Knowledge Base Design for an Operating System Expert Consultant

by

Stephen J. Hegner
Department of Computer Science and Electrical Engineering
Votey Building
University of Vermont
Burlington, VT 05405

and

Robert J. Douglass, Jr.
C-10, MS B296
Los Alamos National Laboratory
Los Alamos, NM 87545

To appear in the Proceedings of the Fifth National Conference of the CSCSI/SCEIO, London, Ontario, May 15-17, 1984.

This work was performed in part under the auspices of the United States Department of Energy.

Abstract

Yucca is an operating system expert consultant currently under development at Los Alamos National Laboratory and the University of Vermont. It consists of two modules, the *natural-language front end*, which translates natural-language queries to and from the formal query language *Quelya*, and the *knowledge base*, which uses a completely formal representation of the behavior of the operating system to answer the Quelya queries submitted by the front end. In this paper, the overall architecture of the knowledge base is outlined.

1. Introduction

Yucca is an operating system expert consultant currently under development at Los Alamos National Laboratory and the University of Vermont. It is designed to provide expert help to experienced computer users who are encountering a new operating system for the first time. To automate such assistance, a utility must go beyond the capabilities of ordinary operating system help facilities in several respects. First of all, the user must not need to know the name of the appropriate command to get help. In other words, the system must be able to answer questions of the "How do I ...?" variety. For example, a user desiring to delete a file must not need to know the name of the delete command in order to get further help. Second, the expert helper must be able to provide information about the static structure of the system; questions of the "What is ...?" variety must be addressed. For example, the user may wish to know what a pipe is (in UNIX[1]). Third, the expert helper must have enough knowledge of other systems to be able to provide meaningful responses to questions not directly applicable to the current system. For example, a TOPS-20[2] user may ask a UNIX consultant about online file expiration. Yucca has been designed to take all of these issues into account.

Communication with the user is in natural English. Any other interface would require the user to learn the language of the expert system, which would be self-defeating. Yucca is a utility which exists as an independent subsystem of a currently existing operating system. This is in contradistinction to systems such as Cousin [Haye82a,82b] and Sara [FeEs82],

---

[1] UNIX is a trademark of Bell Laboratories.

[2] TOPS-20 is a trademark of Digital Equipment Corporation.

which are integral help systems and which involve rewriting parts of the command language interpreter. Such integral help systems can provide detailed information on how to use a given utility, but they require that the user know which utility he needs, and thus violate the first condition listed above.

Yucca is designed to aid in the use of basic commands; it is not intended to be a command language version of the Programmer's Apprentice [Wate82]. The type of answers which it supplies is limited to those which can be accomplished using a single command, or a simple sequence (UNIX piping) of commands.

Yucca is an outgrowth of the earlier system UCC, which was reported in [DoHe82]. It is currently being implemented in Franz Lisp for BSD 4.2 UNIX running on VAX[3] hardware. However, the design also encompasses basic concepts from other operating systems, such as TOPS-20, so that users switching from such systems can have their questions answered intelligently.

2. Design Rationale

The behavior of operating system commands is completely and formally describable. Once a query is understood, the problem is one of finding a correct answer; there is no concept of degree of correctness. To exploit this structure of knowledge, we have divided the system into two distinct but interconnected modules. The *knowledge base* provides a completely formal representation of the behavior of the operating system. It receives input in the form of formal queries in the specially designed query language *Quelya* and produces output which binds the variables of the query. The interface of the user to this knowledge base is provided by a separate module, the *natural-*

---

[3] VAX is a trademark of Digital Equipment Corporation.

language front end. The role of this module is to translate natural-language queries into equivalent Quelya requests and to convert bindings of these requests into natural-language answers. Thus, its purpose is similar to that of the natural-language components of database systems such as Planes [Walt78] and Team [Gros83], although the nature of the formal queries makes its design quite different. We note that this philosophy is distinctly different from that used in the UNIX consultant UC [Wile82], in which the natural-language module is deeply involved in knowledge representation.

In this paper, the overall architecture of the knowledge base is outlined. Details of the natural-language front end will be discussed in a separate report.

3. The Nature of Formal Queries

In a conventional (nonprocedural) relational database query language, answering queries corresponds to the process of binding free variables in a first-order formula. For example, retrieving the set of all employees who make at least $30000/year may be expressed as

$$\{ x \mid employee(x) \wedge (salary(x) \geq 30000) \} .$$

In Quelya, there are two basic query formats, static and dynamic. *Static queries* are analogous to conventional database queries and are used to retrieve simple definitions. For example, the natural-language query "What is a pipe?" would be represented in Quelya as

(Find x (x = description("pipe"))),

which is the Lisp-like Quelya notation for

$$\{ x \mid x = description("pipe") \}.$$

Just as static queries make use of a first-order static logic, *dynamic queries* make use of a first-order dynamic logic. Statements in such a logic

follow the general flavor of Hoare-style propositional semantics [AlAr78]. The query "How do I list the contents of a file with control characters displayed?" is represented in Quelya as

```
(Find (P1 F1 Q1)
   (Dyn (P F Q)
      (Define P
         ((Clauses
             (Existsfile(#X))
             (type(file(#X)) = "plain")
             (Addnecessary P1))))
      (Define F
         ((transform F1))
      (Define Q
         ((Clauses
             (Contents(file(standard-output(%user)))
                    := contents(file(#X))
             (visible-nonprinting-characters(contents(
                           file(standard-output(%user)))) = "yes")
             (Addimplications Q1))))).
```

Here P represents the *precondition* of the action, F the action, and Q the postcondition of the action. This is represented in more familiar notation as {P}F{Q}. The string #X designates a generic file path, while %user represents the terminal connection of the current user. P1, F1 and Q1 are the variables to be bound in the query. F1 names the action and is the primary variable to be bound. It will be bound to "cat -v" for the above example. P1 is bound to any *secondary preconditions*, in this case, to a statement that the file of #X must be readable and the standard output device of the user must be writeable. Similarly, Q1 to be is bound to any *secondary postconditions*. The assignment ":=" in the postconditions relates values before the action to those after; it can be formally eliminated by introducing new variables assigned in the preconditions.

This format may be used to express a wide variety of dynamic query types. However, at the present time, Yucca is only equipped to answer dynamic queries of the "How do I..?" variety, in which the action is the primary item to be bound.

4. Fundamental Components of the Knowledge Base

The knowledge base consists of two components, the encyclopedia and the formal semantics module. The *encyclopedia* is comprised of simple English language paragraphs describing key items, such as pipes and the overall file structure. It is used in a completely straightforward fashion to answer static queries and will not be further discussed in this paper.

The *formal semantics module* is comprised of the object and type definitions, the static predicates, the command semantics, and the item templates. The *object definitions* form the central core of the knowledge base. Objects currently modelled include files, directories, devices, terminal connections, and users. Each object definition consists of a hierarchically organized set of attributes. Part of the definition for the object type *file* is given below.

```
Intrinsic attributes:
     node-definitions [set-of(file-node)]
     owner [user-id]
     mode:
          protection [protection-type]
          type [file-type]
     contents [sequence-of(byte)]

Extrinsic subattributes of contents:
               (condition (type = "plain" or type = "device"))
     numbered-lines [linetype]
     visible-nonprinting-characters [yes-no]
     end-lines-marked [yes-no]
     tabs-marked [yes-no]
     single-spaces [yes-no]
```

Intrinsic attributes are those which any instance of the object must have at all times (whether or not the value is actually known). *Extrinsic attributes*, on the other hand, can only have values when an explicit action has taken place. The term in square brackets next to each attribute is the *type* of the attribute; each such type is declared as a data type in the knowledge base.

The *static predicates* are used to describe states of instances of the various objects. For example, there is a static predicate *Readable*, which takes two arguments, a path definition and a terminal connection. Readable(P,c) is true if and only if the user at terminal connection c has read privileges on the file defined by path P.

Command semantics are represented propositionally. As an example, the semantic description of the basic (one file) UNIX cat command is given below.

```
Operation: HL-cat(t1,t2)
              t1 = path
              t2 = terminal-connection

Preconditions:
     AND : ¦ Readable(t1,t2)
           ¦ Writeable(standard-output(t2),t2)

Postconditions:
     AND : ¦ contents(file(standard-output(t2))) := contents(file(t1))
           ¦ All "yes-no" attributes of
                        contents(file(standard-output(t2))) = "no"
           ¦ numbered-lines(contents(file(standard-output(t2)))) = "none"
           ¦ OK-cat-mods = "yes"
           ¦ Avail-cat-options-list = Cat-parameters
           ¦ Used-cat-options-list = ϕ
```

Note that all extrinsic attributes of contents are initially set to "null" values. To obtain additional attributes of the target file, *filters* for the cat command are logically appended to the basic action. A description of the cat-visible-nonprinting-characters filter ("-v" option) is given below.

```
Preconditions:
     AND : ¦ OK-cat-mods = "yes"
           ¦ Member("visible-nonprinting-characters",Avail-cat-options-list)

Postconditions:
     AND : ¦ Avail-cat-options-list := Avail-cat-options-list \
                                       {"visible-nonprinting-characters"}
           ¦ visible-nonprinting-characters
                    (contents(file(standard-output(t2)))) = "yes"
           ¦ used-cat-options-list := used-cat-options-list ∪
                                      {"visible-nonprinting-characters"}
```

The first precondition makes sure that it is all right to append filters pertaining to the cat command, while the second removes the applied feature

from the available options list, to make sure that it is not used twice. By properly adding and deleting from the avail- and used- options lists of commands, complex option interaction has been modelled.

The powerful UNIX feature of input and output redirection is modelled in a similar fashion, via filters which change the values of standard input and standard output.

The description of the cat command contains all of the information needed to answer the formal query presented in the previous section; once P1 is bound to Readable(#X,%user)∧Writeable(standard-output(%user),%user), all of the postconditions are satisfied by appending the above filter to the basic cat command and defining F1 to be this composition.

For each attribute of each object, there is an *item template* which indexes commands pertinent to that attribute. In our example, the item template of interest is that which corresponds to the visible-nonprinting-characters attribute of the file object type. This template links this object attribute to the visible-nonprinting-characters filter of the cat command. Thus, the *solver module* of the knowledge base can, upon examining the postconditions of the query, quickly zero in on the appropriate command and filter(s).

A key feature of the object descriptions is the inclusion of attributes which are applicable to other systems, but inapplicable to the extant system. For example, the attribute *online-expiration-time* is included in the instrinsic attributes of file. This attribute, meaningless in UNIX but central in TOPS-20, is connected to an item template which contains an explanation of its inapplicability. Thus, the query "How do I determine the online expiration date of a file?" is answered with a statement indicating the inapplicability of this concept, rather than a statement of the form "Question

not understood."

References

[AlAr78] Alagić, S., and M. A. Arbib, *The Design of Well-Structured and Correct Programs*, Springer-Verlag, 1978.

[DoHe82] Douglass, R. J., and S. J. Hegner, "An expert consultant for the UNIX operating system: bridging the gap between the user and command language semantics," *Proc. 1982 CSCSI/SCEIO Conference, Saskatoon, May, 1982.*

[FeEs82] Fenschel, R. S., and G. Estrin, "Self-describing systems using integral help," *IEEE Trans. Systems, Man, and Cybernetics*, **12**(1982), pp. 162-167.

[Gros83] Grosz, B. J., "TEAM, a transportable natural language interface system," *Proc. 1983 Conf. on Applied Natural Language Processing*, pp. 39-45.

[Haye82a] Hayes, P. J., "Uniform help facilities for a cooperative user interface," *Proc. 1982 NCC, Houston.*

[Haye82b] Hayes, P. J., "Cooperative Command Interaction through the Cousin System," *Proc. Intl. Conf. Man/Machine Systems, U. of Manchester Institute of Science and Technology, July, 1982.*

[Walt78] Waltz, D., "An English language question answering system for a large relational database," *Comm. ACM*, **21**(1978), pp. 526-539.

[Wate82] Waters, R. C., "The Programmer's Apprentice: knowledge based program editing," *IEEE Trans. Software Engineering*, **8**(1982), pp. 1-12.

[Wile82] Wilensky, R., "Talking to UNIX in English: an overview of UC," *Proc. 1982 AAAI Conf.*, pp. 107-110.