Built-in User Modeling
Support, Adaptive Interfaces,
and Adaptive Help in UIDE

by

Piyawadee "Noi" Sukaviriya
James D. Foley

GIT-GVU-92-25
October 1992

Graphics, Visualization & Usability Center

Georgia Institute of Technology
Atlanta GA 30332-0280
Built-in User Modelling Support, Adaptive Interfaces, and Adaptive Help in UIDE

Piyawadee "Noi" Sukaviriya
James D. Foley

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

Abstract

Developing an adaptive interface requires a user interface which can be adapted, a user model, and an adaptation strategy. Research on adaptive interfaces in the past lacks support from user interface tools which allow designers to easily create and modify an interface. Also, current user interface tools provide no support for user models which can collect task-oriented information about users.

In this paper, we present the User Interface Design Environment (UIDE) which provides an automatic support for collecting task-oriented information about users. UIDE uses its high-level specifications in its application model as a basic construct for a user model. By using this model, UIDE will be able to provide a number of adaptive features as interface design options: 1) adapting menu and dialogue box layouts; 2) suggesting macros to users; and 3) adaptive help.

1 Introduction

Developing an adaptive interface requires a user interface which can be adapted, a user model, and an adaptation strategy. In simple cases, the user explicitly defines his or her own model and adaptation strategy. For example, an experienced user may be aware of her expertise level and request full menus and brief prompts. A novice, on the contrary, may request short menus and lengthy prompts. In more interesting cases, the adaptive interface infers information about the user based on some aspects of how the user works with the application. A novice user who requests help on more advanced features and manages to use them should be asked whether she wants full menus and less elaborated prompts. A repetitive sequence of actions may be made into a single step by the system or the system perhaps can infer these actions automatically. Regardless of how sophisticated an adaptation is, its purpose is always to make computer interfaces faster to use and match the
skills acquired by users over a period of time. In order to achieve this goal, computers need to maintain information about each individual user.

A user model is a collection of information about a user of an application. The model may vary in its simplicity, from an individual screen configuration file such as that of Apple Macintosh™ desktop or OpenWindows’ .xinit file, to a more sophisticated novice/expert distinction (Rich, 1990), device and interaction technique preferences (Sanderson, 1991), and screen layout preferences. More sophisticated user models can be cognitively-oriented, representing for example how much the user knows about each application command, how the user currently conceptualizes the application, and how the user maps tasks onto command sequences. In general, the model can change over time as the user works with the application and learns (or forgets) various aspects of the application. Our interest is in modeling the user’s procedural knowledge of how to use the application to carry out tasks.

Adaptive interfaces do not just happen. An idea for an adaptation may or may not work in all applications. In fact, some approaches may sound good but may not be useful at all. Adaptations must be tested out with real users to see whether user productivity actually increases after an interface adapts. However, creating an interface takes time, especially if it is adaptable. For this reason, research on adaptive interfaces in the past suffered from a lack of supporting tools which allow an interface to be easily created and modified (Browne et al, 1990). Adding adaptivity to a user interface system would be very beneficial but has not yet been supported by any user interface systems or environments.

We have been working for some time with UIDE, the User Interface Design Environment (Foley et al, 1988; Foley et al, 1991). UIDE uses knowledge of an application to help automate the user interface design of an application, to control the execution of the interface designed, and to provide various forms of run-time help. The UIDE knowledge model is declarative and object-oriented, dealing with objects, attributes of objects, actions on objects, pre- and post-conditions of those actions, and interaction objects and interaction techniques through which the user interacts with the application. The model represents some of the application’s semantics and the interaction mechanisms used to access those semantics. We see an advantage in our representation model that it already represents relevant information needed in a user model which relates to user’s procedural knowledge. We propose that our representation model can be used as a basis to construct a user model.

The UIDE application model and its automatically generated user model will allow us to provide several forms of adaptive behavior. With respect to on-line help, we are able to
provide explanations of how to carry out various tasks. For example, given that there may be several ways to carry out a task, we can utilize information in the user model to choose the one which best maps onto the user’s current knowledge of the application. Similarly, the amount of explanation detail, and extent to which detail is repeated in an explanation, can be adapted. With respect to improving an interface, by combining the application model with a command trace, UIDE will be able to reorganize menus and dialogue boxes and suggest new macro commands to make an interface more “convenient” to use.

The UIDE adaptive interface project is in its first year. Our intention is to provide adaptive interface and adaptive help behavior as optional features to the user interface designer. When an application designer selects an adaptive feature, a user model will be automatically created. Data will be collected at runtime to evaluate when the adaptive behavior should be triggered. Performance data are recorded to be used later to determine whether the adaptation indeed improves the usage of the interface.

In this paper, we first provide a big picture of UIDE in Section 2, followed by discussions on related work in Section 3. In Section 4, we give detailed descriptions of the UIDE knowledge base model, its user interface functional components, and how these components are semantically linked. Section 5 discusses some of the components from the knowledge base which are used in UIDE’s user model and semantics of the information in the user model. Sections 6 and 7 provide summaries of user interface and help adaptive behaviors which can be supported in UIDE and which we are currently developing. Section 8 discusses our current status and future work and Section 9 concludes the paper.

2  UIDE: The User Interface Design Environment

UIDE is designed to support interface designers to easily create, modify, and generate an interface to an application through high-level specifications. Our purpose is to support the interface design process through its life cycle – from its inception to the actual execution stage. This support has been made possible using high-level specification, referred to as an application knowledge base, which describes various details of an application interface including related application semantics. In the past, our emphasis has been on the design support. Within the past few years, we have expanded our research to strengthen the quality of both design aides and runtime support which can be automatically provided from the same common knowledge base. Design aides include automatic layout of menus and dialogue boxes (Kim & Foley, 1990; de Baar & Foley, 1992), and design transformation aides using a compositional model of interface components (Kovacevic, 1992). Runtime
support includes controlling interface objects using pre- and post-conditions (Gieskens & Foley, 1992), automatic generation of animated procedural help (Sukaviriya, 1990; Sukaviriya, 1991), automatic generation of textual explanation on why a widget is disabled (de Graaff, 1992), and coordination of textual and animated help (Sukaviriya & de Graaff, 1992). Should the specifications in the knowledge base change, the generated interface and help at run-time will change accordingly.

Through UIDE, a designer designs an interface by putting in the knowledge base information about the application for which the interface is designed. This is done by creating entities in the knowledge base which capture application actions and objects. The designer then chooses various user interface functional components which best fit the application and link them to the application entities. User interface functional components include user interface actions, user interface objects (commonly known as widgets), presentation objects for representing application objects, and interaction techniques to be used with both application objects and interface widgets. In fact, various design tools have been developed to aid this process.

Kim (1990), in his DON system, analyzed the knowledge representations of an application and logically grouped various actions into menus and dialogue boxes. He also used graphics design principles to improve the visual design of dialogue box contents. Gray (1992) uses OPEN LOOK style guides (OPEN LOOK, 1990) to enforce the generation of dialogue box contents from the data model of an application. Kovacevic (1992) automatically forms the internal representations for UIDE to map application entities to interaction techniques, and allows the designer to easily navigate through alternative designs.

Once application representations are mapped to user interface functional components, either by using the tools mentioned above or manually and piecemeal if the designer wishes, the UIDE runtime environment depends on these specifications to generate a desired interface.

Though we use the OPEN LOOK Intrinsic Toolkit (Sun Microsystems, 1992) to create an interface in C++, UIDE does not generate code with a conventional program structure as one might expect. If that were the case, an application programmer would have to add application callbacks and still have to deal with sequencing of user interface dialogues. UIDE does not generate code to which application programmers must add to link with the application. The designer-given application representations and associated interface components are used for controlling user interactions and status of user interface widgets at
run time. UIIDE also provides automatic support such as help from the same representations. For example, textual and animated help on how to perform actions is constructed from the specifications upon user request (Sukaviriya and de Graaff, 1992). Current context is always used in the help construction process to make the help content closely relate to the current status of the user's work context. Help on why a widget is disabled at a particular time can also be generated when requested using pre-conditions information associated with actions and interaction techniques related to the disabled widget (Gieskens & Foley, 1992).

3 Related Work

Our work on the general UIIDE framework and now adaptive interfaces touches upon multiple research areas within the user interface field – high-level user interface representations, automatic generation of help, adaptive interfaces, and user models. To give a better picture of how these different areas are brought together, here we encapsulate some of the UIIDE design goals:

- Utilize high level user interface representations for automatically generating user interface components and handling interactions with users at run time;
- Automatically generate context-sensitive help from the user interface representations;
- Provide a basic construct for user models for which information is collected at runtime through the user interface framework and of which information is used to support adaptive interfaces;
- Predict the need for user interface and help adaptation based on the interpretation of user models, and adapt the interfaces by, for example, reorganizing menus and dialogue boxes in interfaces, suggesting a macro for a frequently performed task, varying help message content, etc.

The following sections summarize related work in the order of our design goals above.

3.1 High-Level User Interface Representations

UIIDE's knowledge representations center around describing commands and parameters of a user interface. These commands are called actions in our knowledge base and they are means for users to communicate with an application. Using descriptions of commands and parameters to generate interfaces was also the approach used in Cousin (Hayes et al, 1985), MIKE (Olsen, 1986), MICKEY (Olsen, 1989), UofA* UIMS (Singh & Green, 1989), and HUMANOID (Szkeley et al, 1992). Except for HUMANOID, these systems
parsed textual descriptions of commands and parameters, either embedded in a program structure or stored in a file, and used them as inputs to interface generators in their systems. UIDE and HUMANOID, however, created a knowledge base with entities corresponding to commands and parameters and used these entities actively at runtime. In UIDE, these entities also contained additional information which facilitated automated design aides (Kim, 1990) and manipulation of the interface being designed by the designer (Kovacevic, 1992).

Unlike these systems, UIDE's knowledge base captures more information about an interface beyond commands and parameters. It has declarative representations of application-independent components of an interface having to do with interaction techniques and manipulations of interface objects. By allowing an interface designer to manipulate the interaction level of representations, the designer has more control over the kind of interfaces to be generated for an application. Also, unlike other systems, UIDE captures action semantics through pre- and post-conditions of actions. Pre- and post-conditions are used to determine when an action can be invoked by the user, and the effects of an action to the user interface context. HUMANOID used guards, which are similar to our pre-conditions, to prevent an action from being invoked in an inappropriate context. However, HUMANOID did not represent or use post-conditions.

3.2 Automatic Generation of Help & Adaptive Help

There is a little body of research work on automatic generation of help from user interface representations. Tuck & Olsen (1990) used the interface descriptions (commands and parameters) in MICKEY, which were used to generate interfaces, to generate guided tasks. His guided tasks prompted users of what to do, step by step, to complete a task and the user could follow these steps. If the user wished not to follow these steps, Tuck's system used pre-defined parameter values to demonstrate the steps. His help system was tightly integrated with MICKEY, which is the user interface management system for the generated interface. The help system had access to menu item and dialogue box objects designed for each step. Taking advantage of this access, the system displayed help by pointing at these objects on the screen to context-sensitively show what needed to be done. Tuck's help was not designed to systematically handle all interface objects and interaction techniques. The system would require re-coding should interactions with interface objects change. Also, Tuck's help system did not provide help on objects other than menu and dialog boxes. These other objects are mostly application objects, which are created dynamically at runtime.

9/29/92
Neiman (1982) in his system GAK generated animated help for a CAD system from natural language representations. His animation was mostly canned and would have to be reimplemented if the interface changed. Feiner (1985) in APEX system generated illustrations to depict procedural tasks using description of actions, pre-conditions, post-conditions, and objects. His emphasis was on synthesizing the illustrations such that relevant parts stood out for each procedure. Cartoonist (Sukaviriyi & Foley, 1990; Sukaviriyi, 1991) generated context-sensitive animated help from UIDE representations. That is, when a user requested help on an action, Cartoonist showed an icon of a mouse moving on the screen with a mouse button pressed before pulling down a menu, selecting a menu item, and so forth. Cartoonist then proceeded to show all steps which were required to complete an action. Keyboard typing was shown with a keyboard icon and characters flowing to an appropriate dialogue box entry. Each animated help scenario was dynamically generated at runtime taking advantage of descriptions of actions, parameters, pre-conditions, interface actions, and detailed lexical descriptions of interaction techniques.

SINIX Help (Hecking, 1987) was also help about user interfaces; it provided help on UNIX-like operations in the SINIX operating system. SINIX's help knowledge base (Kemke, 1987) represented commands, parameters, and objects which these commands operated on (such as files, directories). However, these representations were created only for help purposes, and were not shared by the user interface mechanism in SINIX. UC (Chin, 1986) also used representations which were generated only for help purposes. Both SINIX help and UC were adaptive. Their help contents adjusted based on previous actions which users had performed. SINIX help suggested a shorter way to achieve the same goal. UC provided explanations of failures caused by previous actions.

Another interesting work on automatic help generation is by Feiner and McKeown (1991) in their COMET system. Unlike these previous systems mentioned which generated help for on-line applications, COMET generated help for operations which were to be performed in the real world. The system used representations of physical objects and their surroundings in the real world to help human operators proceed through their required tasks. Help presentations for a task resulted from an intelligent synthesis which highlighted objects in the scene related to the task at hand. The natural language generator then generated explanations which complemented the graphical depiction of the task. Though COMET only dealt with a rather static environment of fixed objects and scenes, its help presentation was dynamically created and adaptive to the tasks being depicted.
3.3 User Models

Research on user models have been around for quite some time and a large body of research exists as documented in (Wahlster & Kobsa, 1989). We too will support adaptive interfaces and adaptive help by modeling users and use information about each user to guide adaptations. Since user models vary from one system to another depending on what they are for, the application they concern, and how they are used, there are no two user models which are alike. In this section, we attempt to categorize our user model with some existing user model research.

Unlike some user models which revolve around users' mental models (Gentner & Stevens, 1983) or stereotypes of users (Rich, 1989), the UIDE user model is similar to those used in AI research – the user model which contains information collected directly from a person interacting with the system (Wahlster & Kobsa, 1989). One purpose of our user model is similar to that of UC, the Unix Consultant (Chin, 1989), which collected user information from user's past interactions to decide how much help information should be presented to the user.

The user model in UIDE is derived from a model that is used primarily to construct interfaces, be they adaptive or not. Our work is quite different from other research where user models are usually built totally separately from the underlying user interface support.

According to Sanderson's (1991) compiled list of user model features, UIDE's user model falls in the same category as that of his N-CHIME. N-CHIME's user model is explicitly represented for each individual user, dynamic over an interactive session, persistent throughout non-consecutive sessions, and its information is inferred from interacting with users. Interestingly, a number of elements in N-CHIME's user model such as number of times an action has been selected, user experience with various interaction techniques, etc., are similar to UIDE's user model. However, N-CHIME focuses on building a domain-independent adaptive interface, while our research emphasizes a comprehensive interface design environment where adaptive behavior is one aspect of the design.

3.4 Adaptive Interfaces

The kind of adaptive interfaces which we are developing in UIDE are labeled self-adaptive in Browne, Torredell, and Norman (1989). In Browne et al, they even differentiated self-adaptive systems further by levels of automation for adaptation provided by an interface.
Chapter XX of this book already provided a comprehensive survey and a taxonomy of adaptive interfaces. For this chapter, we will only mention research work which is more closely related to the types of adaptations we are pursuing. These types are: 1) re-organization of menus and dialogue boxes; 2) macro suggestion; and 3) adaptive help.

Re-organization of menu is often used to reduce menu access time. In Brown et al, they reported a self-regulating system which changed a menu tree for a telephone directory. The system constantly placed items the user accessed frequently higher in the menu tree. A self-monitoring system was implemented to monitor how the average depth of menu selection changed over time. This allowed the system to evaluate whether the adapted menu tree indeed helped users reduce menu access time. Another example cited in Browne et al (1989) was a system which reorganized a menu of word choices suggested by the system for spelling corrections. The words were organized based on the types of error most frequently made by users. Notice in these two applications that the adaptation in each application was based on one same task over time. The user selected data for the same operation – searching for a phone number or correcting a word. The user operated the systems knowing beforehand that they would have to search for their data choice in the menu to complete the repeated task.

Reorganizing a menu of commands (so a frequently used command is moved to the top of the menu, for example) is more task-oriented. Jovanovic (198x) reported a poorer performance on a system which adapted its menu based on the frequency of usage for each command. This resulted from the user having to relearn a motor distance for invoking a familiar command. We intend to explicitly suggest modifications to menu layouts to users and users can reject or accept the suggestions.

Recent work by Sears (1992) analyzed the frequency of widget-level action sequences to improve layouts of dialog boxes. His Layout Appropriateness metric took into account task descriptions, and also designer's criterion factors such as distance and size of targets, the number of eye fixations needed to extract the necessary information for a task, and other factors of choice into the analysis process. Through the analysis, Sears' system computed an optimal layout which the designer could take and implement. The analysis such as Sears' can be incorporated into the IDE environment. IDE can take an advantage of its task-oriented representations to collect user performance information at runtime and suggest a better dialog box layout to the user. Again, the user should be able to accept or reject IDE's suggestions.
Cote (1990) in his dissertation aided users in extending system functionality by helping them easily create macros. His system AIDA utilized object-drawing command representations to extract common parameters among commands programmed by the user for a macro, and used them as parameters for the macro. AIDA, on one hand, did not automatically suggest macro to users. On the other hand, a more intelligent system EAGER (Cypher, 1991) recorded the history of user's command invocations, detected a repetitive pattern when one occurred, and took the initiative to suggest a macro to the user. UIDE has action representations which can be used for recording history of user actions, hence repetitive patterns can be detected. The representations can be used to extract common parameters among these actions. Once a macro is created, an access to the macro must be automatically generated as part of the current interface.

As for related work in adaptive help, we have already mentioned some adaptive help systems (Feiner, 1985; Hecking, 1987; Chin, 1986) which are relevant to our work. A number of adaptive help research efforts clustered in the area of intelligent tutoring systems as documented rather extensively in (Sleeman & Brown, 1982). Many of the systems mentioned in Sleeman and Brown aided users in solving problems. We will narrow the scope down to only adaptive help which uses history of previous interactions as user models, in which case UC (Chin, 1986) and Sinix Help (Hecking, 1987) fall under this category. One other related system is by Senay (1987). Senay's help system recorded the statistics of command access by users of IBM VM/CMS operating system and used it to suggest new commands related to those already familiar with by the user. All these systems shared a common objective; they were designed to enrich user knowledge of the interface of the systems in which they were embedded.

In this section, we have mentioned a number of both past and ongoing research efforts which are related to adaptive UIDE. In the next section, we will discuss the knowledge representations of UIDE that are used to generate application interfaces. This section should give readers a background on the kind of information captured for interface purposes. The following section will then show how a subset of this information can be used in constructing a user model.

4 The Knowledge Base Model

Because UIDE is object-oriented, we have developed UIDE in Automatic Reasoning Tool (ART) (Inference Corp, 1987), Smalltalk-80, and currently C++. UIDE's knowledge base
model defines generic classes of various representation entities in the model. The knowledge base for an application consists of instances of these classes; each instance contains information specific to an application aspect it represents or a particular interface feature chosen for the application. In addition to these instances, the knowledge base is also formed by semantic connections among components.

We view the knowledge model as consisting of three layers of representations which describe interface functionality at different levels of abstraction. At the highest level, representations capture more abstract components and are designed to capture semantics of application actions. The mid-level representations capture actions which exist in various interface paradigms but are independent of any specific application. The lowest level representations describe mouse and keyboard interactions and are designed to capture lexical groupings which define related lexical input and output. Through these layers of representation, a designer has different granularity of control over interface features to be chosen for an application. Ideally, changing an interface style is a matter of manipulating connections between these layers.

Section 4.1 and 4.2 elaborate on various types of representations in these layers. In Section 4.3, we discuss pre-defined types of connections among these layers which bind application semantics to specific interface aspects.

4.1 Describing an Application

An application is described in the knowledge base as a set of actions which the user can perform within the application and objects on which these actions operate. All the functionality which is available to the users is described. The knowledge model defines action, parameter, parameter constraint, pre-condition, and post-condition classes to capture application actions, and object, attribute, and attribute type classes to capture application objects.

An action is a unit of activity which is meaningful to an application. Executing an action may affect objects, attribute values of objects, or object classes. An action may require parameters, hence the parameter class is defined to describe a parameter of an action. Constraints on parameter types and parameter values are defined as instances of the parameter constraint class. More detailed information about parameter constraint classes can be found in (Sukaviriya, 1991). The pre-condition class is defined to capture what needs to be true before an action can be executed and the post-condition class is defined to
capture what will be true after an action is executed. Both pre- and post-conditions in fact use first order predicate logic\(^4\), and their class definitions capture such a representation. An action which requires parameters has pointers to its parameter instances. Each parameter points to the constraint instances associated with it. An action also points to pre- and post-condition instances associated with it.

To give a better picture of what application descriptions involve, let's use a digital circuit layout as an application. In this application, end-users create different kinds of gates and connect them to form a functional digital circuit. This application will be used throughout this section. Examples of application actions in this application are:

- `create-gate` (gate-type, location)
- `delete-gate` (gate)
- `move-gate` (gate, new-location)
- `rotate-gate` (gate, angle-of-rotation)
- `connect` (gate-1, gate-2)

In addition to action descriptions, objects which are manipulated within an application must also be defined in the knowledge base. For example, objects in the digital circuit layout application are gates, circuit boards or designs. A GATE class would have subclasses of various gate types such as AND, NAND, OR, NOR, etc.

Figure 1 depicts a graphical view of an action. In general, an action is linked to its parameters, pre-conditions, and post-conditions. Each parameter is also linked to its constraints. Figure 2 gives an example of the "connect" action in the circuit layout application. The "connect" action has 2 pre-conditions which altogether state that two gates must exist in the context. Both "gate-1" and "gate-2" have constraints stating that they must be of type GATE, and "gate-1" and "gate-2" must not be the same gate.

---

\(^4\) We currently do not yet handle universal and existential qualifiers.
4.2 User Interface Description

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 actions. For example, the designer may choose a graphical interface for the "rotate" action from the action list above. To do so, the "rotate" action is linked to the "select-action" interface action. Its parameters, objects and angle of rotation, are linked to the "select-object" and "enter-integer-in-dialogue-box" interface actions respectively.

Similarly 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 an angle of rotation, must be visible before a number can be entered. In this case, \textit{visible(box)} is defined as the pre-condition where \textit{box} is the name (pointer) of the dialogue box which is the parent of the numeric input widget. Pre-conditions for interface actions also include those which indicate the sequencing of these actions (Kovacevic, 1992). For instance, an angle 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 yet must be linked to a lower level representation – interaction techniques. Interaction techniques specify how interface actions are to be carried out by the user. For example, the "select-object" interface action can be carried out 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-string" technique to the "select-object" action.

![Diagram](image)

**Figure 2** UIDE Knowledge Base's Relational hierarchy

Figure 2 depicts the representation layers in the UIDE knowledge model. The top level is what we sometimes refer to as the conceptual design of an application. A conceptual design ideally serves as a starting point in the design process and should be independent of what the end-user interface would be like. The same conceptual design can be mapped to multiple sets of interface action and techniques targeted for different environments.

![Diagram](image)

**Figure 3** A Representation of Rotate Action

Figure 3 shows the representation of the rotate action from the example above. Some details are elided. For example, we did not show that the "object" parameter of the "rotate" action has a constraint stating that it has to be of type GATE. The "angle-of-rotation" parameter has constraints stating that its value must be of an integer type and must be

9/29/92
within the range of 0 to 360. Details on the interface actions' parameters are also left out. The "select-action" has a parameter which is the "rotate" action itself. "Select-object" has an object as a parameter, and so forth. Interaction techniques have parameters as well; the "pulldown-menu" technique has a menu and a menu item as parameters. Pre- and post-conditions are also left out completely in this figure.

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. Interface objects such as dialogue boxes, menus, buttons, etc., must be created and named during the design process so they can be referenced in the knowledge base.

In summary, once an application is described in the knowledge base as actions, parameters, objects, and so forth, interface actions can be linked to actions and parameters to specify the interactions which will interface with each application function. As mentioned in Section 2, intelligent tools have been developed to automate the process of linking these components. This is done by using semantic information about actions and parameter types. In the same fashion, interaction techniques can be linked to interface actions to specify the actual interactions which will carry out these interface actions. More details on this automatic process can be found in (Kovacevic, 1992).

4.3 Controlling the Interface Through the Knowledge Base

Remember that each component in theUIDE knowledge model is an instance of a C++ class. These instances have multi-facet behaviors – one of the facets is to control user interactions at run time†. The UIDE runtime architecture is designed such that the UI representations are used to guide the dialogue sequencing. This means that UIDE does not hard-wire the dialogue sequence and bindings to interaction techniques in the runtime code. When the specification changes, the corresponding connections in the knowledge model change and hence the interface behaviors change accordingly.

At run time, UIDE always knows which application actions have all their pre-conditions satisfied thus are available to be invoked by the user. The interface actions and interaction techniques associated with these enabled actions are enabled. Interface objects and widgets

† The other two facets are 1) to maintain the knowledge model, and 2) to cooperate with the run time help system.
associated with these interaction techniques are also enabled, the results of which are reflected in the status of objects in the interface, i.e., a dialogue box is popped up, a corresponding text entry widget is enabled, etc. UIDE only accepts interaction techniques which are enabled at any point in time and sequence the dialogue accordingly.

UIDE uses several levels of specification to control an interface. After the user selects an action from a pulldown menu, for example, the runtime environment checks in the knowledge base to see which parameters the action requires, which interface actions are assigned to them, and which interaction techniques are expected next. UIDE then enables appropriate dialogue boxes, menu items, and other interaction objects. For instance, if a parameter must be entered through a dialog box, UIDE displays the dialogue box. User interactions with the interface are always recognized by UIDE as the action or part of an action that the user is performing. When the user successfully enters all the parameters required for an action, UIDE invokes the corresponding application routine for that action. UIDE is then ready for the user to start another action. Since the knowledge base is constantly consulted by UIDE, changes which occur in the knowledge base becomes effective in the generated interface.

4.4 Automatic Generation of How and Why Help

By using the relational hierarchy of information, procedural steps designed for completing each single application action can be inferred. For example, from the representations, rotating an object can be done by first selecting the "rotate" action, selecting an "object" to be rotated, and then entering the angle for rotation. An object of type GATE is chosen from the current context for the animated help to demonstrate selecting an object. This is derived from the constraint for the "object" parameter, which states that the object has to be of type GATE. An integer number between 0 to 360 is chosen for the "angle-of-rotation" parameter according to its constraints. The "rotate" procedure is animated by showing clicking the left mouse button while pulling down the menu where the "rotate" action is located, releasing the mouse button at the "rotate" menu item to select the action. The animation then proceeds with clicking on the chosen object, and typing the chosen number in the dialogue box which pops up after the object is chosen.

Pre- and post-conditions are used to evaluate whether a context is ready for animating an action. For example, to rotate an object, the pre-condition of the "rotate" action states that an object must exist. If there is no object of type GATE in the current context, a planner is invoked to search for an action in the knowledge base which will create a gate. Animated
help will include animating how to create a gate first, and then how to rotate a gate. Artificial intelligence planning techniques are used for action searches using pre- and post-conditions of actions. Currently, we are working on generating text to accompany animation. Research on task representations is also under way.

UIDE can also generate explanations of why a widget is disabled if the user inquires. A widget is disabled if the interaction techniques for it are disabled. The pre-conditions which are not satisfied are used as the basis for textual explanation generation, as discussed in (Sukaviriya & de Graaff, 1992). The explanations also include which actions must be performed to enable the widget. Currently, we are integrating animation with this type of explanation.

Our recent research efforts have considerably refined UIDE's knowledge model and its architecture, resulting in much finer control over features of a desired interface. Indeed, UIDE removes interface programming from the application design process. What is exciting in the UIDE research is that our knowledge model has made automatic support for user models, adaptive interfaces, and adaptive help realistic extensions of UIDE. In fact, adaptive behaviors that are plausible in the UIDE framework are extensions of our previous automatic generation research such as automatic organization of menus and dialogue box layouts and automatic generation of animated and textual help.

5 User Models as a subset of KB model

The benefit of our representation-based user interface architectural approach is twofold. One is the previously mentioned runtime generation of user interfaces and help. The other is to provide the basis for recording information about user interactions. User models can be created using a subset of entities in the knowledge base. Since the knowledge model consists of layers of information at different levels of abstraction, user data can be captured in these units of information at any layer.

Taking advantage of the fact that UIDE constantly traverses application representations to control the runtime environment, UIDE always knows which action is being interacted with, which parameter of which action is being entered, and which interaction techniques can be used and which is being used by the user. UIDE constantly visits different parts of the representations for various types of information. This makes it possible to select the already existing components of the UIDE knowledge model as a start for constructing a user model.
The UIDE knowledge components are instances of actions, parameters of actions, interface actions, and interaction techniques. One basic approach is to infer user knowledge about each instance based on how the user has employed the instance. For example, several successful invocations of an action implies that the user knows about the action, its parameters, and at least one set of interaction techniques for interacting with the action. Multiple successful uses of an interaction technique implies that, when providing help on an unfamiliar action which employs the same technique, elaboration on the technique is not required. Hence animated details may not be needed. The user should have no difficulties executing the technique, but only needs to know that the technique could be used with the new unfamiliar action. Currently, we are not at a stage where we know how many successful uses of different components actually mean that the user has understood these components. This is part of our adaptive interface research and can only be confirmed by testing. Our current research emphasis is constructing a framework wherein statistics can be collected.

Our approach is to use an overlay of the UIDE knowledge model as a user model to record information pertaining to user history of interactions. However, much of UIDE representations deals with links among semantically related components. These links are not necessary in the user model since they can be inferred from the UIDE knowledge base. Those links hence are dropped out in the user model. Each user has a copy of her own user model.

In most cases, there is an entity in the user model corresponding to an entity in the knowledge base. For example, each application action has a corresponding "shadow" object in the user model. Each of the action's parameters also has a "shadow" parameter object in the user model. However, not every component of the UIDE knowledge base needs to be duplicated in the user model structure. For example, information about the same interaction technique employed by two different actions should be the same "shadow" object for this class of technique.

Following are the categories of information which UIDE can collect for the user model using the current knowledge base structure.

5.1 Statistical History of Interactions

Upon a successful completion of each application action, its shadow in the user model has one count added to a slot marked "completed by the user." The shadows of all parameters
of the action also get one count for completion. After each action is completed, the date and
time are stored. All interface actions and interaction techniques which are linked to the
action and its parameters get one count as being completed by the user.

A more interesting recording would be: when the user requests help on an action right after
cancelling it or right in the middle of performing it, one count is added to "user having
some difficulties" with the action. If the action is cancelled and no help is being requested,
the action is marked "attempted by the user." Timestamps are stored with all of these
recordings. A number of successful completions of the same action following its usage
with difficulties should override the fact that the user had difficulties in the past. Again,
these numbers have yet to be determined by having real users using our system. It may
prove useful to consider the state of user interaction when an action is cancelled, for
example, action A is cancelled after the action is selected and its first parameter is entered.
Such a recording can be easily done in the UIDE framework, even if the parameter values
need to be recorded. Uses of interface actions and their parameters are also recorded in the
same fashion.

Two kinds of statistics related to interaction techniques are recorded. One is the
success/failure usage for each interaction technique, which is stored in one place though the
technique may be used for various application actions. This is based on the assumption
that mastering an interaction technique is a transferrable skill. The other kind of statistics is
about which technique is chosen to perform an action or to enter a parameter. In the case
where multiple interaction techniques can be used for an interface action, the count for that
interaction technique will be added for the interface action.

5.2 Chronological History of Interactions

Chronological history can be recorded rather straightforwardly in UIDE. Taking advantage
of UIDE's layers of representations, the history of user actions can be recorded at various
levels of detail. Upon completing an application action, its use can be recorded as well as
specific parameter values chosen. As part of recording each application action, interface
actions and chosen interaction techniques are also be recorded.

Recording the actual X events the user generates can be easily done even without the UIDE
framework. However, recording events takes much more space and their related semantic
actions are not captured in the data. By recording application actions and specific parameter
values the user chooses, the semantic information is captured. Recording information at
high-level of abstractions and storing specific parameter values make low-level event recording unnecessary. UIDE uses chronological recording to detect repeated patterns which can be suggested to users as macros.

5.3 History of Help Requests

It is often useful to remember the kind of help questions the user had in the past. The history of help requests is stored in our user model to predict how much help to present to the same user the next time the same help is requested. The history of help requests is recorded both statistically and chronologically with timestamps.

Since UIDE automatically generates help on actions, we will focus on recording help requests on actions. The shadow for each action will have slots to record help-related statistics. For example, an action gets one count on help when help on the action is requested. (Recording how long the user spends in each help session is possible though we have not included these data in our adaptation strategies.) The action is marked visited every time help on the action is requested. Recording help is not necessary for action parameters since UIDE describes and demonstrates parameters every time help on an action is requested.

Chronological recording of help requests is recorded separately. This recording is merely a list of actions in the order in which help is requested and timestamps when these requests occur.

6 UIDE Support for Adaptive Interfaces

We discuss in this section adapting the interface, and in the next section adaptive help. In both cases, the user model and information extracted from the chronological interaction history play a role.

Adaptive systems have sometimes adapted the user interface without any user control, leading to user confusion and lack of acceptance (Browne et al, 1989). In essence, the user has seen the interface changing without knowing exactly what changes were made (exploration being the only means of discovery), why the changes were made, and without consenting to the changes. We will give the user control over the adaptive behavior at three levels.
At the first level, the user will be able to completely disable user interface adaptation, either globally or with respect to different types of adaptation (our hope, of course, is that most users will not do so). This is analogous to turning prompts on or off in keyboard-oriented interfaces, or to choosing short versus long menus in contemporary interfaces.

At the second level, the user will be asked to confirm changes. Adaptation suggestions and their rationale will be presented to the user, who will be free to accept, reject, or defer the adaptation. That is, the user will be able to ignore the suggestion and work on: a project deadline might be 10 minutes away, in which case there won't be time to consider even the most worthy suggestion. Also, suggestions will be able to be deferred to a later time.

A third level of user control is the frequency with which adaptation suggestions are made. Because an adaptation requires relearning, and at least a temporary loss of productivity, reorganizations which have a low benefit in terms of productivity increases and a high cost in terms of relearning should perhaps never be made. Adaptations with high benefit and low relearning cost should be made quickly, while those "in between" are more problematical. We will allow the user to be involved in determining thresholds used in deciding when and if adaptations should be suggested.

Our initial focus is on two types of interface adaptation: reorganization of menus and dialogue boxes, and addition of new commands (macros). A simple power-law learning model (Card et al, 1983) will be used to predict how rapidly the user will return to the pre-reorganization level of performance, and a keystroke model (Card et al, 1983) is used to predict performance using the reorganized interface.

The power law learning model is of the form:

\[ T_n = T_1 n^{-a} \]

where \( T_1 \) is the time taken to perform a task the first time, \( T_n \) is the time taken to perform the task the \( n \)-th time, and \( a \) indicates how much performance improves from one trial to the next. The power law is quite robust, covering a wide range of activities from typing to carrying out sequences of operations.

The system will periodically examine a trace of user actions and consider alternative menu structures and dialogue box designs, evaluating each of them against the keystroke performance model applied to the trace and redesign. If the best reorganization found is
predicted to improve user performance within a reasonable number of uses, then the reorganization will be suggested to the user, who will be free to accept or reject the suggestion.

To suggest command macros, the chronological history of interactions will be periodically examined to find repeated sequences of commands. The records of the sequence and timing of actions, parameter values specified by the user, and the interaction technique used for each action or parameter are considered. It will be necessary to identify which command parameter values are literals of the macro, and which are to be variables specified by the user when the macro is invoked. Also, variable-length repeating sequences of commands might take as a variable the number of iterations. In general, identifying macros faces the same challenges as does programming tasks by example, such as are discussed in (Cypher, 1991).

7 IDE Support for Adaptive Help

UIDE currently supports context-sensitive animated help which is automatically generated using the knowledge of an application. UIDE also supports textual help explanations of why an interface object is disabled, and how to enable it. The textual and animated help are being integrated to provide a more complete yet automatically generated help for users by showing and explaining procedural information.

Within the scope of the kind of help UIDE can currently generate automatically, we are developing 2 types of help adaptation: selection of preferred procedural steps and variation of help contents. For the first type of help adaptation, given that there is more than one way to achieve a task, we would present procedural help using methods assumed familiar to the user. The second type of help adaptation is still a research issue. We assume that the user who revisits a help topic in a short period of time must seek for more of or different kinds of information from what she could grab the first time. We plan to research on user's help request patterns and usage of relevant help information. By analyzing both statistical and chronological recordings in the user model, we hope to be able to help users get at required information effectively.

Selection of Preferred Procedural Steps:

The statistical history of interactions allows UIDE to assume that a technique used frequently by a user is a preferred technique. When help on an action which can be performed by multiple interaction techniques is requested, help can demonstrate how to
perform the action based on the techniques which are used more often by the user. This is assuming that the focus of the user is to get the task at hand done without having to learn new techniques. However, if the purpose of help is to introduce alternatives to the user, alternative but less familiar interaction techniques should be demonstrated. A more thorough help in fact should present alternative techniques as well. Adaptive help would present more familiar techniques as a primary way to perform an action, while less familiar techniques are presented as alternatives.

Varying interaction techniques for an action is one level of adaptation in UIDE. Another level of adaptation is at application action level where a series of actions to perform a task is presented in help. For example, when UIDE presents how to make a widget enabled, a series of actions is presented. Planning techniques are applied to derive this series of actions using pre- and post-conditions of actions as the search mechanism. Another example is when animated help occasionally cannot demonstrate a procedure within the current context because not all pre-conditions of the action for which help is requested are true. A planner is then invoked to search for actions which will satisfy the pre-conditions. In essence, the planner is invoked to derive a path which accomplishes a goal, and a goal is to make certain conditions true. During the planning process, the planning algorithm may find that a condition can be satisfied by several paths of actions. An heuristics is needed to determine which path is the most appropriate. Using the statistical history of interactions collected in the model for each user, a path with more actions already familiar to the user is assumed to be the easiest to learn. The statistics in the user model is used to choose among alternative actions, and planning proceeds based on the chosen action. Different users have their own statistics and therefore may see the same procedure delivered differently.

**Variations of Help Contents:**

When the user revisits a help topic in a short period of time, we propose that the system should adapt to assist users in getting the needed information as effectively as as conveniently as possible. Help contents can be adapted in between sessions using the chronological recording of help requests. For example, the first time help on an action is requested by a user, UIDE would elaborate how to perform the action in detail using both textual and animated help. Textual help explains the underlying concepts of the action and animated help demonstrates how the action can be carried out. This is based on our assumption that textual help is brief, quickly presents the overall process while animated help is direct, visual, but time-consuming. After one (or more) requests on the same action, animation can be faster and briefer leaving out continuity of animation. Repeated
requests on the same action over a period of time should result in animation being left out altogether, unless requested by the user. On the contrary, if the user fails to select an appropriate interaction technique or to accurately locate related interface objects on the screen, help should start using animated demonstrations again. We should be able to predict, from the types of errors the user makes when performing an action, which parts of help contents the system should further elaborate and which media to use.

Unlike adaptive interfaces, adaptive help which varies help contents does not require relearning and can be presented without users being aware of it. This is similar to the case of human-to-human communication in which speakers often adjusts their conversations to fit current contexts and listeners. We are interested in how varying help contents, and which variation strategies, would be more effective in getting users to achieve their tasks. What we have presented in this section is adaptive help capability which UIDE can support; to use them effectively depends on our future research on relationships between information collected in the user model and individual adaptation strategies used.

8 Current Status and Future Work

As mentioned earlier, UIDE was developed in ART and Smalltalk-80 in the past. UIDE is currently being developed in C++ running on Sun Workstations and X server. Interfaces are created using OLIT widgets (Sun Microsystems, 1992) as well as our own object sets supporting visual representations of application objects. Major development work has been on representing knowledge entities in the model to represent aspects of interfaces we need to control, and supporting the automatic sequencing control using the specifications in the knowledge base to reason which actions are enabled and how the interface context should change to accommodate the next step in an action. The work on animated help was implemented in Smalltalk-80 and is now being transferred independently to a C++ environment. The work on textual help generation of why a widget is disabled, and how to enable it, has been developed separately in a C++ environment. The work on adaptive interfaces and adaptive help behavior has not been implemented, though changing menu and dialog box layouts can be done fairly easily in UIDE.

Our short term plan is to integrate the core UIDE environment with the automatic help generation mechanisms, and to widen the support base for various kinds of OLIT widgets and presentation objects. Currently the digital circuit layout program is the only application we have modeled. We plan to model another larger application to include more interesting cases for adaptive interfaces and adaptive help.
Our future work includes developing task representations which capture a longer sequence of interactions beyond the action level. We hope to use task representations to guide user tasks and apply adaptivity at this level.

We are now focusing at low-level concerns about adapting an interface – how is one easily created and how user model can be automatically supported in a user interface tool. The next challenging research issues lie in interfaces for those who design and those who benefit from adaptive behavior. For the designer, how should an interface to the designer look like? How do we allow the designer to instruct UIDE to collect additional information in the user model? How does one create a new adaptive feature and how does one relate information in the user model to the new interface feature? As for the user, how does one understand and realize adaptations and how can one customize adaptive interfaces?

9 Conclusions

In this paper, we have described UIDE, a user interface design environment which supports interface design process from creating an interface to executing one. The environment centers around the knowledge model of an application to which an interface is being designed and supported. Various tools which are developed to aid in the design process and at run-time are mentioned in the overview of UIDE in Section 2. We elaborated the knowledge model in great details in Section 4 and described how the various entities and their semantic links in the knowledge model are used at runtime. By elaborating on the knowledge model and how UIDE operates at runtime, we hope that the readers can easily see how a user model can be automatically constructed and information collected at runtime. Section 4 should give a parallel view of information represented in UIDE's knowledge model and information frequently needed in a certain class of a user model. We then described about different recordings in UIDE's user model which follows the control sequencing mechanisms of UIDE. Finally, we discussed possible adaptive interface – reorganization of menus and dialog boxes and macro suggestion – and adaptive help behavior which can be supported in UIDE utilizing the user model.

10 Acknowledgements

This work has been supported by the Siemens Corporate R&D System Ergonomics and Interaction group of Siemens Central Research Laboratory, Munich, Germany, and the Human Interface Technology Group of Sun Microsystems through their Collaborative Research Program. 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: J.J. "Hans" de Graaff, Martin Frank, Mark Gray, Srdjan Kovacevic, Ray Johnson, and Krishna Bharat. We also thank colleagues and former students at the George Washington University who contributed to UIDE: Hikmet Senay, Christina Gibbs, Won Chul Kim, Lucy Moran, and Kevin Murray.

11 References


Branka Jovanovic