

**Automatic Generation of
Context-sensitive
"Show & Tell" Help**

by

**Piyawadee "Noi" Sukaviriya
and J.J. "Hans" de Graaff**

**GIT-GVU-92-18
July 1992**

**Graphics, Visualization & Usability
Center**

**Georgia Institute of Technology
Atlanta GA 30332-0280**

Automatic Generation of Context-sensitive "Show & Tell" Help

Piyawadee "Noi" Sukaviriya
J.J. "Hans" de Graaff

Graphics, Visualization, and Usability Center
College of Computing
Georgia Institute of Technology
E-mail: noi@cc.gatech.edu, graaff@cc.gatech.edu

ABSTRACT

Textual help has become insufficient to guide users to carry out procedural tasks. Help for a direct manipulation interface requires that textual help be verbose in order to describe procedures accurately and specifically. In this paper, we attempt to alleviate this problem by presenting procedural help using coordinated textual and animated help. Our help presentation is dynamically created at runtime and is derived from the underlying user interface representations of the application model. Generating help from the user interface presentations allow help contents to be tailored specifically to the current context.

KEYWORDS: Animated Help, Automatic Help, Textual Help, User Interface Representations, UIDE

1. INTRODUCTION

Textual help has become insufficient in guiding users to carry out their procedural tasks in today's computer interfaces. Direct manipulation and multimedia technology require that textual help be verbose, e.g. to indirectly refer to objects on the screen and continuity of procedures and system feedback, in order to be effective and accurate. We propose that a help system should utilize a human expert's communication style of pointing and talking to explain procedures in an interface. Our work is a step towards realizing this kind of help. Our specific research goal is to develop a help architecture, including appropriate help representations, which can context-sensitize and deliver help information as coordinated telling and demonstrating. This help is part of the automatic interface support provided in a knowledge-based user interface environment [1].

Textual explanations are effective when used to explain abstract concepts. Animation is better when referring to objects on the screen or when demonstrating how to perform an action. By animated help, we mean simulating the mouse moving on the screen and characters being entered on the keyboard to demonstrate the interactions which must be performed to carry out actions. Animated help demonstrates application actions specifically to the user's current context. Textual help is used to accompany animated demonstrations to describe the more abstract concepts behind them. Examples of animated help are shown in Figures 3, 4 and 5.

Pointing and typing together has been used in previous research to make asking questions in graphical interfaces easier and more natural [2,3]. While the work in Moore & Swartout and Cohen et al focused on input to help, research work on automatic generation of coordinating text and static graphics as help output has been investigated in [4]. In Feiner & Mckeown's work, text is generated to complement graphical instructions. Our work focuses on coordinating independently generated text and animation from different levels of representational abstractions, and does not yet focus on adapting the content itself.

We are currently working on generating help for 3 types of questions:

- 1) How do I do X?
- 2) Why is widget Y disabled? and
- 3) How do you make widget Y enabled?

These 3 questions are often asked by a user in different situations. The first question is when the user knows of an action to be performed, but does not know how to carry it out in an interface. The latter two questions are asked when the user is in the middle of a certain task and encounters an interface item which she thinks should be used for the task. The second question, WHY, gives reasons why the item is currently unusable. The third question, HOW, is a follow-up question if the user decides that she needs to know the procedure which will make the item usable. Currently, we are focusing on sophisticated problem solving. We concentrate mostly on using information captured for a user interface environment to generate procedural help appropriate for various situations. The following section discusses the knowledge model which is used to capture this information and should provide a background to understand our help generation.

2. BACKGROUND ON USER INTERFACE REPRESENTATIONS

Our runtime environment is the User Interface Design Environment (UIDE) [5,6], where an interface to an application is designed, modified, and generated at runtime through high-level specifications of interface functionality. The knowledge base representation model of UIDE has been significantly refined within the past few years [7,8,9,10,11] to capture more semantics of an application and to provide a designer with a more refined control over an end-product interface. The heart of the knowledge base is a relational hierarchy of action descriptions, ranging from those

representing semantic actions meaningful to the application, those accommodating getting values and/or selection from the interface to the application, to those dealing with user interactions with the interface. Figure 1 portrays this relational hierarchy.

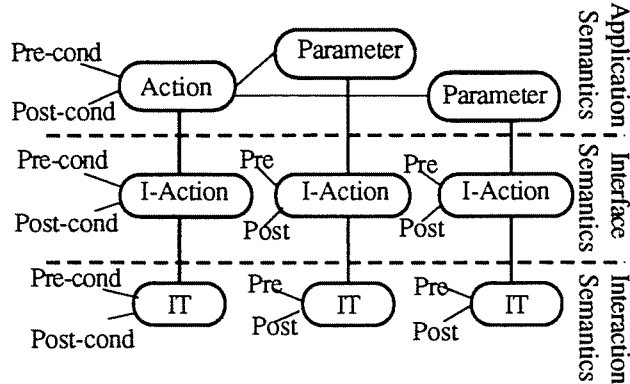


Figure 1 UIDE Knowledge Base's Relational hierarchy

At the highest level, *application actions* which the user can invoke are described in the knowledge base. These actions are described with the *parameters* which they require before they can be invoked. The user can perceive an action at this level regardless of its interface. For example, to rotate an object, she must know about having to tell the system of her intention to rotate, indicate an object, and specify how much rotation to apply to the object. In a Unix like interface, this information could be typed in as a command line. For a graphical interface, this could be accomplished perceptibly as a series of steps—selecting "rotate" from a pull-down menu, selecting an object, and entering a degree of rotation in a dialogue box. The representations at this level do not commit to any particular kind of interface.

Each application action has *pre-conditions*, which state what must be true in the application context before this action can be invoked. Using the same example, the pre-condition for the rotation action could be that at least one object must exist. Each action also has *post-conditions*, which state what will be true in the application context after the action is invoked successfully.

The designer, after describing application actions, has to commit to an interface. Mapping from application components—actions and their parameters—to an interface is specified in the knowledge base as links from these application components to *interface actions*. Various types of interface actions can be linked to application components; this is primarily based on what the designer sees fit as interfaces to enter different parameter values or to indicate selecting different types of actions. From the example above, the designer may choose a graphical interface for the rotate action. To do so, the rotate action is linked to the "select-action" interface action. Its parameters, objects and degree of rotation, are linked to the "select-object" and "enter-integer-in-dialogue-box" interface actions respectively.

Similar to application actions, each interface action has pre-conditions, which state what must be true in the interface context before it can be invoked. For example, the dialogue box, which contains the numeric input widget for entering a degree of rotation, must be visible before a number can be entered. Pre-conditions for interface actions also include those which indicate the sequencing of these actions [12]. For instances, a degree of rotation cannot be entered unless an object has been selected, and an object cannot be selected unless the rotate action has been selected. Each interface action also has post-conditions, which state what will be true in the interface context after it is invoked successfully.

Interface actions must be linked to a yet lower level representations—interaction techniques. Interaction techniques specify how interface actions are to be carried out in an interface. For example, the select-object interface action can be done by using the mouse to click on an object. Mouse-click-object is the interaction technique chosen for this example. More than one interaction technique can be linked to an interface action to designate possible alternative interactions. Typing in an object name can be used to select an object, for instance. In this case, we link the type-in-object technique to the select-object interface action.

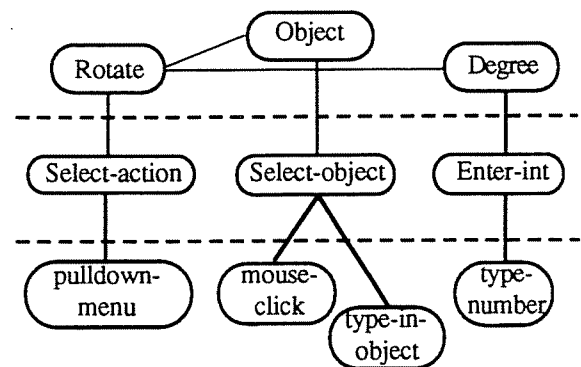


Figure 2 A Representation of Rotate Action

Interaction techniques also have pre- and post-conditions. Their pre- and post-conditions tie closely to the screen context, i.e. the numeric input widget must be enabled for a type-in-number technique, and will be disabled after a number is entered. Figure 2 shows the representation of the rotate action. To make the figure less busy, we left out pre- and post-condition details.

By using this relational hierarchy of information, our help system can infer procedural steps to complete each single application action. Using pre- and post-conditions, a sequence of actions for a task which requires multiple actions can be derived using AI planning techniques. Research on task representations is also under way.

The section below details how the representational hierarchy is used to facilitate the generation of coordinated textual and animated help. The automatic generation of animation and text are then described separately in Sections 4 and 5.

3. COORDINATING TEXTUAL AND ANIMATED HELP

For each type of question mentioned above, we coordinate text and animation differently. Since this is still an ongoing research, issues about the coordination styles will also be discussed.

How do I do X? Given that X is an application action, this question is answered by first explaining about the action and all parameters it requires. This level of explanation comes from the information captured in the highest level representations—the application level. Then we demonstrate how the action can be invoked and how each parameter can be given to the system. The explanation to accompany the demonstration is generated from the second level of representations stating interface actions for selecting the application action and for entering each parameter, one at a time. While the textual explanation for each interface action is displayed, the animation for that interface action is played. The animation utilizes information at the interaction technique level to elaborate on mouse moving or keyboard typing on the screen. Figure 3 shows a very simple mock-up example of help on an action *load-file*[†].

The coordination above, though is simplistic, gives a big picture of our strategy. More synthesis from the generic representations to the current context happens throughout the process to tie the animation specifically to the context. For example, certain objects may need to be selected to demonstrate giving parameters to an action. Rotating a car requires that an object of type "car" must be selected, for instance. Details of how animation is generated is detailed in Section 4. We maintain that some specific parameter values chosen for the animation do not have to be part of the textual description and the textual description should remain less context specific for this type of question.

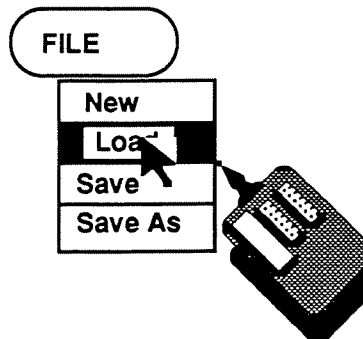
Why is widget Y disabled? The explanation of WHY a widget is disabled is textual and does not incorporate with animation. In UIDE, a widget is not enabled if the pre-conditions of the interaction technique associated with it are not satisfactory. And these pre-conditions may not be true because certain conditions have not occurred as the system expects. The explanation of why the widget is disabled is derived from the pre-conditions of its associated interaction technique which are not true. The actual text generation is described in Section 5. Using the example shown in Figure 3, if the OK button is not enabled unless a file name has been entered, the user can ask why the OK is disabled. Figure 4 shows help on why is the OK button disabled?

The action *load-file* is used to bring a file into your work space.

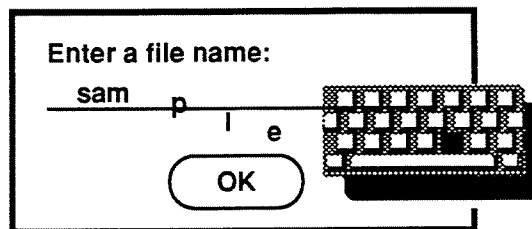
This action requires following parameter(s):
file-name

To perform the action *load-file*:

- 1) Select the *load-file* action



- 2) Enter a *file-name*



- 3) Select OK

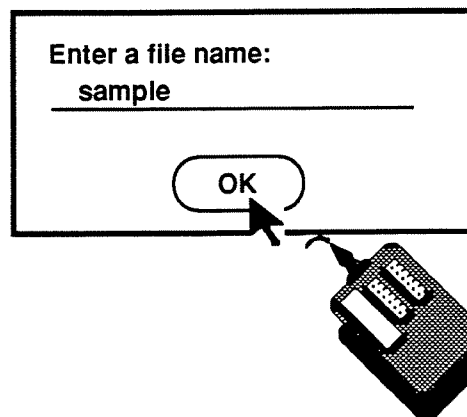


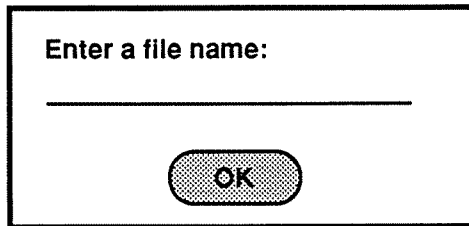
Figure 3 Help on How to Load a File. Each textual step is added to the previous text display before the animation for the step starts. Italicized texts are those which are replaced at runtime.

[†] We use a mock-up figure here as the prototypes for text generation and animated help have been implemented but have not yet been integrated.

The *OK button* is disabled because the *file-name* has not been entered

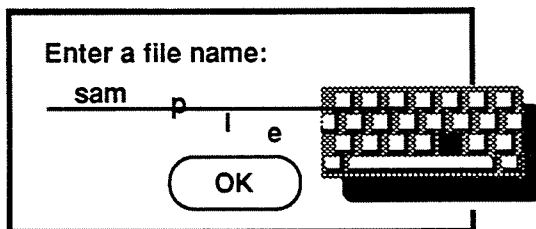
Figure 4. Help on WHY the OK button is disabled.

How do I make widget Y enabled? Following up on those pre-conditions which are not true in the previous question, a planner can be called upon to search for actions which have post-conditions matching the unsatisfied pre-conditions. The goal is to come up with all interactions which, if performed, will make the widget enabled. The planner searches hierarchically in the user interface representational space and makes sure that all related pre-conditions at all levels, starting from the interaction technique level, are satisfied. The actions found in the plan are then used to describe how the widget can be enabled. Again, text is used to describe application actions, action parameters and interface actions. The animation is incorporated to show how these actions can be performed. Figure 5 shows help on how do I make the OK button enabled?



(a) before

The *OK button* can be enabled by:
1) Enter a *file-name*.



(b) After

Figure 5 Help on How to Make the OK button enabled.

4. AUTOMATIC GENERATION OF ANIMATED HELP

In addition to the layer representations in the UIDE knowledge base, animated help depends on additional information related to parameters--*Parameter constraints* information stored with each parameter. Parameter constraints state the parameter type of the value, i.e., of the type "car" class or of an integer type. Parameter constraints also state more specific restrictions on the values such as a range of an integer value, the value in relation to other parameter values of the same action, compositional or positional constraint on objects such as the objects have to reside within the current design, etc. Animated

help use these constraints to guide its selection of appropriate values from the current context, i.e. selecting any object of type "car." Values unrelated to the screen context can be generated, i.e. an integer within the range 1-100 can be randomly selected. These objects or values are then used in the animation. Reader can refer to [8] for more details about constraint types, parameter value selection, and animated help algorithm.

5. AUTOMATIC GENERATION OF TEXTUAL HELP

In order to generate textual explanations, we use templates associated with different entities in the knowledge base to generate pieces of help text. Explaining why a widget is unusable is primarily translating the unsatisfied pre-conditions of its associated interaction technique into help text. These pre-conditions are represented by predicates, where each predicate has a name (predicate symbol), and a number of arguments. For example, `status(Cannon, Loaded)` is a predicate stating that the object "Cannon" must be in the "Loaded" status. A template is a piece of text associated with a predicate and it contains slots for the arguments. For example, for the predicate mentioned, its template could be "the <argument 1> is not <argument 2>" and the generated text would be "the Cannon is not Loaded."

There is one template associated with each predicate symbol. The help system contains a set of predefined templates for often used predicates. The designer, however, can extend this set of templates with application-specific templates.

Explaining why To answer a WHY question, we use the template "The widget is disabled because reasons" as a start. The "widget" slot is filled with a reference to the widget on which the user questions, e.g. "the button *load*." The "reasons" slot is filled with text generated from the predicates that currently are not true. These predicates, translated using associated predicate templates, are then strung together to form a complete text. For instance, if both `status(Cannon, Loaded)` and `status(Safety, Off)` are not true, the complete text would be "The button *fire* is disabled because the Cannon is not Loaded and the Safety is not Off."

Explaining how To answer the question "How do I make widget Y enabled?", the template "The widget can be enabled by: steps" is used. The "widget" slot is again replaced with the reference to the widget in question. The "steps" are generated from the action sequence generated by planner. Each action has a template which states what it does. For example, the select-action interface action has a template "select the action" where the action is replaced by the application action's name it links to. Therefore, if the user ask how to make the button *fire* enabled, the answer would be "The button *fire* can be enabled by the following steps: 1. Select the action *load*. 2. Select the action *safety*." If the user wishes to see the demonstration of these steps, each step will be presented, one at a time, in text first then accompanied by the corresponding animation.

If the user asks for how help on an application action, the template "Action description. This action requires the

following parameter(s): **parameters**. To perform the **action** **steps**" will be used. The "action" slot again is replaced by the name of the application action. The "description" slot is filled with the designer pre-defined description of the action. The "parameters" slot is filled with all parameter names of the action. The "steps" slot is filled with the interface actions which select the action and enter its parameter values. The sequence of interface actions to animate can be derived from the representational hierarchy of this action. When a step or an interface action is listed, animation accompanies each step.

6. CONCLUSIONS

We described 1) the framework of knowledge representations for capturing information for user interface purposes, and 2) how the knowledge of an application captured in this framework can be used to generate coordinated textual and animated help. We explained briefly how animated help is generated and at length on how textual help is generated.

This research work is a start of an automatic help generation from user interface knowledge. The strength of this knowledge is in its capability to capture procedural information in an interface. Procedural help is a fraction of a user needs to understand an application. It is plausible to integrate other kinds of help such as conceptual help, trouble shooting, and error messages into our framework. The challenge is to take advantage of our user interface representations to bring these pieces together such that only contextually appropriate help information will be presented to the user.

7. ACKNOWLEDGEMENTS

This work has been supported by the Human Interface Technologies Group of Sun Microsystems through their Collaborative Research Program, and the Siemens R&D System Ergonomics and Interaction group of Siemens Central Research Laboratory, Munich, Germany. We also thank the members of the Graphics, Visualization, and Usability Center for their contributions to various aspects of the project: Martin Frank, Mark Gray, Todd Griffith, and Srdjan Kovacevic.

REFERENCES

1. Foley, J.D.; D. Gieskens; W.C. Kim; S. Kovacevic; L. Moran; and P. Sukaviriya. A Second Generation Knowledge Base for the User Interface Design Environment. *Report GWU-IIST-91-13*. Washington, DC.: Dept. of EE & CS. The George Washington University, 1991.
2. Moore, J. and W. Swartout. Pointing: A Way toward Explanation Dialogue. *Proceedings of the Eighth National Conference on Artificial Intelligence*, 1990. 457-464.
3. Cohen, P.R.; M. Dalrymple; D.B. Moran, F. Pereira; J.W. Sullivan; R.A. Gargan; J.L. Schlossberg; and S.W. Tyler. Synergistic Use of Direct Manipulation and Natural Language. In *Proceedings of Human Factors in Computing Systems, CHI'89*. 1989, 227-233.
4. Feiner S.K. and K.R. McKeown. Generating Coordinated Multimedia Explanations. *Proceedings of the 6th IEEE Conference on Artificial Intelligence Applications*, 290-303, 1990.
5. 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.
6. Foley, J.D.; W.C. Kim; S. Kovacevic; and K. Murray. UIIDE—An Intelligent User Interface Design Environment. In *Architectures for Intelligent Interfaces: Elements and Prototypes*. Eds. J. Sullivan and S. Tyler, Reading, MA: Addison-Wesley, 1991.
7. 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.
8. Sukaviriya, P. *Automatic Generation of Context-sensitive Animated Help*. A Ds.C. Dissertation, George Washington University, 1991.
9. Kim, W.C., and J.D. Foley. DON: User Interface Presentation Design Assistant. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*. October 1990., 10-20.
10. Kovacevic, S. A User Interface Representation Model Based on a Compositional View. Paper submitted to 1993 International Workshop on Intelligent Interfaces.
11. Neches, R.; J.D. Foley; P. Szekeley; P. Sukaviriya; P. Luo; S. Kovacevic; and S. Hudson. Knowledgeable Development Environments Using Shared Design Models. Paper submitted to 1993 International Workshop on Intelligent Interfaces.
12. Kovacevic, S. A User Interface Representation Model Based on a Compositional View. Paper submitted to 1993 International Workshop on Intelligent Interfaces.