A Second Generation User Interface Design Environment: The Model and The Runtime Architecture

Piyawadee "Noi" Sukaviriya
James D. Foley
Todd Griffith

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

ABSTRACT
Several obstacles exist in the user interface design process which distract a developer from designing a good user interface. One of the problems is the lack of an application model to keep the designer in perspective with the application. The other problem is having to deal with massive user interface programming to achieve a desired interface and to provide users with correct help information on the interface. In this paper, we discuss an application model which captures information about an application at a high level, and maintains mappings from the application to specifications of a desired interface. The application model is then used to control the dialogues at runtime and can be used by a help component to automatically generate animated and textual help. Specification changes in the application model will automatically result in behavioral changes in the interface.

KEYWORDS: Application Model, User Interface Model, User Interface Generation, User Interface Design Environment, Automatic Help Generation

1. INTRODUCTION

It is desirable to have user interface system which allows application programmers and interface designers to create interfaces without programming and without having to painfully learn tedious details of underlying toolkits. In 1988, we reported on UIDE, the User Interface Design Environment, which utilizes a model of an application to automatically create an interface [4]. Our vision however goes far beyond removing the programming burden from the user interface design process. We envision an environment which utilizes the model of an application to assist designers in the design process. We also see an environment where, not only are interfaces generated automatically, but also generated is procedural help on the interface utilizing the same model of the application in question.

UIDE emphasizes a model-based approach where the existence of an application model enables several benefits [14]. An application model enables an interface to be designed and analyzed based on application tasks. The runtime environment can, in the case of UIDE, use the application model to sequence user dialogues based on task descriptions. The application model also enables the runtime environment, which knows about application tasks, to help the user in a more task-oriented fashion. We are currently using the model to automatically construct individual user models for user performance analysis [23], the results of which will be used to support adaptive interfaces and adaptive help.

We realized that the first generation model as reported in [4, 5] was just a start, and as yet is far from being sufficient to provide sophisticated interface features. Since then, we have invested our efforts in revising the model and strengthening both design-time and runtime support capabilities. In 1992, UIDE consists of a much more refined model describing not only the application, but also detailed interface features for the application interface. Its implementation platform evolved, from the Automatic Reasoning Tool [8], to Smalltalk-80, and now to C++. In this paper, we will only focus the description of UIDE's application model and will give an overview of the UIDE C++ runtime architecture.

2. BACKGROUND ON UIDE

UIDE is designed to allow interface designers to easily create, modify, and generate an interface to an application through high-level specifications. The purpose of the environment is to support the interface design process through its life cycle – from its inception to its execution. This support is made possible using a model which describes various details of an application interface including partial 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 runtime support which can be automatically provided using the model. Design assistance includes automatic layout of menus and dialogue boxes [3,9], and transformations on interface styles using a compositional model of the user interface [10]. Runtime support includes using pre- and post-conditions to control interface objects [6], automatic generation of animated help [20], and automatic generation of textual help [22]. Should the

specifications in the knowledge base change, the generated interface and help at run-time will change accordingly.

Through UIDE, a designer creates an interface for an application by first describing objects and operations in the application. The designer then chooses various interface functional components which best fit the application by linking them to application operations. Interface functional components include interface actions, interface objects, presentation objects representing application objects, and interaction techniques to be used with presentation and interface objects. These components will be described in more detail in Section 4.

Though the model has descriptions of both application tasks, and interface tasks and objects, we often refer to it as "application model." The terms "model" and "application model" will be used interchangeably throughout this paper.

Once the application model is mapped to interface functional components, the UIDE runtime environment uses the specifications in the model to generate a desired interface. The specifications are also used to control user interactions and status of interface objects at run time. While the interface is running, context-sensitive animated help on "how" to perform actions can be constructed from the specifications upon user request [20]. Help on "why" a certain widget is disabled can also be generated using pre-condition information associated with actions and interaction techniques related to the disabled widget [6].

3. RELATED WORK

Direct manipulation user interface builders such as DevGuide [24] and Prototyper [2] make designing an interface one step easier. Some direct manipulation tools such as DialogEditor [1] and Peridot [12] are intelligent and they make the interface design process less tedious and less repetitive. These tools facilitate a bottom-up approach of interface design while UIDE emphasizes a top-down approach. That is, designing a conceptual design of an application before designing its interface. Also, these tools mainly assist with layouts and do not incorporate application aspects in their assistance.

Cousin [7], MIKE [16], MICKEY [17], UofA* UIMS [19], and HUMANOID [26] use descriptions of commands and parameters to automatically generate interfaces. The same approach is used in UIDE. It is used, however, with a more sophisticated representation structure. Except for HUMANOID and UIDE, the systems above parse textual descriptions of commands and parameters, either embedded in a program structure or stored in a file, and use them as inputs to interface generators in their systems. UIDE and HUMANOID store descriptions of application actions and parameters as objects in their application model.

All these systems except UIDE do not explicitly model detailed descriptions of interface tasks beyond application commands and parameters. We explicitly model interface actions and interaction techniques, which are intended to be used by designers as ways to specify interaction styles for an application. UIDE's application model allows the mapping of application actions to interfaces to be changed without changing application actions. The model also allows an application action to be mapped to multiple interface actions; this results in multiple access to a single application command, for example.

Both HUMANOID and UIDE have explicit descriptions of application objects. They also have pre-conditions for actions which are used to determine when an action is enabled and can be performed by the user. Only UIDE uses post-conditions of actions as a way to capture application action semantics, and uses them to reason in the design process and to provide help at runtime.

4. APPLICATION MODEL

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

UIDE provides generic classes of various entities in the application model. These classes will be introduced in the next 5 subsections – Application Objects, Application Actions, Interface Actions, Interaction Techniques, and Interface Objects. The application model consists of instances of these classes; each instance contains information specific to an application aspect it represents or the particular interface features chosen for the application. The power of the model comes from the specific contents of these instances and also the semantic connections among these instances.

Application Objects

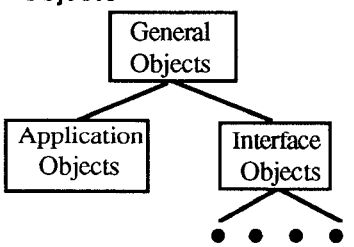

Figure 1 UIDE Object Class Hierarchy

Figure 1 shows a predefined object class-subclass hierarchy in UIDE. *Application objects* are specific entities in an application; they are entities on which application actions operate. *Interface objects* are entities which users perceive and sometimes interact with, such as a window, an icon of

a NAND gate, or an audio output, etc. We will discuss more details on interface objects later in this section.

The application designer defines object classes in an application by subclassing from the UIDE application object class. Attributes of these classes are added by the designer. Figure 2 shows an example of an extended class hierarchy for a digital circuit design 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 paper.

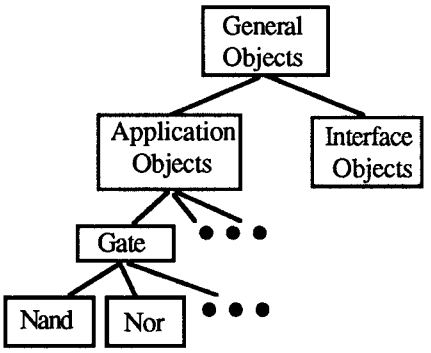

Figure 2 Object Class Hierarchy for a Digital Circuit Layout Application

The extended class hierarchy for application objects is often referred to as the data model. We consider the data model to be part of the application model. In an application such as the circuit design program where objects are instantiated at runtime by end-users, the designer only designs class definitions of objects. Operations to create instances of these objects should be provided in the interface. In different applications where there are a fixed number of objects pre-created for end-users to interact with, the class definitions of these objects must be defined and instances created during the design process.

Application Actions

To complete the application model, the designer describes the functionality of an application which will be available to the user as a set of actions. UIDE provides *action*, *parameter*, *parameter constraint*, *pre-condition*, and *post-condition* classes to capture application actions and their semantics.

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 [21]. 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 use first order predicate logic (but do not yet handle universal and existential qualifiers), 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. Figure 3 depicts a graphical view of an action with links to its parameters, its pre-conditions, and its post-conditions. Each parameter is also linked to its constraints.

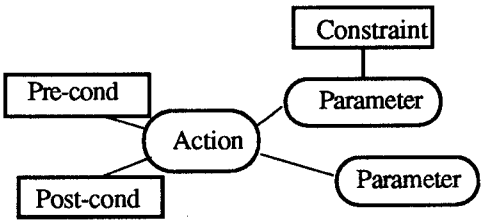

Figure 3 A View of an Action and its Associated Components.

Examples of application actions and their parameters in the digital circuit design application are:

create-NAND-gate	(location)
create-NOR-gate	(location)
delete-gate	(gate)
move-gate	(gate, new-location)
rotate-gate	(gate, angle-of-rotation)
connect	(gate-1, gate-2)

Figure 4 shows the "connect" action instance with its related entities.

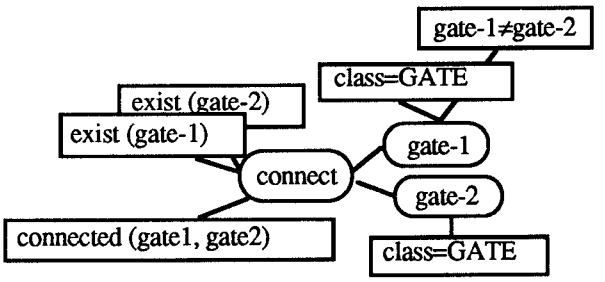

Figure 4 The *connect* Action from a Digital Circuit Design Application.

Interface Actions

Mapping from application components – actions and their parameters – to an interface is specified as links from these components to *interface actions*. The designer does not really define interface actions. There are instances of different interface actions already defined in UIDE from which the designer can clone. Multiple interface actions can be linked to application components of one action. For example, the "rotate" action may be linked to a "select-action" interface action. Its parameters, object and angle-of-rotation, may be linked to "select-object" and "enter-integer-in-dialogue-box" interface actions respectively. On the

current C++ platform, linking is done through C++ code. We do not yet have a friendly interface to the model acquisition process.

Interface actions may have pre- and post-conditions. Some pre-conditions are concerned with the status of an interface prior to performing interface actions. 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. Some pre- and post-conditions are concerned with sequencing of interface actions. 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. Sequencing pre- and post-conditions can be automatically derived by UIDE from the application semantics [10].

Interaction Techniques

Interaction techniques specify how interface actions are to be carried out by the user. Interaction technique instances are linked to interface actions. For example, if the "select-object" action is to be performed by using the mouse to click on an object, a "mouse-click-object" technique must be linked to the action. More than one interaction technique can be linked to an interface action to designate possible alternative interactions. For instance, an object can also be selected by typing in the object name. In this case, a "type-string" technique is attached to the "select-object" action.

UIDE's interaction techniques are similar in function to Myers' interactors [13]. We, however, do not model feedback as objects in UIDE. We treat feedback as different behaviors of presentation objects. Also, we do not treat an object as a parameter of an interaction technique. Rather, as will be described later, interaction techniques are attached to objects.

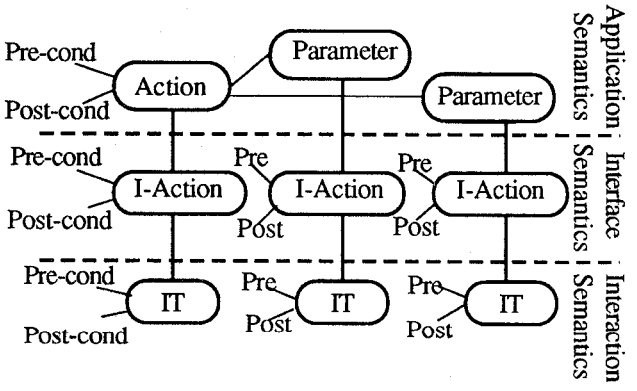

Figure 5 Specifications Layers in the Application Model

Interaction techniques may also have pre- and post-conditions. Their pre- and post-conditions tie closely to the screen context, for example, 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 beforehand so their references can be used in the specifications in the application model.

Figure 5 depicts the overall layers in the application model for each action and figure 6 shows a graphical view of the semantic links of the "rotate" action. We omit pre- and post-condition details, and parameter information to keep the figures clear. The application semantics level serves as a starting point in the design process and is independent of the end-user interface. The application model consists of several application actions described in this structure. The same application can be mapped to multiple sets of interface actions and techniques for different environments.

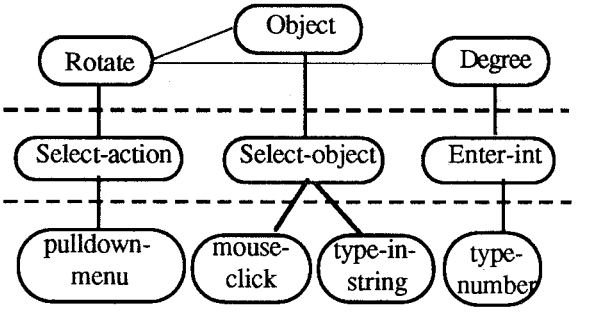

Figure 6 A *Rotate* Action Instance

Interface Objects

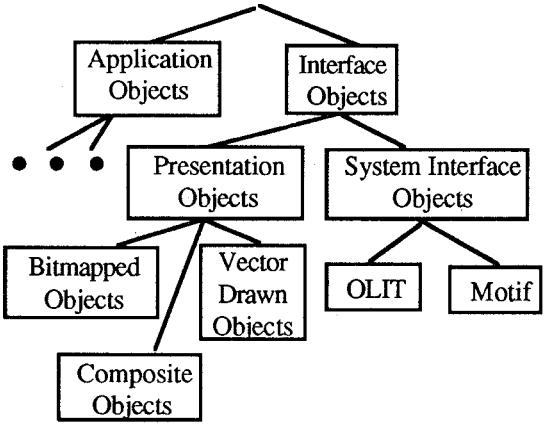

Figure 7 Interface Object Subclasses

Figure 7 illustrates the UIDE class-subclass hierarchy down from the interface object class. The 2 major subclasses of interface objects are: *presentation objects*, which represent application objects and are presented to the user through various forms or media; and *system interface objects*, which are standard objects in interface toolkits such as OLIT [25] or Motif [18]. Currently we are only working on the OLIT widget set.

The system interface object allows common interface objects to be referred to as buttons, scrollbars, etc., regardless of the underlying toolkits. Many of our interface object classes are similar to interactor classes in InterViews [11] and we maintain a part-whole structure for composite objects such as dialogue boxes in a similar fashion.

However, unlike InterViews, we allow interactive behaviors to be programmed on interface objects by attaching interaction techniques to interface object classes or instances. This is used more in the case of presentation objects where different behaviors may be desired for different forms of presentation. We have not yet supported adding behaviors to standard widgets, though this can be done using our current underlying mechanisms.

To make this point clear, let's assume that the designer wishes to make all gates movable by dragging, but only double clicking on a NAND gate would allow the user to change the time delay in nanoseconds of that gate. To achieve such a design, the designer first creates a subclass of the Bitmapped object class called GATE, then attaches a "mouse-drag" interaction technique to the class. Notice that this "mouse-drag" technique is the same instance which is linked to the "move-gate" application action. The designer then creates a NAND subclass of the GATE presentation class and attaches a "double-clicking" technique to the class. This "double-clicking" technique would be one which is linked to a "change-NAND-time-delay" action.

Creating the Model

The designer furnishes an application model for UIDE by first defining application actions, application and interface objects, then choosing interface actions and interaction techniques as she sees appropriate. This process creates the declarative model which serves as specifications for UIDE to sequence dialogues for the application. Currently, defining the model in the C++ version of UIDE is still done through the standard C++ declaration and instantiation syntax. A visual tool to create an application model has been planned.

There are constraints on which interface actions and techniques are appropriate for which types of interface objects. There are also constraints on which interface actions are appropriate for which parameter types with which they are associated. These constraints vary from rigid rules, such as a radio button bank cannot be used for entering a floating point numeric value, to style guide-like rules such as a long list of choices should be presented in a scrolling list. Declaring the model using the C++ syntax does not prevent the designer from using interface actions and interaction techniques inappropriately. It is important to incorporate this kind of constraints to assist designers in a tool for creating the model.

In addition to the model, the designer also provides, for each application action, an application routine which will be invoked at runtime when all parameter values for the action have been entered or selected. Application routines are responsible for handling necessary computations and manipulations in an application. Occasionally, application routines may need to address contextual changes in the interface. Currently, once an application object is created and registered, its corresponding presentation object will be displayed. This works well for the circuit design application we are using. More sophisticated dependencies between application data and its interface need to be modeled.

With application routines in place, UIDE is ready to support the runtime execution of an application interface.

5. RUNTIME ARCHITECTURE

Controlling the Interface Through the Model

Most components in the application model are instances of a C++ class. In addition to members and member functions which are used to maintain specific information contents and semantic relationships in the model, these instances are designed to support runtime execution of the interface. Some examples are: 1) each system interface object has the actual calls to corresponding OLIT widgets to create or modify themselves; 2) parameter instances maintain current parameter values; and 3) parameter constraints are used to validate parameter values, etc. The UIDE runtime architecture is designed such that application action semantics 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 application specification in the model changes, the interface sequencing and behaviors change accordingly. In addition to instance behaviors in the model, UIDE has a "dialogue manager" which maintains the current interface context, sequences the dialogues, and updates the context when appropriate.

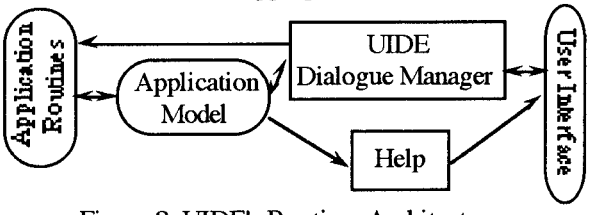

Figure 8 UIDE's Runtime Architecture

Figure 8 shows information flow among various components in UIDE's runtime architecture. The dialogue manager updates the screen context appropriately after the user finishes an interface action – an interface action could be the user has just selected an action or an object. The dialogue manager maintains the user interface context to keep track of which action the user is performing. Each application action has a pointer to its corresponding application routine. Once all parameter values of an action have been entered by the user, it invokes the application routine for that action. The application routine may update data in the data model which resides in the application model. For an application which resides on the same C++ platform such as the circuit application we use, the data model can be directly used as the data structure of the application.

A general theme for controlling the interface is as follows. First, the dialogue manager checks to see whether each application action has all its pre-conditions satisfied. A combination of: 1) traditional unification and resolution mechanisms for the predicate language [15], and 2) treating pre-condition as functions for checking conditions on

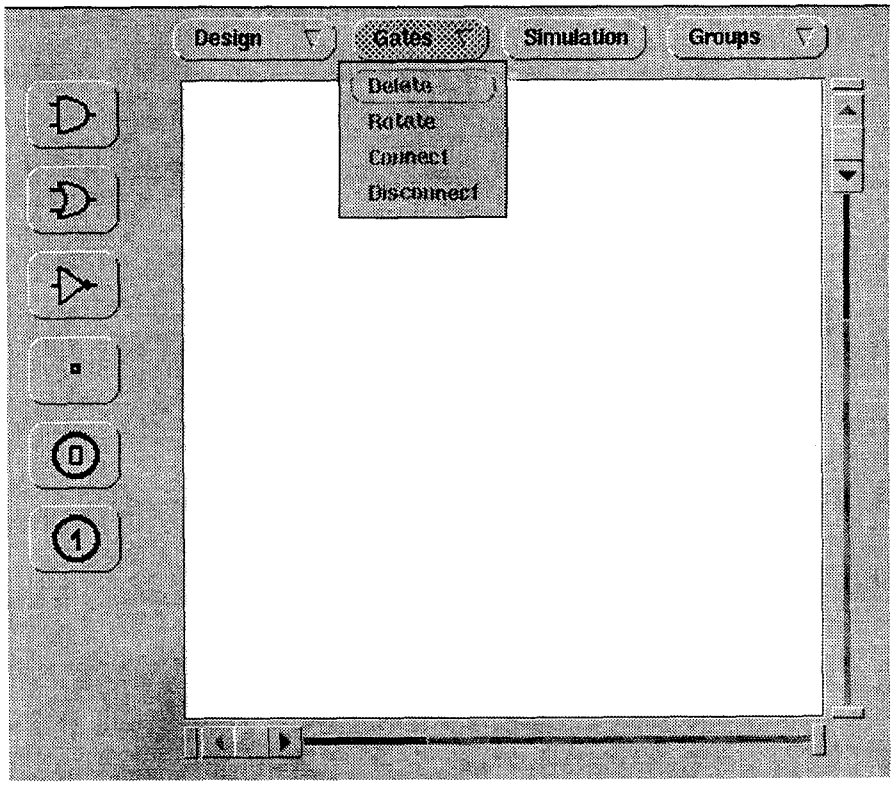

Figure 9 Interface to the Circuit Design Application

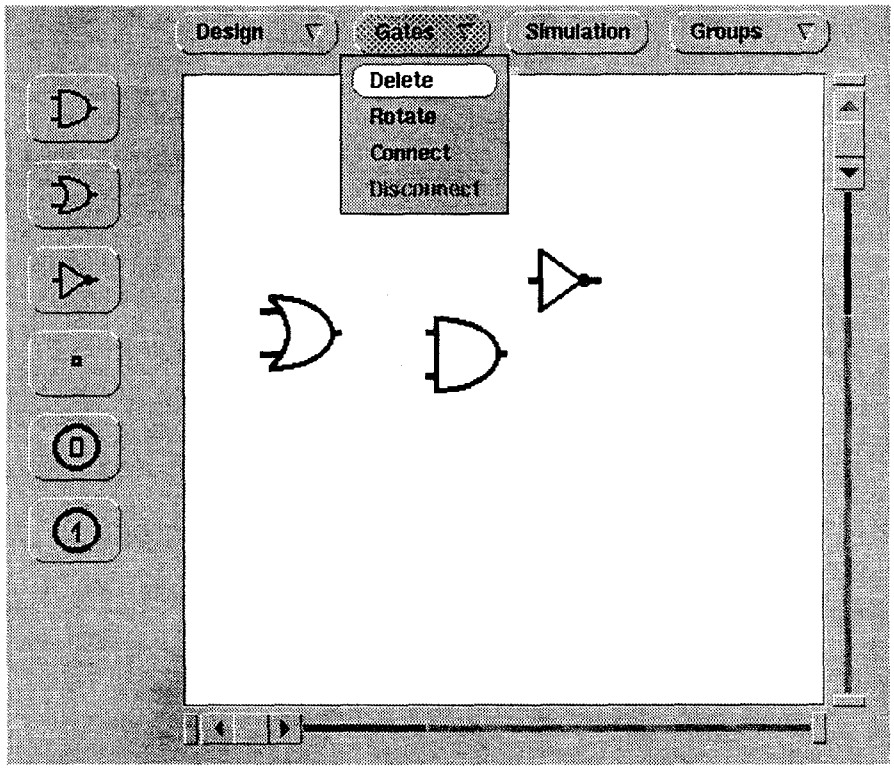

Figure 10 Interface to the Circuit Design Application with some Gates in the Design.

objects, are used for handling pre- and post-conditions. An action with all its pre-conditions satisfied is enabled, and an action with at least one pre-condition unsatisfied is disabled. This situation is reflected in the interface through links in the model. That is, the interface actions and interaction techniques associated with enabled actions are enabled. Interface objects associated with enabled interaction techniques are also enabled and their statuses are changed (i.e, from invisible to visible). Interface objects associated with disabled interaction techniques are disabled and their statuses are also changed (i.e, from solid to greyed-out) Figure 9 and 10 illustrate examples of these conditions.

Notice from the interface to the circuit design application in figure 9 that the gate icons on the left are for invoking application actions which instantiate the corresponding gate types. When one of these icons is selected, UIDE waits for a location input in the current design space. Once a location is given by the user, a gate is instantiated and its graphical view is placed at the given location. In figure 9, since there are no gates in the system, all actions in the "Gates" menu are disabled because all of them require that some gates exist in the design. This interface assumes a prefix interface paradigm – an action is selected first before objects and/or parameter values for the action are selected. Figure 10 shows the menu changed as some gates have been created.

This design can be changed so that an object must be selected before some actions on gates, "delete" and "rotate" for example, are enabled. This is done by making "select-object" interface action an action by itself (as opposed to linking to an application action), which results in a post-condition that an object exists as the current selected object (CSO as referred to in [4]). The resulting interface would be one that, though gates exist, "delete" and "rotate" menu items are not enabled unless one of the gates is selected.

Automatic Generation of How and Why Help

Two kinds of help can be automatically generated from the application model: **how** to perform an action and **why** an interface object is disabled. We use 2 types of help presentation modes, textual and animated help.

It should be obvious now that procedural steps for completing an application action can be inferred from the application model. For example, rotating an object can be done by first selecting the *rotate* action, selecting an *object* to be rotated, and then entering the *degree* of rotation. To animate this action, UIDE chooses an object of type GATE from the current context using the constraint for the "object" parameter which state that the object must be of type GATE. An integer number between 0 to 360 is chosen for the "angle of rotation" parameter according to its constraints. UIDE then illustrates the procedure by showing a mouse icon on the screen, the left mouse button being pressed while pulling down the menu where the "rotate" item is located, releasing the mouse button at the "rotate" item, clicking on the chosen object, and typing the number chosen for the "angle-of-rotation" in the dialogue box designed for this action.

Pre- and post-conditions are used to evaluate whether a context is ready for animating an action. For example, to rotate a gate, a gate must exist. If there is no gate in the current context, a planner is invoked to search for an action in the application model which will create a gate. The animation will show creating a gate first, and then rotating the gate.

A interface object is disabled if the interaction technique associated with it is disabled. An explanation of why an interface object is disabled can be generated using pre-conditions [22]. The explanation also includes which actions must be performed the make the object enabled.

6. IMPLEMENTATION

The 1992 UIDE is written in C++ using the OLIT widget set. It runs on Sun's X Server. Though we use OLIT, UIDE does not generate conventional code with callbacks for application programmers to add their application code. All application routines must be written separately and are connected to UIDE through application actions.

A subset of the OLIT widget set has been incorporated in UIDE's interface object structure. Our major efforts have been placed on connections from the application model to the actual execution of the interface. The animated help was developed earlier in Smalltalk-80 and is now being ported to C++. Generation of WHY explanations has been developed separately in C++ but has not yet been integrated into UIDE.

7. CONCLUSIONS

The application model in UIDE consists of entities which describe an application through actions, action semantics, objects, and interface details of its interface. We have explained the structure of a model which captures the semantic relationships among these entities. We also described the runtime architecture which 1) controls the dialogue sequencing, and 2) automatically generates how and why help using the model.

Taking away the burden of having to program an interface is already a major relief to most of us who need to create an interface and who have no interest in unnecessary details which distract us from the design process. However, a greater contribution of UIDE lies in the fact that it allows application designers to focus on modeling the application even before being hassled by interface design issues. The designer can later link the high-level model of the application to interface details, and can therefore focus on tasks and communication means provided to users through the interface. This contribution is one of many benefits enabled by the model-based user interface design approach.

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. We also thank Thomas Kuehme, John Stasko, John Shilling, and the reviewers for their comments.

REFERENCES

1. Cardelli, L. Building User Interfaces by Direct Manipulation. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*. Banff, Alberta, Canada. October, 1988, 152-166.
2. Cossey, G. *Prototyper*. SmetherBarnes, Portland, Oregon, 1989.
3. de Baar, D.; J.D. Foley; and K.E. Mullet. Coupling Application Design and User Interface Design. In *Proceedings of Human Factors in Computing Systems, CHI'92*. May 1992, 259-266.
4. 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.
5. Foley, J.D.; W.C. Kim; S. Kovacevic; and K. Murray. UIDE–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.
6. 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.
7. Hayes, P.J.; P. A. Szekeley; and R.A. Lerner. Design Alternatives for User Interface Management Systems Based on Experience with COUSIN. In *Proceedings of Human Factors in Computing Systems, CHI'85*. April 1985, 169-175.
8. Inference Corporation. *ART Reference Manual*. Inference Corporation, Los Angeles, CA, 1987.
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 Compositional Model of Human-Computer Dialogues. In *Multimedia Interface Design*. Eds. M.M. Blattner and R.B. Dannenberg, New York, New York: ACM Press, 1992.
11. Linton, M.; J.M. Vlissides; and P.R. Calder. Composing User Interfaces with InterViews. *IEEE Computer* 22(2): 8-22, February, 1990.
12. Myers, B. Creating Interaction Techniques by Demonstrations. *IEEE Transactions on Computer Graphics and Applications* 7 (September 1987):51-60.
13. Myers, B.; D.A. Giuse; R.B. Dannenberg; B.V. Zanden; D.S. Kosbie; E. Pervin; A. Mickish; and P. Marchal. Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces. *IEEE Computer* 23(11): 71-85, November, 1990.
14. Neches, R.; J.D. Foley; P. Szekeley; P. Sukaviriya; P. Luo; S. Kovacevic; and S. Hudson. Knowledgeable Development Environments Using Shared Design Models. To appear in *Proceedings of Intelligent Interfaces Workshop*, Orlando, Florida, January 4-7, 1993.
15. Nilsson, N.J. Principles of Artificial Intelligence. Los Angeles, CA: Morgan Kaufmann Publishers, Inc, 1980.
16. Olsen, D. MIKE: The Menu Interaction Kontrol Environment. *ACM Transactions on Graphics* 5,4 (1986): 318-344.
17. Olsen, D. A Programming Language Basis for User Interface Management. In *Proceedings of Human Factors in Computing Systems, CHI'89*. May 1989, 171-176.
18. Open Software Foundation. OSF/Motif Style Guide. Revision 1.0. Englewood Cliffs, New Jersey: Prentice Hall, Inc., 1990.
19. Singh, G. and Green, M. A High-Level User Interface Management System. In *Proceedings of Human Factors in Computing Systems, CHI'89*. May 1989, 133-138.
20. 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.
21. Sukaviriya, P. *Automatic Generation of Context-sensitive Animated Help*. A Ds.C. Dissertation, George Washington University, 1991.
22. Sukaviriya, P. and de Graaff, J. Automatic Generation of Context-sensitive "Show & Tell" Help. *Technical Report GIT-GVU-92-18*. Atlanta, Georgia: Graphics, Visualization, and Usability Center, Georgia Institute of Technology, 1992.
23. Sukaviriya, P. and J.D. Foley. Supporting Adaptive Interfaces in a Knowledge-Based User Interface Environments. To appear in *Proceedings of Intelligent Interfaces Workshop*, Orlando, Florida, January 4-7, 1993.
24. SunSoft. *OpenWindows™ Developer's Guide 3.0 User's Guide*. Sun Microsystems, Inc. Part No:800-6585-10, Revision A, November 1991.
25. SunSoft. *OLIT 3.0.1 Reference Manual*. Sun Microsystems, Inc. Part-No: 800-6391-10, Revision A, june 1992.
26. Szekeley, P.; P. Luo; and R. Neches. Facilitating the Exploration of Interface Design Alternatives: the HUMANOID Model of Interface Design. In *Proceedings of Human Factors in Computing Systems, CHI'92*. May 1992, 507-515.