. This cluster is instantiated here with the goal being the particular goal expressed in the question. The entry expressing the system's need for a plan for the user's goal leads to the plan in question being introduced also. This happens because the system happens to already have this association stored. When the system looks over the entries in the Context Model and comes to the one concerning the need to find the plan in question it will check to see if an entry for such a plan already exists, and in our case it does. But if no plan were found, the system would insert a new entry into the Context representing its intent to pass the information about the user's request to the planner, PANDORA (Faletti, 1982). PANDORA will in turn return the plan to be inserted in the Context Model.

When the system finds the plan (issuing the command above) and inserts a new entry instructing the system to output it to the user. This ultimately results in generating [2].

The topic shifts and the previous context is stored, indexed by the most highly activated entries, including the file name, the mention of the line printer, the event of printing the file, and the command issued.

When [3] is asked, this cluster is loaded again into the Context Model. To determine the referent of 'it' in [5], the Context Model is examined for highly activated entries. Since the command to print the file would have just have been brought back into the Model, it will be more highly activated that other, more recent request.

7.3. Shortcomings

The system is not currently able to determine on its own that the topic has changed and that it must store the current context. When it is instructed to, the current system stores essentially a copy of the more highly activated elements of the Context Model when creating a new cluster. They are not assumed to have any particular structure or relations among them other than all being highly activated at the same time. This causes two problems:

1) As a result it is very difficult to generalize over such clusters (cf. Lebowitz, 1980). The system may at some point determine a plan for changing the ownership of a particular file, and store a cluster containing it. If it is faced with the need to change the ownership of another file, however, the system will not be able to use this information.

2) There is no way to compare two clusters and determine that in fact they are similar. Thus we may have many clusters indexed by a certain entry all of which actually describe essentially the same situation.

8. Goal Analysis

Goal analysis is the process of inferring a user's plausible goal from an input statement or request. The UC Goal Analyzer, implemented by James Mayfield, works by attempting to apply to the input a set of rules designed explicitly for this purpose. Each rule consists of an input and an output conceptualization. Should a rule match an input, the associated output conceptualization is inferred. This process is iterated on the inferred conceptualization until no more rules are found to apply. The final product of this process is assumed to be the user's intention.

For example, consider the following indirect request and UC response:

User:I want to delete a file.
UC: Typing 'rm filename' will remove the file with name filename from your current directory.

This response is generated as follows. The Goal Analyzer has a rule that states that the assertion of a goal means that the user does not know how to achieve that goal. This rule is represented in this form:

```
(Goal-Analysis-Rule
  [In-Concept
    (goal (planner ?pers) (objective ?obj))]
  [Out-Concept
    (not
      (state
        (know
          (actor ?pers)
          (fact
            (causation (antecedent *unspec*) (consequent ?obj)))))))])
```

The application of this rule to the input produces a conceptualization denoting that the user does not know how to delete a file. Iterating the process, the Goal Analyzer finds a rule applicable to this inference, namely, that an assertion of not knowing how to do something means that the user wants to know how to do that thing. This is represented as:

```
(Goal-Analysis-Rule
  [In-Concept
    (not
      (state
        (know
          (actor ?pers)
          (fact
            (causation
              (antecedent *unspec*)
              (consequent ?obj)))))))]
  [Out-Concept
    (goal
      (planner ?pers)
      (objective
        (know
          (actor ?pers)
          (fact
            (causation
              (antecedent *unspec*)
              (consequent ?obj))))))])
```

Applied to the inference just produced, this rule instructs UC that user wants to know how to delete a file. In the next interation, the Goal Analyzer finds no rule applicable to its latest conclusion, and passes this conclusion along to UC as the user's intention. It is then straightforward for UC to produce the response shown above.

The Goal Analyzer has several other rules of this sort. One set of such rules tries to "correct" a user's misstatement of his goal. This is done by checking to see if a stated (or inferred) goal is an instance of a known normal goal. For example, UC contains the facts that not wanting others to learn the contents of one's files is normal, and that reading is a way of coming to know something. Thus a user's statement that he is trying to prevent someone from reading his files will be interpreted to mean that he is trying to prevent them from coming to know the content of his files. This enables UC to give a broader class of answers, as indicated in the beginning of this report.

The current UC Goal Analyzer has two potential drawbacks:

1) There are probably some inputs that require more elaborate plan-goal analysis. For example, the statement of a goal that is normally instrumental to some other goal may entail inferring that goal. This situations might require the sort of plan analysis that we have postulated in story understanding (Wilensky 1982). However, in practice, such cases seem not to arise in the UC task environment.

2) The Goal Analyzer is not sensitive to context nor does it incorporate a model of the user. For example, if a sophisticated user says that he wants to delete the file 'foo', UC should not interpret this as a request for information about how to delete a file in general. Rather, it is more likely that there is some problem with this particular file. This has not arisen as a problem in our application where we make some relatively simplistic assumptions about the user.

9. Extending UC to Process More Complex Requests

Most requests require more complex processing, however. For these situations, UC uses a reasoning component based on the PANDORA planning mechanism (Wilensky 1981a). PANDORA, implemented by Joe Faletti, is based on a model of planning in which goal detection and goal interactions play a prominent role. For example, consider the previous example of the indirect request: "I need some more disk space." A literal response to this remark might be "Type 'rm *'", which is most likely not what the user had in mind.

The problem with this response, of course, is that it violates an unstated user goal, namely, that the user wants to preserve what he has already put on the disk. An intelligent consult must be able to infer such goals, and reason about the interactions of such goals with those explicit in the user's request. In this example, an implicit goal (preserving a file) may conflict with the stated goal (getting some more space), and this possibility must be explored and dealt with.

Although it was originally constructed to be an autonomous planner, PANDORA's architecture is well suited for this sort of reasoning. PANDORA first tries to apply a stored plan to a given goal. It then simulates the situation that may result from the current state of the world using a mechanism called a *Projector*. In the above example, the simulation will reveal, among other things, that some files will get destroyed, as this is a consequence of the 'rm' command.

Another of PANDORA's basic components is called the *Goal Detector*. This mechanism determines the goals the planner should have in a given situation. The goal detector is essentially a collection of demons that respond to changes in the environment, including the simulated environment created by the projector. In this example, when the simulated future reveals the possible destruction of a file, the goal detector will react to this possibility by inferring the goal of preserving this file.

Since this preservation goal arises from a plan of the user's, PANDORA also infers that there may be a goal conflict between this goal and the goal underlying the user's original request. PANDORA makes this inference by considering a goal giving rise to a preservation goal as another situation in which to detect a goal (namely, the goal of resolving the goal conflict). Then a plan for this "resolve-goal-conflict" goal can be sought by successive application of the whole planning process.

This algorithm makes use of a *meta-planning* representation for planning strategies. The goal of resolving a goal conflict is actually a meta-goal, that is, a goal whose successful execution will result in a better plan for other goals. This formulation allows the goal detection mechanism to be used to solve the problem of goal conflict detection, and the normal planning process to find a resolution to such a problem. More detail on meta-planning and the associated algorithms is given in Wilensky 1981a).

In the example at hand, the presence of a goal conflict is only a possibility, as the user may well have some files that he doesn't need. A general strategy in such situations is to determine whether the possibility actually exists. This would lead to the generation of question "Do you have any files that you do not need?" If the user's response is negative, then the conflict does in fact exist, and a conflict resolution strategy must be employed.

A strategy that is applicable to all conflicts based on a shortage of resources is to try obtaining more of the scarce resource. In the example above, the scarce resource is disk space. PANDORA would then create a new goal of obtaining more disk space. Since a stored plan for this goal is to ask the systems manager for more room, UC can respond with the advise that the user request more room from the systems manager. An implementation of PANDORA and a fuller explanation of its role in UC is found in Faletti (1982).

Of course, it is possible to store requesting more room as a plan for the original goal, and by-pass all this complex reasoning. The problem with such a solution is that it would preclude the possibility of informing the user that he could delete a file that he doesn't need. Alternatively, we could make a more complex canned plan that checks for just this situation. That is, the stored plan would be to delete a file one doesn't want, and, if this fails, then to request more space. The problem here is that it would be necessary to include all possible interactions in the canned plan. While it may be desirable to include some of these some of the time, to deal with unanticipated interactions, a more general reasoning mechanism is required. For example, if the systems manager is unavailable and the user's desire for the disk space is great enough, deleting a file may still be a reasonable solution; however, it is unlikely that such a possibility would be anticipated by the canned plan.

10. UC Teacher

One of the primary design goals of the various components of our system is that it be a relatively straighforward task to extend them. Our basic approach is similar to that used in expert system and other areas of AI. This is to formulate knowledge in a declarative data base of some sort so that the straighforward addition of facts may be accomodated without reprogramming.

The use of highly declarative representations has enabled us to provide another sort of extensibility for UC. This is the power to extend the system by telling it in English addition facts about the language and about the UNIX domain. UC can be extended using natural language through the use of an accompanying UC Teacher component, developed by Jim Martin. For example, consider the following dialogue, in which a version of UC does not previously know about the words 'delete' and 'remove', or about how to accomplish these actions in UNIX:

User:To delete something means to cause it to not exist.
UCT:Entering "delete" into PHRAN patterns.

User:You can delete a file by typing rm filename.
UCT:Adding new cluster to UC memory.

User:To remove a file means to delete it.
UCT:Entering "remove" into PHRAN patterns.

User:How do I remove a file?

UC: Typing 'rm filename' will delete the file with name filename.

User:How do I delete a file?
UC: Typing 'rm filename' will delete the file with name filename.

UC Teacher uses definitional information, such as that contained in the first sentence, to create new PHRAN pattern-concept pairs. To do so, UC Teacher requires some PHRAN patterns that help it analyze sentences containing words like *means*, since such sentences are likely to also contain words that have no previous PHRAN definitions. In addition, UC Teacher needs to makes some assumptions about the linguistic properties of new items. For example, the PHRAN pattern-concept pair created from the first sentence above is as follows:

```
([(PERSON) (ROOT DELETE) (THING)]

 [P-O-S 'SENTENCE
  CD-FORM '(causation
             (antecedent (do (actor ?Actor)))
             (consequent
               (state-change
                  (actor ?Object)
                  (state-name physical-state)
                  (from 10)
                  (to -10))))

  ACTOR (VALUE 1 CD-FORM)
  OBJECT (VALUE 3 CD-FORM)])
```

To built this pattern, UC Teacher makes the assumption that the verb will take a person as a subject and that this person will be the cause or actor of the concept produced. The concept part is taken directly from pattern PHRAN used to understand the definiens portion of the original sentence. The resulting pattern can now be used in conjunction with the rest of PHRAN to parse sentences involving the work *delete*.

PHRAN interprets the second sentence as a statement of a plan. UC Teacher uses a PHRAN definition of verbs like *typing* that enables them be followed by a literal string. UC Teacher assumes certain conventions for this string. For example, it assumes that a word in it that it cannot parse is the name of a UNIX command, and that the particular word *filename* refers to a generic file. UC Teacher can now assert the output of this analysis, which is a quite ordinary looking fact, into the PEARL data base that contains the rest of the system's knowledge.

The third sentence, which establishes *remove* as a synonym for *delete*, is treated similarly to the first sentence.

When the questions in the last two sentences of the example are subsequently asked, UC will be able to analyze them into the appropriate conceptual content using the PHRAN patterns just created. Then the system will be able to retrieve the fact stored in the PEARL knowledge base for use in answering the question.

Currently, the example shown is as complex a situations as UC Teacher can handle. In particular, no mechanism exists for creating patterns with optional parts with more complex syntactic features. Nor is the indexing of either new pattern-concept pairs or facts about UNIX done in an intelligent manner.

11. Problems with UC

In addition the limitations associated with the various UC components that have been discussed above, there are a number of more general difficulties. Probably the most significant problem in UC involves representational issues. That is, how can the various entities, actions and relationships that constitute the UC domain best be denoted in a formal language? Of course this problem is central to AI in general, and UC's domain is rich enough for all the traditional problems to apply.

The representation used in UC has continually changed as the system has matured. A rather stable body of central concepts has emerged in this process, although a discussion of these ideas and a comparison to other systems of representation is beyond the scope of this paper.

A pragmatic problem with UC is efficiency. Currently, it takes about a minute of real time on a VAX to respond to a request, most of which seems to be spent in the context mechanism. As this is one of the more experimental components of the systems, we feel that there is a great deal of room for improvement. The context mechanism is needed only for more complex requests, thus there may be some way of avoiding the overhead inherent in its operation when it is not essential. In addition, some UC components, such as PHRAN, run considerably faster on the DEC10 than they do the VAX, after machine power is factored out. This may mean that there is room for improvement on the implementation level.

Lastly, we have not yet road tested UC with real users. Primarily, this is because we feel that the UC knowledge base is not yet large enough to guarantee a high enough hit ratio to sustain its use. We are confident that requisite extention of the knowledge base will be relatively straightforward. However, we feel less sure that the the kinds of questions we have designed UC to answer will be the ones users will find it useful to ask. We intend to collect as data the questions UC is unable to answer in its initial test runs in order to determine subsequent modifications of the system.

References

Arens, Y. 1981. Using Language and Context in the Analysis of Text. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*. Vancouver, B. C., August 1981.

Arens, Y. 1982. The Context Model: Language Understanding In Context. In *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*. Ann Arbor, Michigan, August 1982.

Brachman, R., Bobrow, R., Cohen, P. Klovstad, J., Webber, B. L., & Woods, W. A. 1979. Research in natural language understanding. Technical Report 4274, Bolt, Beranek and Newman Inc.

Burton., Richard R. 1976. Semantic Grammar: An Engineering Technique for Constructing Natural Language Understanding Systems. BBN Report No. 3453, Dec 1976.

Deering, M., Faletti, J., and Wilensky, R. 1981. PEARL: An Efficient Language for Artificial Intelligence Programming. In the *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*. Vancouver, British Columbia. August, 1981.

Deering, M., Faletti, J., and Wilensky, R. 1982. The PEARL Users Manual. Berkeley Electronic Research Laboratory Memorandum No. UCB/ERL/M82/19. March, 1982.

Faletti, J. 1982. PANDORA - A Program for Doing Commonsense Planning in Complex Situations. In *Proceedings of the Second Annual National Artificial Intelligence Conference*. Pittsburgh, PA, August 1982.

Grosz, B. J. 1977. The Representation and Use of Focus in a System for Understanding Dialogs. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*. Carnegie-Mellon University, Pittsburgh, PA.

Hayes, J. H., and Carbonell, J. G. 1981. Multi-Strategy Construction-Specific Parsing for Flexible Data Base Query and Update. In the *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*. Vancouver, British Columbia. August, 1981.

Held, G. D., Stonebraker, M. R., and Wong, E. 1975. INGRES - A relational data base system. AFIPS Conference Proceedings vol. 44, NCC.

Hendrix, Gary G. 1977. The Lifer Manual: A Guide to Building Practical Natural Language Interfaces. SRI International: AI Center Technical Note 138.

Lebowitz, M. 1980. Generalization and Memory in an Integrated Understanding System. Yale University Department of Computer Science Technical Report 186.

Riesbeck, C. K. 1975. Conceptual analysis. In R. C. Schank, *Conceptual Information Processing*. American Elsevier Publishing Company, Inc., New York.

Robinson, J. J. 1982. DIAGRAM: A Grammar for Dialogues. *Comm. ACM* Volume 25, pp. 27-47.

Schank, R. C., Lebowitz, M. and Birnbaum, L. 1980. An Integrated Understander. In *American Journal of Computational Linguistics* vol. 6 no. 1, January-March 1980.

Waltz, D. L., Finin, T., Green, F., Conrad, F., Goodman, B., and Hadden, G. 1976. The PLANES system: natural language access to a large data base. Coordinated Science Lab., University of Illinois, Urbana, Tech. Report T-34.

Wilensky, R. 1981(a). Meta-planning: Representing and using knowledge about planning in problem solving and natural language understanding. *Cognitive Science*, Vol. 5, No. 3. 1981.

Wilensky, R. 1981(b). A Knowledge-based Approach to Natural Language Processing: A Progress Report. In the *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*. Vancouver, British Columbia. August, 1981.

Wilensky, R. 1982 Planning and Understanding. Addison-Wesley. Reading, Mass.

Wilensky, R. and Morgan, M. 1981. One Analyzer for Three Languages. Berkeley Electronic Research Laboratory Memorandum No. UCB/ERL/M81/67. September, 1981.

Woods, W. A. 1970. Transition Network Grammars for Natural Language Analysis. *Comm. ACM*, Volume 13, pp. 591-606.