SUPPORTING IMPLEMENTATION OF SEMANTIC-LEVEL USER INTERACTION PARADIGMS

Peter Aberg and *Robert Neches*

University of Southern California, Information Sciences Institute
4676 Admiralty Way, Marina del Rey, CA 90292
aberg@isi.edu; neches@isi.edu

ABSTRACT

Many computer applications present their users with large information spaces that are difficult to understand and navigate. One class of solutions to this problem relies on allowing users to easily explore the information space, guided by continuous feedback provided by the system. Unfortunately, instantiating such a paradigm for a new application often requires a great deal of effort on the part of the developer. We are currently working on a shell environment that merges a model-based user interface development system with a proven interaction paradigm (a generalization of *retrieval by reformulation*) to assist developers in this task.

INTRODUCTION

A large body of recent work on human-computer interaction (see for instance [2, 5]) focuses on ways of making complex information systems, such as very large databases, more accessible to inexperienced users. This research has shown that users of such systems are faced with two major problems. The first is being able to mentally grasp the scope and structure of the information spaces presented by the systems. The second is formulating commands to navigate through such information spaces to locate items of interest. These two problems are aggravated by the difficulty for users of understanding the terminology used by the systems, which unavoidably differs from their own.

Both these problems can be reduced by building user interfaces that allow users to conduct explorative navigation through the information space. One such technique lets users iteratively refine a *description* of the items sought after, permitting them to elaborate and experiment with the problem at hand to determine the precise information needed. The viability of this paradigm of user interaction, called *specification by reformulation* (or, in its more limited form, *retrieval by reformulation*), has been demonstrated in systems such as RABBIT [4], HELGON [1], and three versions of our own BACKBORD [5] system.

Specification by reformulation is based on the technique of *solution by successive approximations*. Users find items they are searching for by successively refining a *partial description* of the items, guided by feedback provided in the form of *example items* matching the description as it appears at each iteration. These examples also help users overcome mismatches between the system's terminology and their own, as well as play an inspirational role by showing what it is possible to talk about.

Unfortunately, instantiating the specification by reformulation paradigm for a new application requires a great deal of coding. Despite the fact that the systems mentioned above share a common user interaction paradigm, they share no code. To promote the use of the specification by reformulation paradigm, we decided to investigate the possibility of using a model-based User Interface Development System (UIDS) for our application development. By defining a number of building blocks that implement the core elements of the paradigm using the UIDS, we can create a shell-like environment for developers to work with, thereby reducing the amount of work required to put together a specification by reformulation-based user interface for an application.

IMPLEMENTING SEMANTIC-LEVEL PARADIGMS

Model-based UIDSs, such as the HUMANOID system [3], provide application developers with an environment that allows them to work at a higher level of abstraction than if they were using widget toolkits or most other conventional UIDSs. Rather than forcing developers to begin design of a new application by putting effort into determining screen appearances and writing tedious event-callback code, HUMANOID permits design to begin by defining behaviors and presentations in a terminology much closer to that of the application. Instead of talking about *buttons* and *scrolling lists*, HUMANOID operates at the level of *application commands* and *command input alternatives*. Once developers have laid down the basic functionality of the application, they may proceed to tweak presentation details and lower-level command functionality if so desired.

Model-based UIDSs also allow an application's presentations and behaviors to be determined at run-time, according to context-sensitive criteria such as the type of data being presented or the current task.

The specification by reformulation paradigm is especially suitable for implementation using a model-based UIDS

such as HUMANOID. As opposed to many other, lower-level interaction paradigms, specification by reformulation does not focus on issues such as screen layouts and the appearances of widgets such as command buttons. Instead, the paradigm operates at a higher, semantic level corresponding to dialogs and dialog acts.

We have already developed a new implementation of our specification by reformulation-based BACKBORD system using HUMANOID. This BACKBORD version has been incorporated into a large-scale knowledge base development environment, as well as a logistics management system, with positive results.

A SHELL ENVIRONMENT

Our goal in providing application developers with a specification by reformulation-based shell environment is to raise the level of abstraction they work with by yet another notch, to that of *dialog components*. To do this, we use HUMANOID to create customizable building blocks from which user interfaces for application programs can be assembled. Each building block corresponds to one component of the interaction paradigm.

We define three building blocks. The first is a *description* module, the main focus of interaction for users, where they create and modify the partial description used to generate feedback. This includes a *generator* component that produces feedback based on the description, and a *filtering* component used to filter feedback before it is passed on to the example list. The second is the *example list* module, which displays a list (in some form) of items matching the description. The third is an *example viewer* module, used to display individual examples selected by the user from the example list in order to provide inspiration for possible changes to the description. This viewer enables users to copy components of items directly from the example to the partial description. It can also function as the link to the application, allowing users to apply application-specific commands to its contents (such as editing them, sending them off as an e-mail message, etc.)

Putting together a user interface for a new application is a matter of configuring each module described above. This procedure involves mapping the paradigm's components onto the terminology of the application. There are six basic steps to performing this mapping.

- *Establish the topic of the partial description.* What is it users are trying to find or accomplish?
- *Determine what operators are relevant to refining the description.* The paradigm defines a few generic operators for modifying descriptions, such as adding or deleting parts to it. Developers need to determine how, and if, these can be instantiated, and then decide what application domain specific operators are needed as well.
- *Decide what kinds of feedback can be generated based on the partial description.* A list of items matching the description is a minimum requirement. Other forms of feedback are possible as well, for instance a list of syntactically correct completions of the description.
- *Determine how to convert the description to a form suitable for generating feedback.* For instance, this might mean constructing a database query from the description.
- *Decide how feedback should be presented.* Is there a need for filtering the feedback, or for converting it into another form? Numerical data may, for instance, be converted into graphs.
- *Determine in what form user guidance can be provided from the feedback.* Allowing users to look at, and work with, example items returned as feedback is one form of guidance.

Each of these steps involves writing small functions (in LISP) to hook into the shell's modules. These functions should be no more than a few lines of code in most cases, since much of the functionality required by the modules probably already is used in other parts of the application itself and therefore already exists.

HUMANOID provides default presentations and interaction methods for each component. These can be specialized by the developer if necessary. Each component can also have associated help functions to assist users of the completed application. These can also be specialized according to need.

CONCLUSIONS

We hope to show that by implementing a shell, rather than instantiating the paradigm from scratch for each new application that comes along, we will promote software reuse, consistency of the user interface, and reduction of the development burden for new applications.

We also believe that coupling the use of model-based UIDSs with a high-level interaction paradigm provides more support to both users and developers than using a high-level paradigm alone.

REFERENCES

1. Fischer, G., Nieper-Lemke, H. HELGON: Extending the Retrieval by Reformulation Paradigm. In *Proceedings of CHI '89*, ACM, 1989, pp. 357-362.
2. Fischer, G., Stevens, C. Information Access in Complex, Poorly Structured Information Spaces. In *Proceedings of CHI '91*, ACM, 1991, pp. 63-70.
3. Szekely, P., Luo, P., Neches, R. Facilitating the Exploration of Interface Design Alternatives: The HUMANOID Model of Interface Design. In *Proceedings of CHI '92*, ACM, 1992, pp. 507-515.
4. Tou, F. N., Williams, M. D., Fikes, R., Henderson, A., Malone, T. RABBIT: An Intelligent Database Assistant. In *Proceedings of AAAI-82*, 1982, pp. 314-318.
5. Yen, J., Neches, R., DeBellis, M., Szekely, P., Aberg, P. BACKBORD: An Implementation of Specification by Reformulation. In *Intelligent User Interfaces*, Sullivan, J. W., Tyler, S. W. (Eds.), ACM Press, 1991, pp. 421-444.