Programs as data for their help systems

by ELAINE A. RICH

The University of Texas at Austin
Austin, Texas

ABSTRACT

The goal of this research is to develop ways of representing the knowledge available to a help system in such a way that the system can actually reason with the knowledge rather than being restricted to simply retrieving and presenting stored answers to a restricted and anticipated class of questions. One kind of information that is useful to such an intelligent help system is knowledge of how the underlying system operates. This knowledge is contained in the code for the system. By exploiting system code as part of the help database, many problems of inconsistency between programs and their documentation can be avoided. In our initial investigations of this problem, we are representing the system code as a set of productions that are easier to manipulate than is code in most standard languages. As we develop techniques for answering questions by reasoning with knowledge about the system, we become increasingly able to answer the growing variety of questions that will occur as the language interface to a help system becomes more flexible.
INTRODUCTION

As complex software systems become more and more widely used by a larger and more heterogeneous group of users, it becomes increasingly important to provide, along with the systems themselves, good interactive help facilities. But as the size of a system grows, so too grows the size of the task of building the database to be used by its help facility. And building the database initially is only a small part of the problem. As the underlying system changes, the help database must also be maintained so that it always corresponds to the current version of the system. This is a large, boring, and all too often ignored task. A major goal of my research in building help facilities is to explore ways in which the task of building a database specifically for a help facility can be minimized by exploiting the underlying system itself as a major part of the required knowledge base. Using the system code as the help facility database guarantees that the two will always correspond.

A second reason for building a help facility based on the underlying system itself, rather than on a body of stored text that is fed to users on demand, is that a wider variety of user questions can be handled. If prestored text is used, all the questions to which the system will be able to respond must have been anticipated at the time the text was created so that an appropriate answer can be written. But if answers can be computed from the code of the system itself, this is not necessary. So, for example, questions such as “What is the difference between a and b?” can be answered by looking at what happens for a, looking at what happens for b, and comparing the results. An answer can then be generated that is based on that comparison. This flexibility becomes increasingly important as we move toward natural-language help systems in which the superficial flexibility of the system is high, leading users to expect comparable flexibility in the actual power of the system.

To explore the issues raised in the design of this sort of help facility, I have begun building a help system for the document formatter Scribe. Scribe is a sufficiently complex program that most users never learn all of it. Thus a help system for Scribe will need to be able to handle a wide range of questions from a broad class of users, ranging from novices to experts. As discussed in the following sections, the approach we are taking to answering user questions by referring to the system code makes it possible to tailor the responses generated to the individual needs of all these users.

ANSWERING USER QUESTIONS FROM SYSTEM CODE

Although it is true that not all the kinds of questions that people will want to be able to ask a help facility can be answered by examining the system code, a great many of them can be, particularly if the code is well structured. There are three basic categories of questions whose answers can be derived from the code for a system:

1. The user gives a description of a result and wants to know what causes that result to appear. A few examples of this kind of question are as follows: “Why is my paper coming out single-spaced?” and “How can I get the page numbers printed at the bottom of the page?” As these examples show, sometimes result-description questions occur because an undesirable (or perhaps surprising) result has occurred and users are curious about the reason. Other times these questions describe a desired result, and users want to know what they can do to get it. In either case, the way to answer the question is to find the place in the code where the described result is generated. Then look to see what conditions must be satisfied for that particular code to be executed.

2. The user gives a description of a set of circumstances and asks what would happen if they occurred. A few examples of this kind of question are as follows: “What will happen if I change the reference format to alphabetic?” and “How does Scribe float figures?” As these examples show, circumstance-description questions occur both when users are curious about what will happen if they do some new thing and when they want to find out exactly how Scribe performs a function in which they are interested. The way to answer this type of question is to find the place in the code that corresponds to all the specified conditions being met. Then look to see what action is performed. Depending on the level of detail appropriate for the answer, either the action can be described as a single action, or the lower-level procedures that compose it can be described.

3. The user gives two descriptions and asks for the difference between them. An example of this sort of question is, “What is the difference between the itemize and the enumerate environments?” This kind of question is answered by searching the code to find out what happens in each of the two circumstances given. Those answers are then compared, and the dissimilar parts are reported as the answer.

By using these three mechanisms, a variety of user questions can be answered easily from the system code, provided that the structure of the code is uniform and corresponds well to the structure of the operations being performed.

REPRESENTING A PROGRAM AS A SET OF PRODUCTION RULES

There are two ways that one could build a complex system and the help facility for that system so that both use the same
One is to write the code for the system in the conventional way and then to write a help facility question answerer that can exploit that code. Scribe is written in Bliss, so for this experiment this approach would mean building a question answerer that reasons about Bliss code. The second approach is to develop a new notation for writing the system code and then to build both an interpreter for that system and a question answerer that manipulates it. I have chosen the latter path, for several reasons, including the following:

1. Bliss allows unconstrained use of global variables and side effects. This means that merely on the basis of a static examination of a particular code fragment it is not possible to guarantee much about the behavior of that fragment.
2. Bliss is not a typed language. This means that it is not possible to tell simply from looking at a piece of code what kind of object is being operated on. Since a question answerer will need to be able to give responses that describe what pieces of code are doing, it is important that it have access to information about the types of the objects being manipulated.

In addition to the restrictions that need to be put on the language in which code is written if that code is to serve as the basis for a question answerer, there are other limitations that need to be put on the way that code is written.

Often code is written with deeply nested conditional statements. To determine the exact set of circumstances under which a particular fragment of code will be executed, it is necessary to search up several levels and collect all the conditions that lead down the relevant path to the code. This will be a common operation for the question answerer; so to make its job easier, code should be written with conditions flattened so that all necessary conditions immediately precede the code they guard.

The question answerer must be able to answer questions at a variety of levels of detail. Broad questions should not be answered at the lowest level of detail. To make this possible, the code should reflect a top-down decomposition of the behavior of the system. This will make it possible for the question answerer to select the appropriate level of decomposition to answer each individual question.

The question answerer will generate answers that describe the operation of units of the program. Since people can only comprehend fairly small units at a time, it is important that the code be highly modular, each module corresponding to a comprehensible set of operations.

The only way that the question answerer will know that some intermediate results have a meaning that can be discussed, whereas others do not, is for the important results to be marked in the code. This suggests that function composition should be limited to a few levels and that the results of these compositions should be assigned names that correspond to their function in the task domain.

To make it easy to write code that lends itself well to use by a help facility question answerer, I have designed a production system language in which rules describing a system's performance under a variety of conditions can be written. The rules correspond to a top-down decomposition of the system. Each rule consists of a left-hand side describing the conditions that must be satisfied for the rule to fire and a right-hand side describing the actions that are performed if the rule fires. For example, one of the top-level rules describing the behavior of Scribe is

(PLACE#CHAR C : CHARACTER)
(EQUAL C END#OF#LINE) •
(DISCARD C)

This rule says that if you want to process a file x, then open it, process each character of it, and close it.

This rule is simple. Its left-hand side consists only of the action that needs to be performed. Other rules have more complex left-hand sides reflecting the fact that the way an action is performed may depend on a variety of factors. As an example, consider the following rule for placing an individual character in the output file:

(PLACE#CHAR C : CHARACTER)
(EQUAL C END#OF#LINE)
(EQUAL FILEMODE FILL) •
(DISCARD C)
(PLACE#CHAR BLANK)

This rule says that if the system is trying to place a character, the character is the end-of-line character, and the text is being formatted in fill mode (in which each output line is completely filled and input line breaks are ignored), then ignore the end-of-line character but send a blank to the output file.

To experiment with code-based techniques of question answering, I am recoding Scribe as a set of rules such as these. Meanwhile, Scribe still exists and runs as a Bliss program. But this same production rule language could also be used to define a new system so that the rules would also serve as the code for the system. All that would be required would be an interpreter (and probably also a compiler) for the rule-based language.

THE OVERALL STRUCTURE OF THE HELP SYSTEM

In order to turn the rule-based question answerer that has just been described into a useful and complete interactive help facility, a set of other components is necessary. Figure 1 shows what these components are and how they communicate with each other.

The numbers in the figure indicate the information that passes between the components:

1. A user question stated in English.
2. A parsed form of the question.
3. An interpretation of the question in light of known patterns of dialogue structure (as described, for example, by Grice). For example, the question “Can I get Scribe to make an index for me?” would be interpreted as “How can I get Scribe to make an index?”

4. The complete set of answers.

5. An appropriate subset of the answers, chosen to match the individual user’s current interests and state of knowledge.

6. A response that makes correct references in the context of the current dialogue.

7. A properly worded English response.

The capabilities of each of the components of this system make possible a greater exploitation of the abilities of the others. For example, without the ability to compute the answers to questions by using knowledge about how the system behaves rather than by simply retrieving pieces of stored text, knowledge about an individual user and how much detail he/she is interested in is often wasted.

REFERENCES

