Knowledge Structures in UC, the UNIX* Consultant†

David N. Chin

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

ABSTRACT

The knowledge structures implemented in UC, the UNIX Consultant are sufficient for UC to reply to a large range of user queries in the domain of the UNIX operating system. This paper describes how these knowledge structures are used in the natural language tasks of parsing, inference, planning, goal detection, and generation, and how they are organized to enable efficient access even with the large database of an expert system. The structuring of knowledge to provide direct answers to common queries and the high usability and efficiency of knowledge structures allow UC to hold an interactive conversation with a user.

1. Introduction

UC is a natural language program that converses in English with users in the domain of the UNIX operating system. UC provides information on usage of system utilities, UNIX terminology, and plans for accomplishing specific tasks in the UNIX environment, all upon direct query by the user. In order to accomplish these tasks, UC must perforce have a considerable knowledge base, a large part of which is particular to the UNIX domain. The specific representations used in this knowledge base are essential to the successful operation of UC. Not only are the knowledge structures used in parsing, inference, planning, goal detection, and generation, but also the format of representation must permit the high efficiency in access and processing of the knowledge that is required in an interactive system like UC. This paper describes the details of this representation scheme and how it manages to satisfy these goals of usability and efficiency. Other aspects of the UC system are described in Arens (1982), Faletti (1982), Jacobs (1983), Rau (1983), and Wilensky and Arens (1980a and b). An overview of the UC system can be found in Wilensky (1982).

2. Specifications for the Representation

The first step in the design of knowledge structures involves determining what forms of knowledge will be needed. In this case, the domain of conversation for a UNIX consultant is specific enough that it reduces the range of knowledge structures necessary for the task. Some insight into the kinds of knowledge structures that are needed can be gleaned from looking at the variety of questions which users actually ask. Since UC is aimed at the naive user of UNIX, a majority of the queries UC receives are of the following forms (taken from actual UC sessions):

User: How can I change the write protection on my terminal?

UC: To turn on write permission on your terminal, type 'mesg y'.
To turn off write permission on your terminal, type 'mesg n'.

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: Why can't I remove the directory Tmp?

UC: The directory Tmp must be empty before the directory can be deleted.

Questions of the first form, asking how to do something, are usually requests for the names and/or usage of UNIX utilities. The user generally states the goals or results that are desired, or the actions to be performed and then asks for a specific plan for achieving these wishes. So to respond to how questions, UC must encode in its database a large number of plans for accomplishing desired results or equivalently, the knowledge necessary to generate those plans as needed.

The second question type is a request for the definition of certain UNIX or general operating systems terminology. Such definitions can be provided easily by canned textual responses. However UC generates all of its output. The expression of knowledge in a format that is also useful for generation is a much more difficult problem than simply storing canned answers.

In the third type of query, the user describes a situation where his expectations have failed to be substantiated and asks UC to explain why. Many such queries involve

---

* UNIX is trademark of Bell Laboratories

† 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.

plans where preconditions of those plans have been violated or steps omitted from the plans. The job that UC has is to determine what the user was attempting to do and then to determine whether or not preconditions may have been violated or steps left out by the user in the execution of the plans.

Besides the ability to represent all the different forms of knowledge that might be encountered, knowledge structures should be appropriate to the tasks for which they will be used. This means that it should be easy to represent knowledge, manipulate the knowledge structures, use them in processing, and do all that efficiently in both time and space. In UC, these requirements are particularly hard to meet since the knowledge structures are used for so many diverse purposes.

3. The Choice

Many different representation schemes were considered for UC. In the past, expert systems have used relations in a database (e.g. the UCC system of Douglass and Hegner, 1982), production rules and/or predicate calculus, for knowledge representation. Although these formats have their strong points, it was felt that none provided the flexibility needed for the variety of tasks in UC. Relations in a database are good for large amounts of data, but the database query languages which must be used for access to the knowledge are usually poor representation languages. Production rules encode procedural knowledge in an easy to use format, but do not provide much help for representing declarative knowledge. Predicate calculus provides built-in inference mechanisms, but do not provide sufficient mechanism for representing the linguistic forms found in natural language. Also considered were various representation languages, in particular KL-one (Schmolze and Brachman, 1981). However at the time, these did not seem to provide facilities for efficient access in very large knowledge bases. The final decision was to use a frame-like representation where some of the contents are based on Schank's conceptual dependencies, and to store the knowledge structures in PEARL databases (PEARL is an AI package developed at Berkeley that provides efficient access to Lisp representations through hashing mechanisms, c.f. Deering, et. al., 1981 and 1982).

4. The Implementation

Based on Minsky's theory of frames, the knowledge structures in UC are frames which have a slot-filler format. The idea is to store all relevant information about a particular entity together for efficient access. For example the following representation for users has the slots user-id, home-directory, and group which are filled by a user-id, a directory, and a set of group-id's respectively.

```
(create expanded person user
    (user-id user-id)
    (home-directory directory)
    (group setof group-id))
```

In addition, users inherit the slots of person frames such as a person's name.

To see how the knowledge structures are actually used, it is instructive to follow the processing of queries in some detail. UC first parses the English input into an internal representation. For instance, the query of example one is parsed into a **question** frame with the single slot, cd, which is filled by a **planfor** frame. The question asks what is the plan for (represented as a planfor with an unknown method) achieving the result of changing the write protection (mesg state) of a terminal (terminal1 which is actually a frame that is not shown).

```
(question
    (cd (planfor (result (state-change (actor terminal1)
                                       (state-name mesg)
                                       (from unspecified)
                                       (to unspecified)))
                 (method *unknown*))))
```

Once the input is parsed, UC which is a data driven program looks in its data base to find out what to do with the representation of the input. An **assertion** frame would normally result in additions to the database and an **imperative** might result in actions (depending on the goal analysis). In this case, when UC sees a question with a planfor where the method is unknown, it looks in its database for an **out-planfor** with a query slot that matches the result slot of the planfor in the question. This knowledge is encoded associatively in a **memory-association** frame where the recall-key is the associative component and the cluster slot contains a set of structures which are associated with the structure in the recall-key slot.

```
(memory-association
    (recall-key (question
                  (cd (planfor (result ?conc)
                               (method *unknown*)))))
    (cluster ((out-planfor (query ?conc)
                           (plan ?*any*)))))
```

The purpose of the memory-association frame is to simulate the process of reminding and to provide very flexible control flow for UC's data driven processor. After the question activates the memory-association, a new out-planfor is created and added to working memory. This out-planfor in turn matches and activates the following knowledge structure in UC's database:

```
(out-planfor (query (state-change (actor terminal)
                                  (state-name mesg)
                                  (from ?from-state)
                                  (to ?to-state)))
             (plan (output (cd (planfor67 planfor68))))))
```

The meaning of this out-planfor is that if a query about a state-change involving the mesg state of a terminal is ever encountered, then the proper response is the **output** frame in the plan slot. All output frames in UC are passed to the generator. The above output frame contains the planfors numbered 67 and 68:

planfor67:

```
(planfor
   (result (state-change (actor terminal)
                         (state-name mesg)
                         (from off)
                         (to on)))
   (method
      (mtrans (actor *user*)
              (object (command
                        (name mesg)
                        (args (y))
                        (input *stdin*)
                        (output *stdout*)
                        (diagnostic *stdout*)))
              (from *user*)
              (to *Unix*))))
```

This planfor states that a plan for changing the mesg state of a terminal from on to off is for the user to send the command *mesg* to UNIX with the argument "y". Planfor 68 is similar, only with the opposite result and with argument "n". In general, UC contains many of these planfors which define the purpose (result slot) of a plan (method slot). The plan is usually a simple command although there are more complex meta plans for constructing sequences of simple commands such as might be found in a UNIX pipe or in conditionals.

In UC, **out-planfors** represent "compiled" answers in an expert consultant where the consultant has encountered a particular query so often that the consultant already has a rote answer prepared. Usually the question that is in the query slot of the out-planfor is similar to the result of the planfor that is in the output frame in the plan slot of the out-planfor. However this is not necessarily the case, since the out-planfor may have anything in its plan slot. For example some queries invoke UC's interface with UNIX (due to Margaret Butler) to obtain specific information for the user.

The use of memory-associations and out-planfors in UC provides a direct association between common user queries and their solutions. This direct link enables UC to process commonplace queries quickly. When UC encounters a query that cannot be handled by the out-planfors, the planning component of UC (PANDORA, c.f. Faletti, 1982) is activated. The planner component uses the information in the UC databases to create individualized plans for specific user queries. The description of that process is beyond the scope of this paper.

The representation of definitions requires a different approach than the above representations for actions and plans. Here one can take advantage of the practicality of terminology in a specialized domain such as UNIX. Specifically, objects in the UNIX domain usually have definite functions which serve well in the definition of the object. In example two, the type declaration of a search-path includes a use slot for the search-path which contains information about the main function of search paths. The following declaration defines a search-path as a kind of functional-object with a path slot that contains a set of directories and a use slot which says that search paths are used in searching for programs by UNIX.

```
(create expanded functional-object search-path
   (path setof directory)
   (use ($search (actor *Unix*)
                 (object program)
                 (location ?search-path)))
   . . . )
```

Additional information useful in generating a definition can be found the slots of a concept's declaration. These slots describe the parts of a concept and are ordered in terms of importance. Thus in the example, the fact that a search-path is composed of a set of directories was used in the definition given in the examples.

Other useful information for building definitions is encoded in the hierarchical structure of concepts in UC. This is not used in the above example since a search-path is only an expanded version of the theoretical concept, functional-object. However with other objects such as directory, the fact that directory is an expanded version of a file (a directory is a file which is used to store other files) is actually used in the definition.

The third type of query involves failed preconditions of plans or missing steps in a plan. In UC the preconditions of a plan are listed in a **preconds** frame. For instance, in example 3 above, the relevant preconds frame is:

```
(preconds
   (plan (mtrans (actor *user*)
                 (object (command
                           (name rmdir)
                           (args (?directoryname))
                           (input stdin)
                           (output stdout)
                           (diagnostic stdout)))
                 (from *user*)
                 (to *Unix*)))
   (are ((state (actor
                  (all (var ?file)
                       (desc (file))
                       (pred (inside-of
                               (object
                                 ?directoryname))))))
                (state-name physical-state)
                (value non-existing))
        . . . )))
```

This states that one of the preconditions for removing a directory is that it must be empty. In analyzing the example, UC first finds the goal of the user, namely to

delete the directory Tmp. Then from this goal, UC looks for a plan for that goal among planfors which have that goal in their result slots. This plan is shown above. Once the plan has been found, the preconds for that plan are checked which in this case leads to the fact that a directory must be empty before it can be deleted. Here UC actually checks with UNIX, looking in the user's area for the directory Tmp and discovers that this precondition is indeed violated. If UC had not been able to find the directory, UC would suggest that the user personally check for the preconditions. Of course if the first precondition was found to be satisfied, the next would be checked and so on. In a multi-step plan, UC would also verify that the steps of the plan had been carried out in the proper sequence by querying the user or checking with UNIX.

5. Storage for Efficient Access

The knowledge structures in UC are stored in PEARL databases which provide efficient access by hash indexing. Frames are indexed by combinations of the frame type and/or the contents of selected slots. For instance, the planfor of example one is indexed using a hashing key based on the state-change in the planfor's result slot. This planfor is stored by the fact that it is a planfor for the state-change of a terminal's mesg state. This degree of detail in the indexing scheme allows this planfor to be immediately recovered whenever a reference is made to a state-change in a terminal's mesg state.

Similarly, a memory-association is indexed by the filler of the recall-key slot, an out-planfor is indexed using the contents of the query slot of the out-planfor, and a preconds is indexed by the plan in the plan slot of the preconds. Indeed all knowledge structures in UC have associated with them one or more indexing schemes which specify how to generate hashing keys for storage of the knowledge structure in the UC databases. These indexing methods are specified at the time that the knowledge structures are defined. Thus although care must be taken to choose good indexing schemes when defining the structure of a frame, the indexing scheme is used automatically whenever another instance of the frame is added to the UC databases. Also, even though the indexing schemes for large structures like planfors involve many levels of embedded slots and frames, simpler knowledge structures usually have simpler indexing schemes. For example, the representation for users in UC are stored in two ways: by the fact that they are users and have a specific account name, and by the fact that they are users and have some given real name.

The basic idea behind using these complex indexing schemes is to simulate a real associative memory by using the hashing mechanisms provided in Pearl databases. This associative memory mechanism fits well with the data-driven control mechanism of UC and is useful for a great variety of tasks. For example, goal analysis of speech acts can be done through this associative mechanism:

```
(memory-association
     (recall-key (assertion (cd (goal (planner ?person)
                                      (objective ?obj)))))
     (cluster ((out-planfor (cd ?obj)))))
```

In the above example (provided by Jim Mayfield), UC analyzes the user's statement of wanting to do something as a request for UC to explain how to achieve that goal.

6. Conclusions

The knowledge structures developed for UC have so far shown good efficiency in both access time and space usage within the limited domain of processing queries to a Unix Consultant. The knowledge structures fit well in the framework of data-driven programming used in UC. Ease of use is somewhat subjective, but beginners have been able to add to the UC knowledge base after an introductory graduate course in AI. Efforts underway to extend UC in such areas as dialogue will further test the merit of this representation scheme.

7. Technical Data

UC is a working system which is still under development. In size, UC is currently two and a half megabytes of which half a megabyte is FRANZ lisp. Since the knowledge base is still growing, it is uncertain how much of an impact even more knowledge will have on the system especially when the program becomes too large to fit in main memory. In terms of efficiency, queries to UC take between two and seven seconds of CPU time on a VAX 11/780. Currently, all the knowledge in UC is hand coded, however efforts are under way to automate the process.

8. Acknowledgments

Some of the knowledge structures used in UC are refinements of formats developed by Joe Faletti and Peter Norvig. Yigal Arens is responsible for the underlying memory structure used in UC and of course, this project would not be possible without the guidance and advice of Robert Wilensky.

9. References

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

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

Douglass, R., and S. Hegner. 1982. An Expert Consultant for the Unix System: Bridging the Gap Between the User and Command Language Semantics. In the *Proceedings of the Fourth National Conference of Canadian Society for Computational Studies of Intelligence.* University of Saskatchewan, Saskatoon, Canada.

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

Rau, L. 1983. Computational Resolution of Ellipses. Submitted to IJCAI-83, Karlsruhe, Germany.

Jacobs, P. 1983. Generation in a Natural Language Interface. Submitted to IJCAI-83, Karlsruhe, Germany.

Schmolze, J. and R. Brachman. 1981. *Proceedings of the 1981 KL-ONE Workshop.* Fairchild Technical Report No. 618, FLAIR Technical Report No. 4. May, 1982.

Wilensky, R. 1982. Talking to UNIX in English: An Overview of UC. In the *Proceedings of the National Conference on Artificial Intelligence.* Pittsburgh, PA. August, 1982.

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., and Arens, Y. 1980(a). PHRAN – a Knowledge-Based Natural Language Understander. In the *Proceedings of the 18th Annual Meeting of the Association for Computational Linguistics.* Philadelphia, PA.

Wilensky, R., and Arens, Y. 1980(b). PHRAN – a Knowledge Based Approach to Natural Language Analysis. University of California at Berkeley, Electronic Research Laboratory Memorandum No. UCB/ERL M80/34.