Automatic Generation of Multimedia Help in UIDE: The User Interface Design Environment

by

Piyawadee "Noi" Sukaviriya
Jeyakumar Muthukumarasamy
Anton Spaans
and
Hans J.J. de Graaff

GIT-GVU-93-36
September 1993

Graphics, Visualization & Usability Center

Georgia Institute of Technology
Atlanta GA 30332-0280

Automatic Generation of Textual, Audio, and Animated Help in UIDE: The User Interface Design Environment

Piyawadee "Noi" Sukaviriya
Jeyakumar Muthukumarasamy

Graphics, Visualization, and Usability Center
Georgia Institute of Technology
Atlanta, GA
E-mail: {noi, jk} @cc.gatech.edu

Anton Spaans
Hans J.J. de Graaff

Delft University of Technology
Delft, the Netherlands
E-mail: {winfasp,j.j.deGraaff}
@IS.TWI.TUDelft.NL

ABSTRACT

Research on automatic help generation fails to match the advance in user interface technology. With users and interfaces becoming increasingly sophisticated, generating help information must be presented with a close tie to the current work context. Help research also needs to utilize the media technology to become effective in conveying information to users. Our work on automatic generation of help from user interface specifications attempts to bridge the gaps, both between help and user interface making help truly sensitive to the interface context, and between the help media and the interface media making communication more direct and more effective. Our work previously reported emphasized a shared knowledge representation for both user interface and help, and an architecture for automatic generation of context-sensitive animated help in Smalltalk-80. This paper presents a new integrated architecture in C++ which not only generates animation, but also audio as procedural help. The architecture also uses the knowledge representation to automatically provide textual help of why an object in an interface is disabled.

KEYWORDS: Automatic Help Generation, Animated Help, Multimedia Help, User Interface Representations

1. INTRODUCTION

Automatic generation of help unfortunately has not been a major issue in user interface software research. While research in user interface technology has advanced rapidly, few researchers in this area have paid attention to advancing help technology. Nevertheless, help needs to catch up with the user interface technology. It needs to utilize the current media technology to be effective in conveying information to users. With users and interfaces becoming increasingly sophisticated, and eventually adaptive, traditional canned help may not suffice to help users with their specific problems at hand. The task of creating appropriate help for all possible contexts is, however, impractical.

There are many interdisciplinary research issues related to using multimedia to convey help. Human factor issues of when to use which media and how effectively do people learn from these media must be understood; far more research is still needed in this respect. Previous experiments on multimedia help such as [2,12] and a multimedia help experiment being concluded by our group begin to explore the effectiveness of multimedia on-line help. As for the user interface software research, we are interested in how a user interface environment can use high-level specifications acquired from the interface designer as a means to bootstrap the generation of help information. Our objective is also to preserve the consistency between the current design of an application interface and the help information delivered to end-users.

The help research presented in this paper is part of a broader research project called UIDE, the User Interface Design Environment [6,8,16]. The overall objective of UIDE is to empower user interface development environments with knowledge about application semantics. The knowledge is captured through a task-oriented, high-level specification. Once captured, the knowledge is used to partially automate the user interface design process and to provide automatic runtime support such as generation of help and user task event logging. This paper only presents HelpTalk, the help generation part of the project.

Text, audio, and animation are used as media to deliver automatically generated help. The representation of the user's current context is used to construct animation sequence and the animation is played on the user's current screen. Currently, the automatic help generation algorithm works well but the quality of the help generated is still rough and machine-like, especially the sound bites. We have not yet provided means for designers to author or add information to improve the quality of the help the machine generates.

Help systems such as one presented in this paper are a beginning step towards automatically generating on-line help which is useful and is truly context-sensitive. In our case, the design knowledge of procedural tasks in an application, and their semantic constraints captured as pre- and post-conditions, is synthesized by HelpTalk to produce on-line assistance. It is important to realize that, not all help responses needed by users could be automated from procedural knowledge. For example, abstract definitions of tasks or objects do not exist in any form in the design specification, hence help regarding abstract information cannot be automatically generated. Lastly, making sure the appropriate kind of help is provided to end-users at the right time is very important. We have not yet addressed this issue in depth in our research.

In the following sections, a brief overview of precursor help research in UIDE and other related work will be given. A small set of examples showing the kind of help HelpTalk can generate will be shown, followed by explanations of how the help information is generated.

2. BACKGROUND ON UIDE'S HELP RESEARCH

Our work in the past [3,15] pioneered the automatic generation of help from user interface specifications. One of our earlier prototypes [3] utilized pre- and post-conditions attached to interface widgets to produce help information. An interface mechanism was built using pre-conditions to determine when a widget should be visible and/or enabled. In this system, a widget was enabled only when its pre-conditions were satisfied. The help system based on this mechanism [15] used unsatisfied preconditions of a widget as a basis to explain why the widget was disabled. Also, the system performed backward reasoning on widget definitions to derive a series of commands which must be performed to make the widget of interest enabled (i.e. this button and that button must be clicked to make this button enabled). The result of the backward reasoning process actually yielded a series of widgets, since it was performed on widget definitions. The series of widgets was translated into procedurally-oriented (commands) response such as "To make Object A enabled, 1) button B must be pressed, 2) button C must be pressed."

Our previous approach has a number of limitations, one of which is the restricted assumption that there is only one way to interact with an object or a widget, i.e., a button is pressed, a menu item is selected. This approach will not scale up to application-specific objects upon which multiple interactions may apply, i.e. an object can be dragged, clicked, or double-clicked on. Furthermore, stringing objects together to derive at a procedural description of a task does not always guarantee a meaningful explanation of a high-level procedure, as a high-level procedure may require interacting with multiple objects to complete a task. Lastly, from a software engineering point of view, attaching semantic conditions directly to interface objects makes changing interfaces to an application cumbersome. Changes an interface means transferring semantic descriptions to new interface components. This is especially hard when there is a paradigm shift in the new interface. This drawback is similar to that in the current interface programming practice where semantic conditions are embedded in callback routines of widgets.

Though pre-dated by automatic generation of animated [10] and graphical and textual [4] procedural demonstrations, Cartoonist [15] was the first system which generated animated help within the user interface runtime context. Cartoonist was implemented in Smalltalk-80; its emphasis was automatic generation of animated help from procedural knowledge. Cartoonist distinguished application and interface representations and kept the help generation mechanism independent of application-specific procedures. Cartoonist consisted of a planner which was used to fill action context with appropriate parameters, and to derive a series of action which would satisfy the pre-conditions of an action on which help is requested. Cartoonist demonstrated how to perform an action by showing a mouse icon moving on the screen onto objects with which the user must interact with, gesturing what needed to be done with the object such as which button must be pressed, and simulating the action by generating events to the underlying user interface handler. Typing was also displayed by using a keyboard icon showing characters typed in while actual character codes are simulated to the underlying interface handler. An animated help scenario consisted of a series of mouse movements and keyboard animation to demonstrate how an action must be performed. Cartoonist only silently animated. The animation algorithm has been re-implemented in C++ for HelpTalk.

3. OTHER RELATED WORK

Previous work on automatic generation of help has been primarily in the context of semantic inference from program code [7,13]. These help systems relied on rule-based code which was at a higher level than syntax-oriented languages such as C or C++. Work reported in [20] coupled executable code generated from the C language with sophisticated semantic structures which could produce context-sensitive runtime help directly related to the runtime context. This approach is rather expensive requiring duplicate effort in addition to creating the application. Natural language help [19] used representations designed to capture discourse structures, and worked well when the language of discourse was in the same medium as the interaction style to achieve tasks. The limitation becomes clear when interactions involve direct manipulation objects and the discourse structure does not lend itself to the nature of the interaction styles. All systems cited in this paragraph only produced textual help.

When user interface specifications were used to generate help as reported in [14,15,18], more interface knowledge was used which allowed help to take advantage of graphics on the screen. Palangue and his colleagues [11] used Petri Nets with objects as their underlying interface specifications and were able to produce help similar to our precursor system reported in [3]. However, they did not take advantage of the interface context and only used text in their help responses.

A number of plan-based explanation systems have been reported [5,17]. Feiner and McKeown [5] emphasized the use of synthesized graphics and complimentary natural language in help systems for off-line applications. Thies [17] used animated demonstrations in the actual interface context, and favored a two-tiered knowledge base of application model and interaction knowledge similar to Cartoonist's approach. A common thread among all these systems is the need for application tasks and object definitions to dynamically create procedural help. Systems as reported in [19,20,17] favored the approach which called for specifications of interactive applications solely for on-line help purposes. Palangue's [11] and our work are a step further; we used the same representation to drive both the interface and help. While the semantic information of the applications is declaratively captured and the user interface programming task is brought up to a much higher-level, help can be automatically generated.

3. WHAT CAN HelpTalk GENERATE?

Currently, HelpTalk generates answers to only two types of questions: "**Why** is *this widget* disabled?" and "**How** can one invoke *this widget*?" Through the representation, *a widget* is associated with actions represented in the UIDE's knowledge base of the application interface. Responses to the first type of questions are currently presented as text strings, while responses to the second type of questions are presented as audio and animation.

As you will see from examples shown later in this section, these questions may not necessarily be typed in by the users. They could be embedded as part of an application interface. The way in which these questions are embedded depends solely on the designer's preference. To give readers an idea of the help invocation mechanism and the kind of help HelpTalk generates, the following screens are shown and explained as examples.

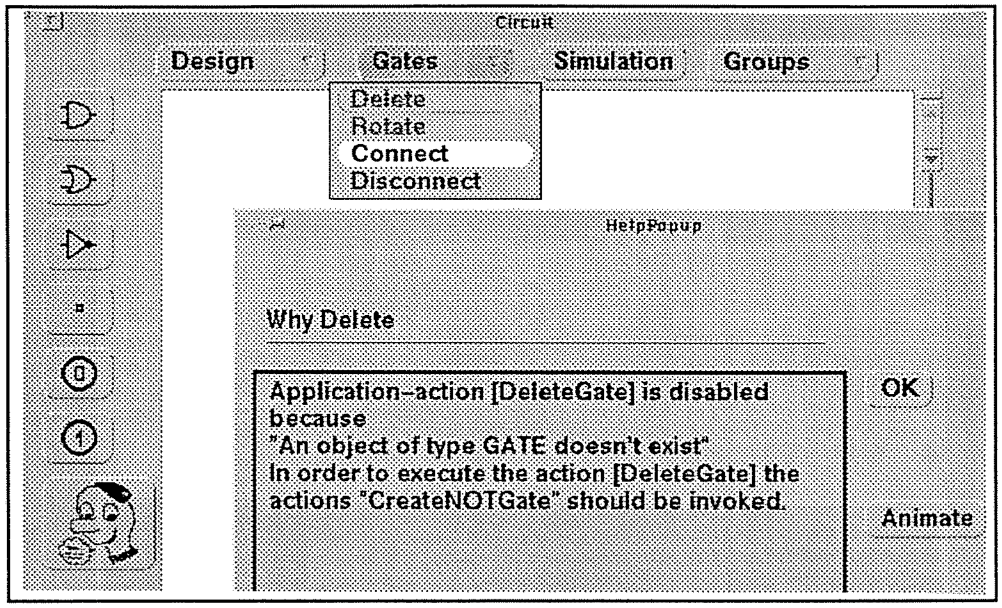
Textual WHY Help

Figure 1 shows an interface created in UIDE using the OPEN LOOK Intrinsic Toolkits (OLIT). In this figure, the menu items "Delete" and "Rotate" are grayed out. Located below is the help dialog box where help questions can be typed in. A question "why Delete" (equivalence of "why is the Delete widget disabled?") has been typed in. HelpTalk's response is displayed in the help dialog box. As you may notice, in addition to explaining why *Delete* is disabled, it also explains what needs to be done in order to make the *Delete* menu item enabled. In this example, HelpTalk looks at the pre-conditions of the action associated with the *Delete* menu item, the *DeleteGate* action. Its pre-conditions state that there must be a gate in the system for the *DeleteGate* action to be enabled. Since the user has not created any gate so far, the condition "exist (x,GATE)" is used as a basis to provide an explanation. As you may see, the explanation in Figure 1 states that "an object of type GATE doesn't exist." In addition to this explanation, it also states "To perform the *DeleteGate* action, the action *CreateNOTGate* should be invoked."

Readers may notice a couple of glitches in this help interface. First, unique labels are assumed for interface objects on the screen. Otherwise, HelpTalk will not be able to match the object from help questions to correct objects on the screen. Another glitch is, HelpTalk currently presents only one action out of multiple choices of actions which can satisfy conditions. In this example, the planner detects a number of actions which can produce a gate in the context (*createNOTGate*, *createNANDGate*, *createNORGate*, etc.). HelpTalk picks the first option and uses it to explain to the user ("To perform the *DeleteGate* action, the action *CreateNOTGate* should be invoked."). This problem can be solved by modeling create-gate as a generic create action, of which creating each type of gate is an alternative means to achieve the action. This requires a hierarchical representation of action structure currently not supported by UIDE.

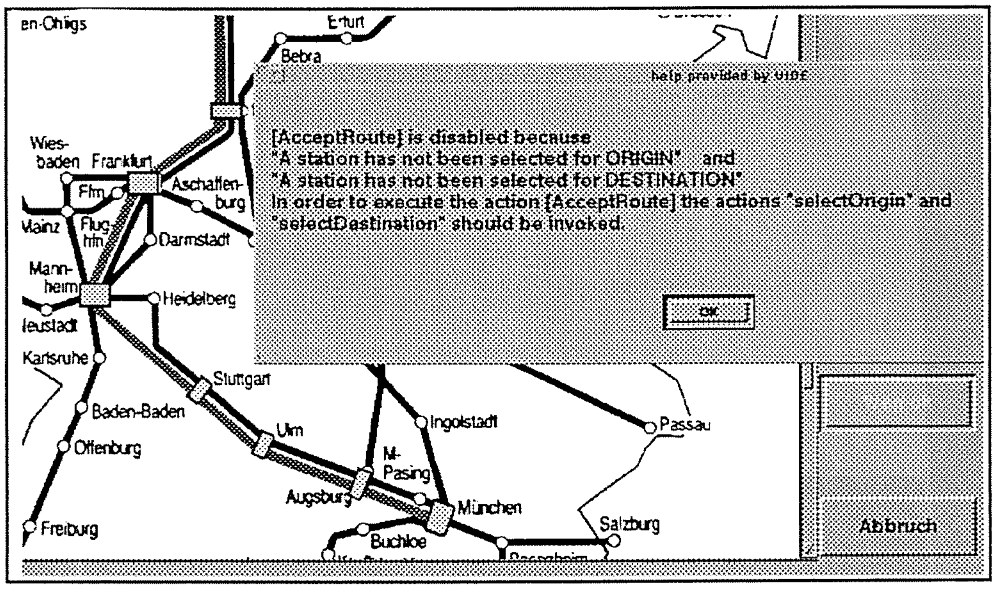
Figure 2 shows an interface to a computerized reservation system for the German InterCity Express train (ICE) for German audience. This interface is created in UIDE using SX/tools, a rather sophisticated graphical interface builder [9], as the interface front-end. At this point in the interface, the user has not chosen a city for an origin or a destination yet. Attempting to select the button labeled "Weg OK" (confirm the route selection) which is currently disabled causes the help dialog box to pop up. HelpTalk detects two unsatisfied pre-conditions for the corresponding action, *AcceptRoute*, which state that cities must be chosen for origin and destination of the route. From these two pre-conditions, the explanation "A station has not been selected for ORIGIN and A station has not been selected for DESTINATION" is generated. Since the model of the interface is in English and the explanation of predicates is given in English, help messages are presented in English though the labels of button are in German. We chose to be inconsistent to make it easier for readers to distinguish which parts are automatically generated from the representation and which parts are independent of the internal representation.

In the two examples above, the same help generation mechanism is used for two different help access mechanisms. Help is requested explicitly in the first example while the information is voluntary in the second example when the user attempts to select a disabled button. Here is a good point to emphasize the flexibility of separating the help access and help generation as two mechanisms. Though the two issues tie closely together and help systems will not succeed without good interfaces



Application-action [DeleteGate] is disabled because
"An object of type GATE doesn't exist"
In order to execute the action [DeleteGate] the actions "CreateNOTGate" should be invoked.

Figure 1. HelpTalk's explanation of why the action Delete is disabled in a digital circuit layout application.



[AcceptRoute] is disabled because
"A station has not been selected for ORIGIN" and
"A station has not been selected for DESTINATION"
In order to execute the action [AcceptRoute] the actions "selectOrigin" and "selectDestination" should be invoked.

Figure 2. HelpTalk's response to the user's attempt to select the disabled "Weg OK" button in an InterCity Express (ICE) train reservation system. In this application, user can repeatedly select cities of origin and destination until the route is satisfactory. The "Weg OK" button becomes enabled when both origin and destination cities have been selected.

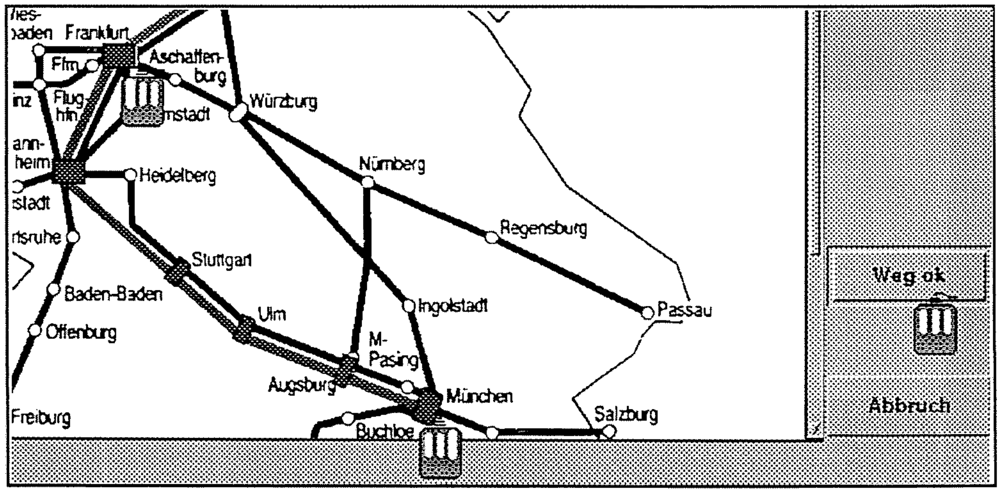


Figure 3. An illustration of animated sequences created by HelpTalk. The animated mouse first selects an originating city for a travel route with the left mouse button, then selects a destination city with the right mouse button. The animated mouse then proceeds to select the "Weg OK" button which is currently enabled in this figure.

on both sides, they can be thought of separately. Though the importance of the help access is always realized, our research work so far has concentrated on the generation side and not in depth on the interface side.

Audio/Animated HOW Help

In Figure 3, a composition of 3 different snapshots of an animated help scenario is shown. Prior to this snapshot, the screen looks like what is shown in Figure 2 where the "Weg OK" button is disabled. In this figure, the user presses the right mouse button on the disabled "Weg OK" button to invoke the animated help. Currently, we use clicking on the right mouse button as a signal to HelpTalk to animate what needs to be done to make the object enabled. Since the "Weg OK" button is disabled for the reason mentioned in the explanation of Figure 2, the planner is invoked to come up with a series of actions which must be done to make the "Weg OK" button enabled. In other words, the planner must look for actions which will create conditions satisfying the pre-conditions of the *AcceptRoute* action. Once the planner is done, HelpTalk animates the plan by selecting Frankfurt for the origin with the left mouse button, selecting Munich for the destination with the right mouse button, and selecting the "Weg OK" button with the left mouse button. Figure 3 shows the mouse after selecting Frankfurt, the mouse after selecting Munich, and the mouse before selecting the "Weg OK" button.

What cannot be shown in Figure 3 is the audio narration, which is played during the animation. The audio is generated by extracting information from the knowledge base which corresponds to different parts of the animation. In Figure 3, the audio states "The AcceptRoute cannot be done in this context. To perform the AcceptRoute action, you must perform SelectOrigin, SelectDestination first. To SelectOrigin, select this object using the left mouse button. To SelectDestination, select this object using the right mouse button. To perform AcceptRoute, select the Weg OK button using the left mouse button."

Readers may also notice that the messages generated have too strong of an implication that these two specific cities must be selected for the route. The message should indicate flexibility by stating that these two cities are chosen as examples of an originating and a destination cities. Currently, HelpTalk cannot distinguish between objects which must be chosen, for example only the "Weg OK" button can be selected to invoke *AcceptRoute* action, and objects which can be selected such as any city for an origin or a destination.

4. HELP KNOWLEDGE SOURCE

The help messages shown in the above 3 examples are possible because procedural knowledge is captured in UIDE's knowledge base. We often refer to this procedural knowledge also as "design decisions" which make up an interface. As mentioned before, UIDE uses this knowledge as a way to control the execution of an interface [16]. In this section, a brief description of the knowledge base is given. Due to space limitation and to keep the paper in focus, we will not elaborate in detail.

UIDE Knowledge Model

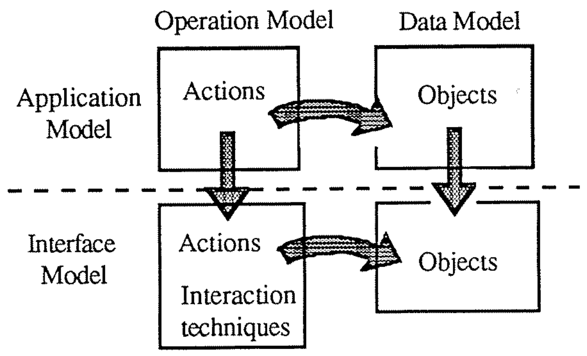


Figure 4 UIDE's Knowledge Base Diagram

UIDE separates its specification, or its knowledge, of an application as two separate models – the application and the interface. An application may have multiple interfaces designed for it, one of which is used in an instance of an application interface. The application model contains action and object descriptions specific to an application domain for which an interface is designed. The interface model contains interface actions and objects which are generic and can be used in various application domains. An interface model of a particular application consists of those interface actions and interface objects which are chosen for the interface of this application.

Actions and objects are related to each other through action-parameter relationships. For example, to "rotate a gate" requires a "gate" and an "angle of rotation" as parameters. Figure 4 graphically depicts UIDE's knowledge components and their relationships.

```
Action  Rotate
{
    Parameters:        (gate : GATE),
                       (angle : INTEGER)
    Pre-conditions:    exist(x, GATE)
    Post-conditions:   angle(gate, angle)
}
```

Figure 5 Example of an Action Representation

Since UIDE is a model-based user interface environment, a designer specifies an application by defining objects and listing actions which users can perform within an application. Figure 5 shows the representation of the "Rotate" action in a digital circuit layout program. Please notice that pre- and post-conditions are expressed in the predicate notation. Readers may refer to [16] for a full description of the representation.

The designer then specifies how actions in the application model maps to actions at the interface level. For example, the *Rotate* action can be invoked by 1) do *Select-Command-from-a-Pulldown-Menu*, which has a menu and a menu item as its parameters, 2) do *Select-Object*, which has a graphical object as its parameter, and 3) *Enter-Integer*, which has a dialog box, a text-entry widget, and an integer as parameters. When specific objects such as buttons, menus, and menu items are used in actions, the designer specifies object names in the specifications (not shown in this paper). HelpTalk uses this information of which specific objects are associated with which actions for a demonstration. HelpTalk acquires positions of related objects at runtime to create accurate animation scenarios. In case of non-specific objects such as the example shown in Figure 3 where any city can be selected, HelpTalk randomly selects an object of a correct type from the current context.

Actions at the interface level such as *Select-Object* are mapped to interaction techniques which specify input devices required for the interactions. For example, *Select-Object* can be performed by using the *Mouse-Click-Object* technique. Interaction techniques contain specific device and input event information about the techniques. For example, *Mouse-Click-Object* consists of pressing the mouse button on a graphical object, and releasing the button while the cursor position is still within the same object. Interaction techniques are part of the interface model.

UIDE is capable of handling multiple mappings between different levels. This accommodates representing multiple ways to perform an action, and multiple interaction techniques to perform an interface action.

5. ARCHITECTURE

Figure 6 illustrates the logical components of the UIDE's runtime architecture. At the heart of the architecture is the knowledge base which is created from parsing designer inputs of application actions and how they connect to interface tasks and objects; the former is stored in the application model, the latter in the interface model. At runtime, the blackboards associated with the application and the interface models hold declarative status of the application and its interface, respectively.

The User Interface Controller (UIC) uses the application knowledge and the interface specification in the knowledge base as its source to drive the dialog sequencing. When the user interacts with the screen interface, UIC determines which application action is invoked, processes the information, and sequences the dialog accordingly. More details of how UIC controls interfaces are described in [16].

HelpTalk has full access to UIDE's knowledge base including the blackboards. Much of the information to answer how-questions is constructed from the application and the interface models combined. It is the blackboards which allow HelpTalk's responses to be closely tied to the current context. The blackboard contents are facts which altogether represent the current context, allowing HelpTalk to complete context-sensitive help messages or scenarios. (The blackboard contents also allow UIC to determine whether an action is enabled.) For example, to rotate a gate, a gate must be used to demonstrate the procedure. The blackboard has references to all gates which have been created in the application .

Once HelpTalk is ready to show the user how to perform an action, it animates by first figuring out the animation scenario such as which objects to interact with and in which

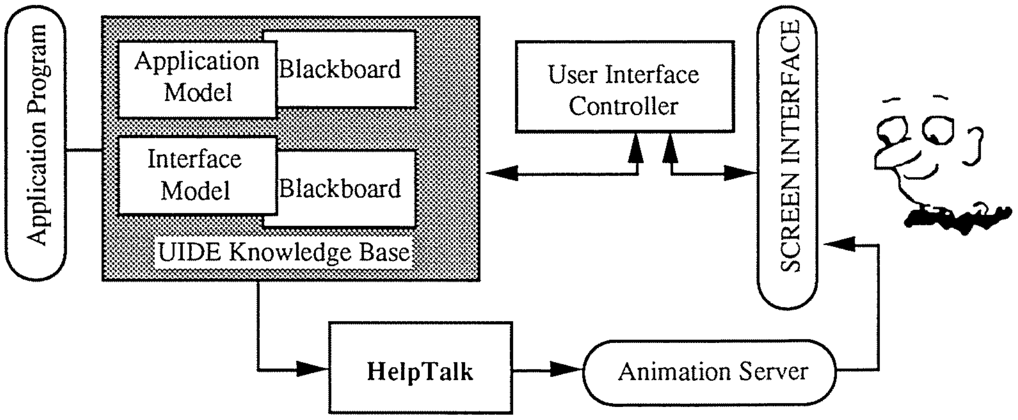


Figure 6 UIDE's Runtime Architecture with HelpTalk

nature should the interactions be. It then sends out low-level scripts to the Animation Server [1] which plays out the scripts. The Animation Server runs as a separate process. It responses to low-level commands in scripts such as "move the mouse to position (20,20)" or play-audio "Select this object using the left mouse button." The Animation Server draws the mouse on the screen, moves it around, and sends X events to the application interface which is controlled by UIC. It also sends audio information to its partner, the Audio Server. UIC responds to events generated by the Animation Server as if the user were interacting with the system; it then updates the interface context accordingly. Blackboards get updated and HelpTalk continues its procedural demonstration until the help request is completed.

6. GENERATION ALGORITHMS

Currently, HelpTalk uses two strategies for generating help responses. A response to a **why** question is generated based on unsatisfied pre-conditions associated with an action. A response to a **how** question is generated based on traversing the UIDE knowledge model to derive at procedural descriptions. A planner is used in both strategies to furnish the responses according to the actual context.

Textual WHY Help

For each predicate symbol used in action representations, UIDE requires the designer to give a sentence which will be used to explain when the predicate is true, and a sentence for when the predicate is false. HelpTalk depends on the latter to generate WHY explanations. The designer can specify in the sentence where runtime substitutions for predicate variables must be used. An example of a predicate sentence-specification is shown in Figure 7.

```
exist ({a}, {b})
True:  "Object {a} of type {b} exists."
False: "There is no object of type {b}."
```

Figure 7 A Predicate Description for WHY Explanations

The predicate *exist(a,GATE)* evaluates to true when the blackboard contains a statement such as *exist(myObject, GATE)*, which means there is an object called *myObject* in the system. In this case *a* is unified to {*myObject*}. When it is evaluated to false, HelpTalk would generate the sentence "There is no object of type GATE" for a WHY explanation, since *b* is unified with {*GATE*}. The sentence is then displayed in a dialog box as a text string. The text string could be sent to a speech synthesizer should we decide to present it using audio.

A WHY explanation is generated in two parts. The first part is the reason part as described above. The second part is the generation of what needs to be done. For the second part, HelpTalk invokes the planner to derive an action path which will satisfy the false pre-conditions. (Currently, the planner only searches in the application model.) HelpTalk then uses the actions in the derived path to generate the text which states that such and such actions must be performed before invoking the action for which help is requested. This process is rather straightforward. The actions derived by the planner are listed out as steps. It is possible to use animation as part of showing the steps. However, we feel that users may not want to see demonstrations at this point, and animation should be left as an option.

Audio/Animated HOW Help

Once the planner is invoked and a list of actions which must be performed is derived, procedural steps for completing each application action can be constructed by first traversing the mapping from the application model to the interface model. For example, rotating an object can be done by first selecting the *rotate* action, then *selecting an object* to be rotated, and then *entering an angle* of rotation. To animate this action, HelpTalk randomly chooses an object of type GATE from the current context, and picks an integer between 0 to 360 for the "angle" parameter. The selection is possible using UIDE's parameter type definitions.

For each interaction technique mapped to an interface action, there is a corresponding script which can be sent to the Animation Server. The script is parameterized such that variables, such as positions and which keys to type on the keyboard, can be substituted. HelpTalk resolves these substitutions based on objects and values it has chosen for each animation scenario. Normally, when an object is chosen for a selection, HelpTalk has to request its position and uses the position for substitution in a script. An integer value chosen or a character string usually can be used as is for substitution. HelpTalk then sends scripts with all parameters resolved to the Animation Server, one interaction technique at a time. When the Animation Server completes animating each interaction technique, UIC reacts to the generated events and modifies the context accordingly. HelpTalk then makes sure that following actions can be performed. If the following actions have pre-conditions unsatisfied in the now modified context, it will invoke the planner again to remedy the situation.

The audio source comes directly from action and object names in the application and the interface models. For example, when animating the rotate action, the audio is played as follow:

	To rotate	*...pause...*
(first)	you must select a **Rotate** item	
	in the **Gates** menu	*...animate...*
(then)	select an object	
	(to be rotated)	*...animate...*
(and then)	enter an integer	
	(for an angle of rotation).	

where **boldfaces** indicate substitutions at runtime, *italics* represents non-verbal actions, and (parentheses) represents what should be said to smoothen up the audio message. HelpTalk does not add those phrases in parentheses in its messages.

7. IMPLEMENTATION

Both UIDE and HelpTalk are implemented in C++ running on Sun SPARC stations. UIDE's knowledge representation is implemented as C++ classes. An application designer can choose to use OLIT widgets or SX/Tools as her interface front-end. Both toolkits use the same high-level application representation. HelpTalk only deals with high-level representation, therefore it is toolkit independent. At runtime, UIDE, HelpTalk, and the Animation Server run as separate processes.

8. CONCLUSIONS

Human factor issues must be realized for automatically generated help to be acceptable: generated text must be readable; audio and animation must synchronize properly; the generation process must not take too long, etc. Our generated text becomes too tedious to read, especially when multiple conditions are false. Narrations of animations must contain words which maintains continuity and precise references to the actual context. Refinements are needed in these respects. The response time of our help generation has been acceptable.

In this paper, we have demonstrated how some of help questions often asked by users can be answered automatically by a knowledgeable help system. Sharing knowledge representations with user interface control mechanisms guarantees consistency between help and actual interfaces. HelpTalk takes advantage of its tightly coupled architecture with the underlying interface controller, and access to runtime context, to construct truly context-sensitive help explanations. As mentioned earlier, these two types of help generated by HelpTalk by no means complete the genre of help which users need at runtime. However, we do not see this work as an end, but as an exciting beginning of automatic help generation research from user interface perspectives.

ACKNOWLEDGEMENTS

This work has been supported by the Siemens Corporate R&D System Ergonomics and Interaction group of Siemens Corporate Research Laboratory, Munich, Germany; the Human Interface Technology Group of SunSoft, Inc.; Digital Equipment Corporation; Schlumberger; and US West. The work builds on earlier UIDE research supported by the National Science Foundation grants IRI-88-131-79 and DMC-84-205-29, and by the Software Productivity Consortium. We thank the members of the Graphics, Visualization, and Usability Center for their contributions to various aspects of the UIDE project: Jim Foley, Martin Frank, Mark Gray, Krishna Bharat, and Hernan Astudillo. We also thank our former colleagues who contributed to UIDE: Thomas Kühme, Srdjan Kovacevic, Won Chul Kim, Daniel Gieskens, Dennis de Baar, Ray Johnson, Hikmet Senay, Christina Gibbs, Lucy Moran, and Kevin Murray.

REFERENCES

1. Bharat, K; and P. Sukaviriya. Animating User Interfaces Using Animation Servers. To appear in *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology.* November 1993.

2. Booher, H.R. Relative Comprehensibility of Pictorial and Printed Words in Proceduralized Instructions. *Human Factors* 17,3 (1975): 266-277.

3. de Graaff, J.J.; P. Sukaviriya; and C. van der Mast. Automatic Generation of Context-sensitive Textual Help. GVU-Technical-Report. Georgia Institute of Technology, Atlanta, GA. 1993.

4. Feiner, Steve. APEX: An Experiment in the Automated Creation of Pictorial Explanations. *IEEE Transactions on Computer Graphics and Applications* 5 (November 1985): 29-37.

5. Feiner S.K. and K.R. McKeown. Generating Coordinated Multimedia Explanations. *Proceedings of the 6th IEEE Conference on Artificial Intelligence Applications*, 290-303, 1990.

6. Foley, J.D.; C. Gibbs; W.C. Kim; and S. Kovacevic. A Knowledge-based User Interface Management System. In *Proceedings of Human Factors in Computing Systems, CHI'88.* May 1988, 67-72.

7. Genesereth, M.R. The Role of Plans in Intelligent Teaching Systems. In *Intelligent Tutoring Systems.* Eds. D. Sleeman and J.S. Brown, London: Academic Press, 1982.

8. Gieskens, D. and J.D. Foley. Controlling User Interface Objects Through Pre- and Post-conditions. In *Proceedings of Human Factors in Computing Systems, CHI'92.* May 1992, 189-194.

9. Kühme, T. and M. Schneider-Hufschmidt. SX/Tools - An Open Design Environment for Adaptable Multimedia User Interfaces. In *Proceedings of EuroGraphics'92, Computer Graphics Forum.* Vol. 11, No. 3. 1992. 93-105.

10. Neiman, D. Graphical Animation from Knowledge. In *Proceedings of AAAI'82.* 1982, 373-376.

11. Palanque, P.A.; R. Bastide; and L. Dourte. Contextual Help for Free with Formal Dialogue Design. In *Proceedings of the Fifth International Conference on*

Human-Computer Interaction. Vol. 19B. Orlando, Florida. August, 1993. 615-620.

12. Palmiter, S., and J. Elkerton. An Evaluation of Animated Demonstrations for Learning Computer-based Tasks. In *Proceedings of Human Factors in Computing Systems, CHI'91*. May 1991, 257-263.

13. Rich, E. Programs as Data for their Help Systems. In *AFIPS Proceedings of the National Computer Conference*. 1982, 481-485.

14. Spaans, A. Integration of Automatically Generated Context-sensitive Animated and Textual Help into UIDE. Master Thesis. Delft University of Technology. Delft, the Netherlands. 1993.

15. Sukaviriya, P., and J.D. Foley. Coupling a UI Framework with Automatic Generation of Context-Sensitive Animated Help. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*. October 1990. 152-166.

16. Sukaviriya, P; J.D. Foley; and T. Griffith. A Second Generation User Interface Design Environment: The Model and the Runtime Architecture. In *Proceedings of Human Factors in Computing Systems, INTERCHI'93*. 375-382.

17. Thies, M.A. Animated Help as a Sensible Extension of a Plan-Based Help System. In *Proceedings of the Fifth International Conference on Human-Computer Interaction*. Vol. 19B. Orlando, Florida. August, 1993. 712-717.

18. Tuck, R., and D. Olsen. Help by Guided Tasks: Utilizing UIMS Knowledge. In *Proceedings of Human Factors in Computing Systems, CHI'90*. April 1990, 71-78.

19. Wilensky, R., Y. Arens, and D. Chin. Talking to UNIX in English: An Overview of UC. *Communications ACM* 27. June, 1984. 574-593.

20. Young, D.; C. Smith; M. Washechek; R. Wolven; S. Haines; and W. Barge II. Dynamic Help: Automated Online Documentation. In *Proceedings of the 2nd Internatioanl Conference on Systems Integration*. June, 1992, 448-457.