An Expert Consultant for the UNIX Operating System: Bridging the Gap Between the User and Command Language Semantics

Robert J. Douglass
C-10, MS B296
Los Alamos National Laboratory
Los Alamos, N.M. 87545

Stephen J. Hegner
Department of Applied Mathematics and
Computer Science
Thornton Hall
University of Virginia
Charlottesville, Va. 22901

1. INTRODUCTION

The UNIX Computer Consultant (UCC) is an expert system currently under development at the Los Alamos National Laboratory and at the University of Virginia. It is designed to aid both novice and advanced users of the UNIX operating system in their use of the system.

UCC is far more powerful than the typical operating system help utility, which provides on-line retrieval only of documentation of specifically named commands. Thus, a user who does not know the name of the command he needs, or who needs information on items other than command-specific definitions (such as information on the organization of the operating system's file system), can get little or no help from such a help utility. UCC, on the other hand, contains a sophisticated retrieval facility that can key on a number of concepts, including not only specific command definitions, but also logical operations without a given command (e.g. listing a text file) and concept definitions (e.g. a UNIX path). Furthermore, UCC, unlike existing help systems, can respond with a specific answer given a specific question.

Because UCC is designed to be a consultant for an existing system, it is not integrated into specific utilities, but rather exists as an independent susbsystem on top of the opertating system. This is in contrast to some systems that incoprporate user support facilities within the utility. While the direct integration approach shows much promise for help facilities for newly designed systems, its applicability to existing systems (such as UNIX) is made difficult by the necessity of rewriting part of the existing system, in addition to the entire help facility. Furthermore, such direct incorporation of help features make the answering of queries not keyed on specific commands very difficult.

1.1. An Overview of UCC

The UCC system consists of two major modules, the front end module and the UNIX knowledge base and solver module. The major role of the front end is to translate a user's natural language query into the specially designed formal

query language UCCquel, and to translate answers to UCCquel queries back into natural language. The role of the UNIX knowledge base and solver module is to produce answers to the formal UCCquel queries.

The front end module accepts questions from users usch as "How do I list a directory file?" and calls a lexical analyzer to delete noise words from the question, convert synonyms and tensed words to a standard token, and identify unknown words. The tokenized question is then parsed by an ATN, phrase by phrase if an attempt to parse an entire sentence fails. The ATN produces three pieces of information: a) the type of question; b) a template, called a case frame, describing the main operating system action mentioned in the question; and c) a set of predicates, deriverd from noun phrases, prepositional phrases, adverbs, and other modifying phrases and clauses, that assert properties of entities referred to in the question (such as files).

The parser passes its output to a query generator, which produces: a) a formal query with a set of predicates describing a function to be performed by the operating system; b) preconditions describing the input to that function; and c) postconditions describing the function's output. The formal query will contain unbound variables that must be instantiated by the knowledge base and solver module to answer the original question.

The knowledge base and solver module contains descriptions of UNIX commands and concepts, and a set of propositional semantic defintions of commands similar to Hoare semantics [1]. The descriptions are used to answer questions such as "What is a directory file?", or "What does 'ls' do?" The formal semantic definitions are used to answer questions such as "How do I list a directory file?"

Once the variables in a query have been bound by the knowledge base, then the instantiated query is passed back to the front end to produce an answer in English for the user. The front end formulates an answer by noting what type of question was asked, selecting an appropriate template for an answer, and using a dictionary of predicate definitions to describe the relevant parts of the instantiated query.

In this paper, we discuss briefly the nature of the formal query language UCCquel, and then we detail the process of transforming a natural-language query into UCCquel. In a companion paper we will detail the workings of the UNIX knowledge base and solver module.

2. QUERY MODELLING AND CLASSIFICATION

To understand the operation of the natural-language front end, it is first necessary to understand the formal queries that it is to produce. In this section, we outline the nature of the various queries that are consisdered and show how they are represented in UCCquel.

2.1. Static Queries

Static queries are the most basic type that are handled by UCC. A static query is one that requests information not involving system dynamics, and, expressed in natural language, are typically of the "What is ... ?" variety. Examples include "What is a pipe?" and "What is a home directory?" In each case, a simple retrieval of a definition is all that is necessary. In essence, a static query is just a retrieval of the form "get {x | P(x)} " (which is the form of a

nonprocedural query to a relational database [2]). For example, "What is a pipe?" translates into the formal query "get {x | x is the definition of a pipe }.

All queries in UCCquel are expressed in a Lisp-like notation. Our representation of this query looks like:

```
(Find X
   (Basic-object-description ("pipe") = X)).
```

X is the free variable that must be bound in answering the query.

2.2. Dynamic Queries

To be useful, UCC must also have the capability of processing dynamic queries. These queries differ from their static counterparts in that they deal with the dynamic aspects of the UNIX operating system. Examples include "How do I list the contents of a file paged and with page headers?" and "What happens if I try to print a directory file with the pr command?" In each case, the query deals with the determination or action of a system command.

We may still classify such queries as "get { x | P(x) }", provided we choose P appropriately. In the case of a static query, P is just a well-formed formula (wff) in the (ordinary) logic that describes the retrieval of concept definitions. In the case of dynamic queries, P must be replaced by a wff in an appropriate dynamic logic, since we are now dealing with system dynamics. For the purposes of UCC, the usual propositional-style semantics, commonly used in programming language semantic definition [1], provide a convenient framework. A wff in this style of semantics takes the general form: {R} F {Q}, where R and Q are wffs in an appropriate static first order logic (known as the underlying base logic), and F is an action that may change the truth values of statements in the base logic.

As a particular example, consider the query given above: "How do I list the contents of a file paged and with page headers?" Here R is of the form "#x is a (generic) text file" and Q is of the form "The contents of the standard listing device is what the contents of the file #x was before the command, modified to be paged with standard page headers." F is an unknown operator, to be found in answering the query. In the retrieval, P(x) is {R} X {Q}.

2.3. Fundamental Types of Dynamic Queries

A dynamic query of the form "get { x | P(x) }", where P = {R} F {Q} may be placed into one of 8 possible classifications, depending upon which subset of R, F, Q is known and which is not. In the initial implementation of UCC, only two of these will be considered.

(1) R and Q are known, but F is not. This is the general structure of a "How do I ..." type of query, and is illustrated by the example in section 2.2.

(2) R and F are known, but Q is not. This is the general structure of a "What happens if ... ?" type of query. As an example, once again consider "what happens if I try to print a directory file using the 'pr' command?" Here R is "#x is a (generic) directory file" and F is "pr #x", with Q to be found.

2.4. Secondary Responses in Dynamic Queries

While the classification of dynamic queries as outlined in the previous section is a useful guideline, it is not always completely adequate. For example,

consider once again the query "How do I list the contents of a file paged and with page headers?" To answer this query with R and Q as given in section 2.2, it is not sufficient to simply supply the response "pr #x", since, for this response to be correct, we must have that #x is a file which is currently readable by the user. In responding with the answer, we must add this readability condition to the preconditions R. We term such an addition a secondary response. In the fromalization of dynamic queries in UCCquel, we always permit the addition of appropriate secondary responses.

2.5. An Example UCCquel Query

Below is displayed the UCCquel query for "How do I list the contents of a file paged and with page headers?"

```
(Find (R1 F1 Q1)
  (Dyn R F Q)
    (Define R
      ((Clauses
         (file #x (type = "text")))
       (addnecessary R1)))
    (Define F
      (( transform F1)
       (logical (list-text-file))))
    (Define Q
      ((Clauses
         (contents(%user-terminal)
           = modified (contents (time (-1,file(#x)))
                 (type = "text")
                 (paged = "yes")
                 (pageheaders = "yes")
                 (value standard))))
       (addimplications Q1))))
```

P1, F1, and Q1 are the variables to be bound in the solution of the query. (Dyn R F Q) means {R} F {Q}. F1 is the fundamental variable to be bound, and is declared to be a transform. P1 and Q1 are to be bound to secondary preconditions and postconditions, respectively. The definitions of R and Q are wffs in our underlying logic known as the static definition logic. The "time(-1 ...)" notation in the postcondition Q is used to ensure that the value of the user terminal after the execution of the command will be the contents of #x before the command. This is merely a shorthand and does not violate the constraint that R and Q be statements in a static logic; by setting #x to a dummy file @x in the preconditions and using file @x instead of time(-1,file (#x)), we can stay entirely within the constraints. The "logical ... " part of the definition of F is indicating which case frame was used in constructing the query, and will be mentioned again in the next section.

2.6. Specific Queries

In the example query illustrated above, the file to be listed is not specified, and so is represented as a generic text file, #x. If the user instead had asked "How do I list the contents of my file /bin/paper paged and with page headers?", the formulation would differ fundamentally in that #x would be replaced by

/bin/paper. Also, the assumption that /bin/paper is a text file must be dropped. Rather, it must be left to the knowledge base to decide whether or not /bin/paper is listable.

3. THE INTERFACE BETWEEN THE USER AND THE KNOWLEDGE BASE

The front end module is responsible for converting a user's English-questions into formal UCCquel queries, and formulating English responses. This process consists of five levels of analysis: lexical (or word), phrase, clause, sentence, paragraph(or dialogue), and topic. Five major data structures are used by these levels, including a dictionary, a set of case frames, context registers, predicate descriptions, and answer frames. We will examine these levels and data structures more closely as we follow the front-end processing of the question "How do I list a file paged and with page headers?" This question is the input to the front end module and the output is the UCCquel query given in section 2.5. Only the analysis of a user's question will be discussed here; we leave a discussion of the more straightforward process of generating an answer to a future paper.

3.1. Lexical Analysis

Lexical or word level analysis is performed by a tokenizing routine. Each time it is called by the parser, it scans the English question and returns the next token. The tokenizer looks up words in the dictioinary to see if they are defined and replaces them wtheyh standard tokens if it are found. It catches multiword idioms and common noun-noun modification, such as "how do I" or "directory file"; it also flags unknown words, identifies file names when possible, replaces synonyms with a standard root, and replaces tensed and pluralized words with a standard token plus the features indicating tense or number.

The tokenizer uses a token definition stored in the dictionary and a look-ahead mechanism to determine what English words or UNIX file names should be grouped into one token. Words with no dictionary entry are passed along to the ATN, because they may be names of specific files.

For example, given our sample question "How do I list ...", the tokenizer would return the following tokens (the $ designates a token): $hwd (representing the phrase "how do I"), $i (for "I"), $list, $a, $file, $with, $phead (plural) (for page headers), $and, and $paged.

3.2. Phrase Level Analysis

Tokenized questions are parsed, phrase by phrase, using a grammar represented by an augmented transition network (ATN). ATNs are a standard tool for parsing natural language [2]. Although other parsing techniques have proved useful for parsing restricted English questions [3,4], ATNs are very versatile, parsing statements that include ellipsis and grammatical errors.

UCC's ATN produces a parse tree for a question, either for a whole sentence or phrase-by-phrase. It uncovers and enforces syntactic rules and semantic constraints within a phrase. Predicates are generated by the ATN to represent the meaning of the phrases. These predicates form the preconditions, {R}, and

postconditions, {Q}, for dynamic queries.

The ATNs also select a small set of case frame candidates that could correspond to the main action described in a clause. Case frames and their role are described in the next section.

A parse of our sample question would produce the following results:

(1) Question Type: hwd (meaning "how do I").

(2) Phrase Level Semantics:

Noun groups: (NG1 (file #x) (type = "text") (paged = "yes")) (NG2 (pageheaders)
Verb groups: (VG1 (verb list) (direct-object NG1) (tense present))
Preposition groups: (PG1 (with NG2))

(3) Case Frames:

(logical list-text-file)
(logical list-directory)
(logical enumerate)

3.3. Clause Level analysis

The ATN parses the phrases in a question to produce predicates describing the noun groups and preposition groups. These phrases are grouped at the clause level by a "case-frame-fitter". Case frames are templates representing the main action of a clause and the constituents of the action, such as the actor and receipient of the action; usually they correspond to the main verb in the clause.

Several natural-language understanding systems have used case frames to represent the action in a sentence [2,5]. For Schank and his coworkers, a small set of conceptual case frames represent all actions expressed in natural language [5]. In UCC, case frames correspond to logical operations in an operating system, and they form the main link between English-language operating system concepts and the formal semantic definitions of specific UNIX commands. Some have a direct correspondence to UNIX commands, while others may be associated with several different commands that could be used to accomplish the same logical operation.

For example, as shown in the last section, the verb "list" could represent one of three case frames, list-text-file, list-directory, or enumeration. List-text-file is associated in the knowledge base with three UNIX commands for listing a text file, "cat", "pr", and "more". The case-frame-fitter must select one of the three possible case frames using the parsed question and semantic constraints specified in the case frame, and the knowledge base and solver module must decide which UNIX operator associated with the list-text-file case frame, "cat", "pr", or "more", is the appropriate answer to the user's question.

Case frames contain default information on preconditions and postconditions associated with the logical action represented by the case frame. They also specify slots to be filled from a user's question or from context, and they provide semantic constraints on what information can fill those slots.

In our example, the case-frame-fitter selects the list-text-file case frame because the direct object of "list", identified by the ATN as the noun group "a file", is assumed to be a text file. The list-directory case frame would have been selected if the object of "list" had been a directory, and the enumeration case frame would have been picked if the object had been a noun group that could be

enumerated, such "the number of users currently logged on the system."

Once the list-text-file case frame is choosen, the case-frame-fitter notes that it specifies that the question should contain phrases describing how the text is listed. In our example, the case-frame-fitter finds the phrases "paged and with page headers" and fits these into slots that modify the description of the result of the logical operation list-text-file. This description constitutes the postconditions, {Q}. For our example, the preconditions will be the direct object of "list" or the phrase "a file". The case frame specifies that the preconditions must be "contents(file (#x) (type = "text"))", and this is consistent with the ATN's parse of the noun group, "a file". Since "a file" is parsed as an indefinite noun group, the case-frame-fitter binds the preconditions to be a generic file, #x.

3.4. Sentence Level Analysis

Once a case frame has been selected for each clause in a question and the case-frame-fitter has formed {R} and {Q} from the parsed phrases, then the "query-generator" is called to produce a formal UCCquel query. For simple questions (questions stated in one clause) the formal query is essentially built by the case-frame-fitter, and the query-generator only has to verify it with the user and pass it along to the knowledge base and solver module. The formal query generated for our example was given in section 2.5.

The query-generator must integrate several clauses into one formal query in the case of questions that are stated in several clauses (such as "If I ..., then ... ?") and questions stated in several sentences (such as "I have a directory file that I made read only. Why can't I list it?"). Such questions will produce several instantiated case frames, and the query-generator must build one query out of them. Usually, multiple clauses serve the function of further describing the preconditions of the action in the main clause.

3.5. Dialogue and Paragraph Level Analysis

Work with natural-language database systems and expert consultants [2,6,7] has shown that users usually ask several related questions and that the system must be able to maintain a dialogue with the user and use intersentence (or paragraph level) context. UCC saves contextual information using "context registers" similar to those used by Waltz [2] and by using clarification dialogue.

Context registers are push down stacks with a short fixed length. They hold information derived from previous questions and answers, such as the last case frames referenced, the pre- and postconditions of previous questions, and the subjects and objects of previous questions. Ellipsis (missing portions of a question) and pronouns are common in questions, and UCC must use the case frames to fill in this information. If the case-frame-fitter or query-generator finds that a portion of a case frame or query is missing from a question, then they consult the context registers to find the most recent piece of information that will satisfy the semantic constraints of the current question. For example, if the user asks the question "How do I list a file?", and then asks "With page headers?", UCC must resolve the ellipsis in the second question by poping off the most recently used case frame and preconditions from the appropriate context registers.

UCC maintains a dialogue with the user by answering his questions, saving the previous questions in context registers, and using clarification dialogue to verify that a user's question has been correctly understood. In our example,

once the query-generator has formed the query, it would verify the query with the user with the following dialogue:

```
UCC: Your query is understood as:
     Find a command that accepts as input:
      1a. a text file.
     and the command will produce output:
      2a. at your terminal
      2b. has page headers
      2c. is paged.
    Is this all right? (yes/no):
USER: no
```

Since the user responded "no", UCC will reply with the following:

```
UCC:  Do you wish to:
      1. rephrase the question entirely?
      2. change an item?
     Enter 1, 2:
USER: 2
UCC:  Which item(s), 1a, 2a, 2b, or 2c:
USER: 2a
UCC:  What is the new condition?
USER: output goes to the spinwriter
UCC:  Any other changes?
USER: no
UCC:  Your new query is understood as ...
```

Once UCC has verified a query, it is shipped off to the knowledge base and solver module to produce an answer.

3.6. The Topic Level

Users usually ask several questions about a particular UNIX topic, such as how to use the editor, manipulating files, or compiling a program. In order to successfully parse and answer questions about a variety of topics, the knowledge base, vocabulary, and case frames have been partitioned into different topic areas. At present, UCC contains information only about the file system and the command language. We are experimenting with a menu-driven system for switching between topics. Complex natural-language processing and question answering requires that questions have a limited and well defined context and that the case frames, vocabulary, and knowledge base be partitioned by topics.

4. CONCLUSION

Natural-language front ends for database systems have been in existence for several years as have expert consultant systems. UCC combines both of these lines of research into a single system that goes well beyond typical operating system help facilities to provide an expert consultant with sophisticated natural-language understanding ability.

We have outlined the process of producing a formal query from a natural-language question and shown how queries can be formally modelled in the UCCquel query language. UCC successfully bridges the gap between users' English questions and formal command definitions by dividing the task into levels of analysis from lexical to topic, and by partitioning the major data structures of UCC by topic. In this way UCC achieves generality and sophisicated language understanding over a broad range of topics. It is intended that UCC serve as a model for the design of expert consultants for other operating systems and for systems that, like operating systems, consist of collections of processes.

5. ACKNOWLEDGMENTS

The authors would like to acknowledge the contribution of David Stotts, Andrew Lacy, Kelton Flinn, and John Taylor to the development and implementation of UCC.

6. REFERENCES

[1] Alagic, S. and M. Arbib, *The Design of Well-Structured and Correct Programs*. Springer-Verlag, 1978.

[2] Waltz, D., "An English Language Question Ansering System for a Large Relational Database," *CACM, 21, 7*, 1978, pp. 526-539.

[3] Harris, L., "User Oriented Data Base Query with the Robot Natural Language Query System," *International Journal of Man-Machine Studies, 9*, 1977.

[4] Hendrix, G., E. Sacerdoti, D. Sagalowicz, and J. Slocum, "Developing a Natural Language Interface to Complex Data," *ACM Transactions on Database Systems*, 1978.

[5] Schank, R., "Identification of Conceptualizations Underlying Natural Language," *Computer Models of Thought and Language*. R. Schank and K. Colby, eds., W. H. Freedman and Company, 1973, pp. 187-248.

[6] Shortliffe, E., *Computer-Based Medical Consultations: MYCIN*. American Elsevier, New York, 1976.

[7] Michie, D. (ed.), *Expert Systems in the Micro Electronic Age*. Edinburgh University Press, Edinburgh, 1979.