A Model-Based User Interface Architecture: Enhancing a Runtime Environment with Declarative Knowledge

by

P. "Noi" Sukaviriya, M. Frank, A. Spaans, T. Griffith, K. Bharat, and J. Muthukumarasamy

GIT-GVU-93-12
April 1993

Graphics, Visualization & Usability Center

Georgia Institute of Technology
Atlanta GA 30332-0280

A Model-Based User Interface Architecture: Enhancing a Runtime Environment with Declarative Knowledge

P. "Noi" Sukaviriya
M.Frank, A.Spaans, T. Griffith, K. Bharat, J Muthukumarasamy

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

ABSTRACT
A model-based user interface environment refers to a runtime environment which contains meta-knowledge about an application and its connection to the interface presented to the user. We capture in our *application model* actions which can be invoked by users, their operational constraints, and data objects on which these actions operate. We capture in our *interface model* interface components, actions which users apply to these interface components, and operational constraints on these actions. We then capture the mappings from the application model to the interface model as a way to construct an interface to an application. The modeling components used in the application and interface models are active at runtime, and are the key which drives the runtime architecture to support intelligent behavior such as partially automatic control sequencing, automatic generation of textual and animated help, and recordings of statistical and chronological command usage history. Also, the modeling components used are task-oriented; specifying an interface through these components eliminate the low-level programming from the user interface creation process.

KEYWORDS: Model-based User Interface, Automatic Generation, User Interface Environment, Help, Animated Help

1. INTRODUCTION

A model-based user interface architecture is based on representing computer processable information about an application in a runtime environment. One of the benefits of having such knowledge is, user interface control and generation responsibility can be delegated from designers and application programmers to the underlying mechanisms in a user interface environment. Our current model captures user operations and data objects within an application, how this level of information maps to interface operations and objects, and how information flows from one level to the other. In addition to automatic control sequencing, other intelligent support can be provided utilizing the model, such as automatic generation of textual and animated help, and recordings of user's action history. These benefits are application-independent; every application with proper representations would have access to the same support.

The research prototype reported in this paper is an extension to UIDE, the User Interface Design Environment. 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. The first generation UIDE was reported in [7, 8]; its emphasis has been on the design support, and not so much on the runtime environment. Its declarative knowledge lacked details of the interfaces which it supported. We have expanded our research to strengthen the quality of runtime support which can be automatically provided from the model. We have also changed the implementation platform from the Automatic Reasoning Tool [14] and Smalltalk-80 to C++ to take advantage of speed, standard interface toolkits, and emerging input/output technology. The C++ environment currently has a greater emphasis on the runtime environment. We continue to call it UIDE, yet it may have to be renamed since now "D"esign is not the only focus of our research.

Within the past couple years, we developed a few small and separated C++ prototypes which take advantage of application knowledge to provide automated behavior. Details of these prototypes can be found in [5] on design assistance which automatically laid out menus and dialogue boxes, in [10] on using pre- and post-conditions to control interface objects at runtime, and in [6] on automatic generation of textual help. Our research focus has been placed on forming the central representations and designing a comprehensive user interface architecture which functions with the representations. We are in the process of changing platform, putting all these pieces together, and rebuilding the complete knowledge model for applications and interfaces. The architecture has been re-designed to fit the C++ and X environment better. We expect the representations and the architecture to advance over time, as more benefits can be realized from having processable semantic knowledge of applications available at runtime.

The details of the model representations have been reported in [23]. In this paper, we only give an overview of the model components which capture application semantics and application interfaces. We then focus on the runtime architecture, discussing how an interface are created and controlled. We then explain the runtime benefits which can leverage from having the model, such as an automatic generation of context-sensitive help, and recordings of user's statistical and chronological command usage history.

2. THE MODEL

The concept of the knowledge model is an extension from that of Cartoonist [21, 22] implemented in Smalltalk-80. We have described the UIDE knowledge model in detailed in [23]. However, for this paper, readers need to have a background on the basic conceptual structure of the knowledge model. Rather than repeating the descriptions of the knowledge model in detail in this paper, we will give a brief overview of the model in the following paragraph.

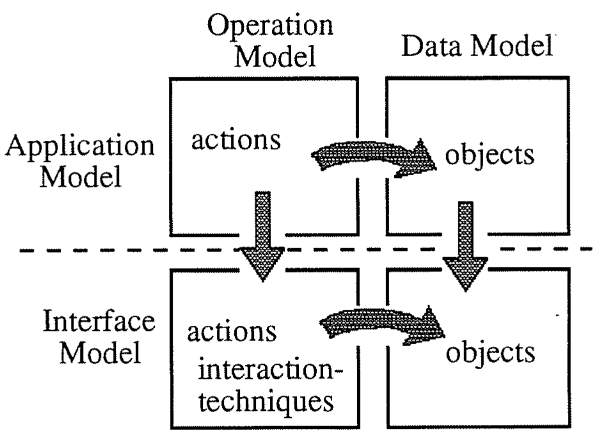


Figure 1 A basic scheme of UIDE's application model, interface model, and their interconnections

We use a two-level view as a basis for representing the knowledge about an application. Each level of representations consists of the **operational** model and the **data** model. The operational model contains declarative information about actions which users can perform within an application. The data model contains data objects on which the actions operate. The distinction between the operational and the data models is based on the original UIDE representation model [7, 8]. The top-level representations as shown in Figure 1 is the **application** model, of which information includes actions and objects specific to a particular application. The bottom level is the **interface** model, of which information includes interface actions and objects which will be present in the interface to the application. At the interface level, we also represent interaction techniques which specify the nature in which input devices are used to interact with interface objects. The same types of interface actions, interface objects, and interaction techniques share the same properties and behavior when used in various application interfaces. However, some behavior, such as contents of interface objects or sequencing of interface actions (which influences sequencing of interface objects, i.e. when a dialog box should pop up) can be customized and stored in the interface model of an application.

Each action in the operational model in both levels is defined with parameters stating inputs which must be acquired for the action so it can be executed. Each action is also defined with pre-conditions, which state the conditions which must be true before the action can be invoked, and post-conditions, which state the consequences of the action. Each object in the data model has a set of attributes. A class-subclass hierarchy and a part-whole hierarchy are presented in the data model.

It is the connections from actions to and from the application model to the interface model which glue pieces of an interface together in a way which is meaningful to an application. The connections between actions and objects are established through action-parameter relationships. The mapping from objects in the application model to objects in the interface model in many cases is just a pointer. For example, each logic gate points to a bitmap object which symbolically represents its existence in the interface. In cases where multiple views of an object are presented, a list of pointers can be used. Some interface objects such as buttons or menus indireclty connect to application actions through interface actions associated with them.

The mapping from the application's operational model to the interface's operational model is one-to-many. That is an application action is often mapped to a set of interface actions. These interface actions can be thought of as procedural steps to achieve the application action. The order in which these interface actions must be performed (by end-users) is dictated by the sequencing pre- and post-conditions added to the pre-defined pre- and post-conditions which come with each interface action.

Figure 2 illustrates a graphical view of this connection, in which each action has a mapping graph to interface actions. Each of the interface actions is linked to at least one interaction technique. The interaction techniques at the bottom are attached to the actual interface objects at runtime, providing gateways from actions performed by the user to the model.

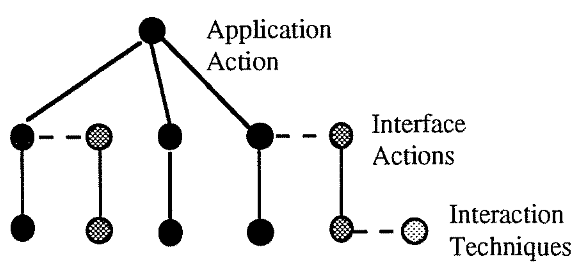


Figure 2 A mapping from an application action to interface actions. Dotted links denote alternative mappings, meaning the same step can be done by alternative interface actions or interaction techniques.

Cousin [12], MIKE [18], MICKEY [19], UofA* UIMS [20], and HUMANOID [25] use descriptions of commands and parameters to automatically generate interfaces. Their approaches center around the same essential concepts of representing command and parameter descriptions as UIDE. However, all of these systems except HUMANOID discard this information once interface programs are generated, therefore cannot make use of the information at runtime. Also, all these systems except UIDE have one-to-one mapping from their command descriptions to corresponding interface tasks and do not have detailed descriptions of interface tasks beyond commands and parameters. 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. We have partially consolidated representations and runtime capabilities of HUMANOID and UIDE as reported in [17].

One may compare the UIDE view with the Model-View-Controller (MVC) model [11]. At first glance, one may assume that the application's data model is similar to the (M)odel, the interface's data model to the (V)iew, and the interface model, interaction techniques in particular, to the (C)ontroller. We argue that our model is rather different. First, the existence of declarative information in parallel with a functioning model makes a big difference in terms of having knowledge to infer from. Declarative information especially that of operations is not present in the MVC model. Second, effects of user actions in our model do not propagate directly from the view to the model. Rather, they propogate through the operational models where consequences of actions can be inferred and made declarative for further reasoning. Also, our application and interface models are user-task oriented and would be easier for designers to use. The MVC model is a much more system-oriented view of the application breakdown.

3. EXAMPLE

Let's use a concrete example. Assuming that the screen in Figure 3 is the interface for a digital circuit layout program which we want to create. By appearance, one would think that this interface does not differ from one which can be created through conventional programming. Internally, this interface is different; it has an underlying model. The interface components are those which have appearance on the screen while other components in the model are functional and most often are not prominent in an interface.

From Figure 3, each object on the screen is represented as an object in the model. The button "Gates" is an object of class Menu. "Gates" has 4 subparts, "Delete", "Rotate", "Connect", and "Disconnect" menu items, each of which is of class Menu Item. Those icons on the left are objects of a class ICON. An AND gate is an object of an application-specific class called AND-Bitmapped, which is a derived class of a Bitmapped-Object class. All these objects belong to the data model of the interface. The operational model of the interface consists of actions which can be performed on these objects, for example, a select-graphical-object action which applies to all gate objects, a select-an-icon action for all command icons, a select-a-button action for each command button, and a select-from-a-pulldown-menu action of each menu item. As you can see, having just the interface model captures some of the semantics behind the screen which is not necessarily meaningful without an application behind it.

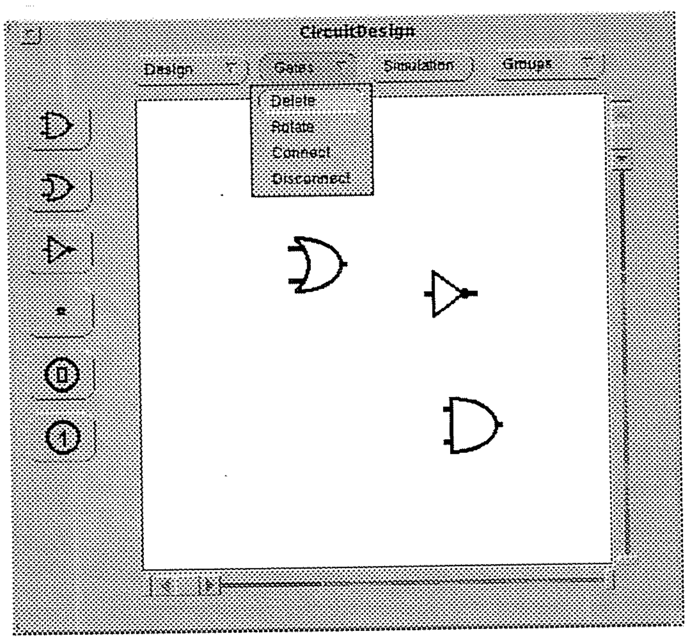


Figure 3
A Screen from a Circuit Design Application Created with the UIDE's Model

The interface model consists of objects which are created for the interface of the application, and actions which can be used on these objects. Objects at this level are called *interface objects*, and they include objects such as those provided by standard X toolkits (Motif or OLIT) and those which are application specific, such as file icons in a file management application, or various types of gates in a circuit layout application. We attempt to model these two classes of objects in a homogeneous fashion, though internally dealing with them at runtime is rather different. Those objects in standard toolkits have fix behavior and has their handling mechanism built in, while application-specific objects are handled in UIDE through interaction technique handling. We attempt to hide this difference and let designers model these objects in a homogeneous fashion.

A gate which appears on the screen, for example, is one form of presenting the concept of a gate to the user. As a matter of fact, each bitmap of a gate is classified as a presentation object in UIDE. One can imagine a gate being presented in a textual form, as a table of properties, and so forth. Each of these views is a presentation object in UIDE. The application object which each bitmap represents is of class GATE, which is defined in the data model of the application. Each gate has attributes such as number of inputs, number of outputs, delay time, connection status, etc.

Let's say a gate is to be created at runtime by the user, and the user can connect gates. Application actions "create" and "connect" must be defined in the application model. The textual representations for a create-gate action and the connect action are shown in Figure 4.

```
Action  Create-NAND-gate
{
     Parameter  object : NAND
     Parameter  location : Position
     Pre-condition: "exist(x, DESIGN)"
     Post-condition: "exist(object, NAND)"
}

Action connect
{
     Parameter  gate-1 : GATE
     Parameter  gate-2 : GATE
     {Constraint    "gate-1 ≠ gate-2"}
     Pre-condition:
     "exist(gate-1, GATE)&exist(gate-2, GATE)"
     Post-condition:
     "connected(gate-1, gate2)&connected(gate-2, gate-1)"
}
```

Figure 4 Textual Input for Application Actions

4. CREATING THE MODEL

Currently the application and the interface model are created by parsing in textual inputs. An application model parser reads in descriptions of application actions and generate a structure consisting of actions, parameters, parameter constraints, and pre- and post-conditions. An interface model parser reads in information which links application actions to interface actions. Descriptions of interface actions are pre-defined by real interface designers and stored in the Interface Functions Library using the same action structure as in the application model. Most of these interface actions are already binded to interaction techniques.

Figure 4 shows a flow diagram of how the application and the interface models are created from textual inputs. Figure 5 shows an example of textual inputs which create connections in the interface model.

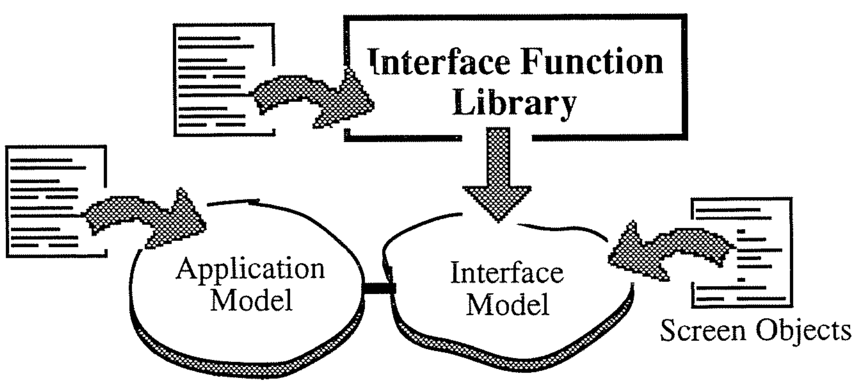


Figure 4 A Diagram Showing Input Files which are the Sources for Creating the Runtime Model

```
ActionLink
{
  ApplicationAction  connect
  InterfaceAction    select-command-icon
  {
      Post-condition: "selected(CONNECT, ACTION)"
       Set {icon, @connect-icon}
  }
  InterfaceAction    select-graphical-object
  {
       Pre-condition: "selected(CONNECT,ACTION)"
       Bind {object, gate-1}
  }
  InterfaceAction   select-graphical-object
       Pre-condition: "selected(CONNECT,ACTION)"
       Bind {object, gate-2}
  }
```

Figure 5 Example of Textual Description of the Binding from *Connect* to Interface Actions

Descriptions of screen objects are parsed into the interface model from a separate text file. This is under the assumption that class definitions of these objects already exist within the runtime environment. Notice here that we cannot allow new classes of objects to be dynamically created due to the C++ restriction. C++ class descriptions of application objects and application-specific interface objects must be compiled into the system before instances of these objects can be created; this process could be automated by making the design process a 2-step process – first, generating C++ class definitions from high-level class descriptions, and second, creating the rest of the model.

From the input shown in Figure 5, the additional application-specific pre- and post-conditions are attached to the interface actions binded to the action for sequencing purposes. Figure 6 depicts an example (not related to inputs in Figure 5) where the interface action c depends on the post-condition of a, and b depends on the post-condition of c. Therefore, to execute the application action ß, the user must perform a, c, and b interface actions.

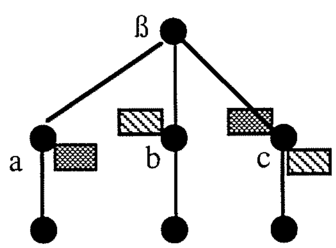


Figure 6 A mapping graph showing condition dependency among interface actions

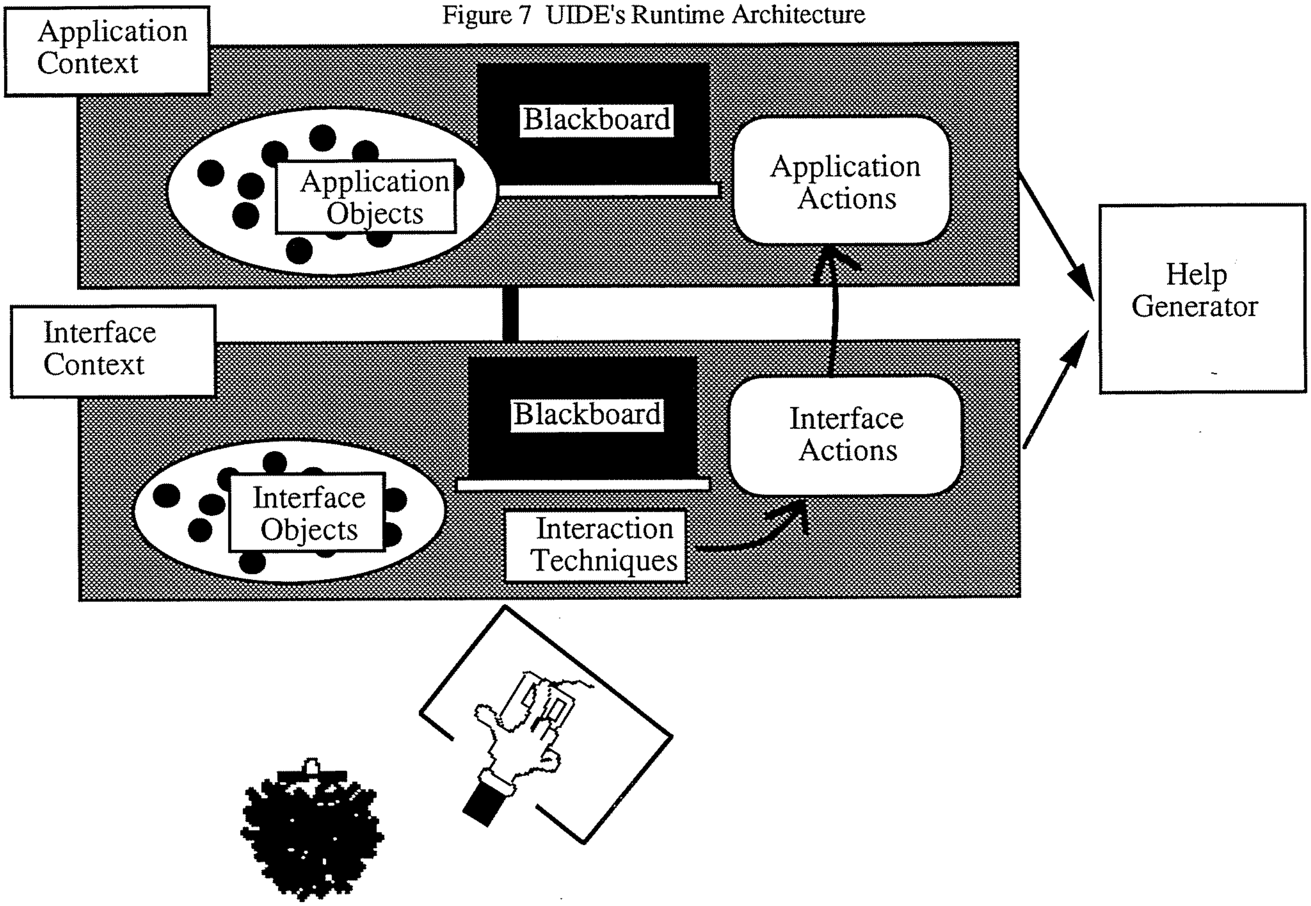
Currently, a designer creates an interface for an application by first describing objects and operations in the application. The designer then chooses various interface functional components which best fit the application by linking them to application actions. This requires quite a bit of understanding of which actions are appropriate for which interface objects, which interface actions are appropriate for which steps of application actions. This process could be automated by having the system suggest alternatives under appropriate guidelines and policies. Graphical, model-based UI tools which facilitate both bottom-up and top-down approaches are also needed to allow designers to start and navigate a design from either end of the design process.

5. THE ARCHITECTURE

UIDE is object-oriented; actions, parameters, objects, pre-conditions, and post-conditions are represented as objects. These objects are linked together through pointers to maintain semantic relationships among related components. We maintain two facets of an object – the declarative facet and the functional facet. So far, we have explained the declarative part of our modeling components, of which information is used for reasoning at runtime. Some of these objects, such as interface objects and interaction techniques, also work as functional objects in the runtime environment. We have kept the declarative side of the UIDE model such as actions, parameters, pre- and post-conditions, rather constant within the past 2 years. The detailed interface components have been modeled after the Cartoonist's knowledge base. The functional side of these components and how they fit together in an architecture, on the other hand, has been revised significantly especially with the change to the new platform.

The architecture of the runtime environment is shown in Figure 7. As you can see from the picture, we maintain the runtime context in 2 parts – one concerning the application program itself, and the other concerning the context of the interface to the application. Within the application context, a list of all application actions together with its related semantic components is maintained. Objects which are created within an application are also maintained in a list. The interface context has a similar structure which maintains a list of interface objects, interface actions, and also interaction techniques. Both application context and interface context maintains a blackboard which contains declarative information concerning various status of the context such as currently selected objects, current action, etc. A blackboard is a list of text-string, each of which represents a predicate.

Figure 7 UIDE's Runtime Architecture



The major deficiency in the C++ environment from the point of view of a model-based user interface architecture is the lack of automatic support for declarative object information and object identification. We require all modeling components in the system to be named, so the names can be used by the reasoning engines in the system. We did not implement a meta-class structure for the knowledge model. Instead, we implemented mechanisms which can traverse the model and retrieve the right information when needed.

Processing Input Information

One way to look at this architecture is to think of the layout of interface objects as a visual interface to the application. This may not be necessarily true in case an interface object is an audio piece. But let's assume a graphical interface for now. When the user performs an action on an interface object, essentially that is handled by an interaction technique which is attached to that particular interface object. Upon completing the technique, the corresponding interface action is informed. Parameter information is also passed along, i.e., a technique for mouse-selecting a graphical object would pass the selected object to the interface action.

Following up on this particular example, the interface action processes the graphical object, derives the application object behind it, and passes it along to the corresponding application action. The application action sees to it that the constraints upon this object are satisfied as specified in parameter constraints, then it deposits the information in the context. At the same time, the interface action gets notified and it processes its post-conditions, the result of which may change the condition in the interface blackboard. The application context then checks what it needs to do next. If all parameters of the current action have been acquired, the application routine associated with this action is then invoked. Upon finishing the application routine, the post-conditions of the action are processed and this often results in changes in the application blackboard.

Application Routines

In order for an application to work with UIDE, we require that each application action defined in the UIDE model have an application routine associated with it. Once UIDE obtains all parameter values for an action, UIDE passes a list of the action's parameters to the application routine. Currently, each application routine is required to know about the list and the parameter structures, and how to take the actual parameter values out of the list for further computation. The control is returned to UIDE once the routine is executed.

The circuit design application used in this paper is written in C++. Its data objects which are dynamically created at runtime are stored in the UIDE's application context once they are created. We required that each object created in the application be registered with the application context. The registration process makes sure that the presentation object associated with this class of object be displayed properly. Currently, if an application is not written in C++, either a set of redundant C++ objects must created in the application context parallel with objects in the actual application, or communication channels must be established to share these data objects.

Automatic Dialog Sequencing

Let's assume from the scenario above that some parameters of the application action are still not specified. Upon realizing that more inputs are needed for an application action, the application context passes control to the interface context which has the interface model for all application actions. The interface actions which are binded to the current application action and whose pre-conditions are met become enabled. Those interface objects which are connected to interaction techniques binded to these enabled interface actions are enabled or made visible. For example, if the next input parameter must be entered through an interface object contained in a dialog box, the interface context handler will pop up the dialog box and enable the corresponding interface object.

So far it is straightforward, assuming that all interface objects start out disabled, and changing conditions in the environment renders them visible and/or enabled. There are times when the designer may want to disable interface objects after they have been enabled. When do we disable interface objects then? On one hand, the designer puts in guarding pre-conditions for interface actions to specify when they should be enabled. These conditions, on the other hand, should render interface objects disabled when they are not satisfied. The table below summarizes UIDE's action upon evaluating pre-conditions of an action with regards to the current status if an interface object. Remember that connections from pre-conditions to interface objects are derived from the semantic connections between interface actions and their interface object parameters.

	Pre-condition **Met**	Pre-conditions **Not Met**
Currently **Enabled**	No Action	Disable
Currently **Disabled**	Enable	No Action

Any interface objects could be effected by the change in the current condition. Evaluating pre-conditions of all interface actions and a la associated interface objects is inefficient. We only re-evaluate pre-conditions of those interface actions which are binded to the current application action, and enable and disable objects associated with this set of actions. A more appropriate scope for evaluation is needed. For example, in a case where a dialog box contains items which are related to other application actions, these objects are potential candidates to be enabled/disabled. In this case, the scope for re-evaluating pre-conditions is broader than one application action.

Notice that the automatic control sequencing depends on the pre- and post-conditions which are associated with interface actions. Pre-conditions which enable an interface object may be those which make sure that certain values exist (i.e., have been entered) prior to interacting with the object. Pre-conditions can also depend on exact values which are previously entered. For example, if exist(x, TEMPERATURE) & status(x, greater-than(100)) is true, a set-alarm-on icon will be enabled. One of our C++ prototypes [10] uses a predicate mechanism to maintain these dependencies. The prototype is based on predicates which are attached directly to objects. This approach has its advantage of removing conditional dependency from the actual code. The same mechanism is being revised to work with the current knowledge model where pre- and post-conditions are attached to actions where they semantically belong. Our claim is, in the current model, semantic predicates need not be removed and re-entered if the designer decides to change interface objects. Also, it will be easier to automate the transfer of predicates for a shift in interface paradigms where a new set of interface actions replaces current interface actions.

Putting in interface-level pre- and post-conditions manually and textually becomes unnatural very quickly. Too many logical connections will quickly exceed the designer's capability to keep track of details while maintaining the big interface picture. A better input mechanism such as a visual interface where dependencies can be specified through examples will be of great help.

5. OTHER BENEFITS FROM THE UIDE'S MODEL-BASED ARCHITECTURE

Automatic Generation of Context-sensitive Help

Another C++ prototype which generates textual help for 2 questions, **why** an interface object is disabled and **how** to enable a disabled object, is reported in [6]. The help system uses pre-conditions which are associated with widgets as a basis for explanations of reasons why a widget is disabled at runtime. It uses a text template associated with each predicate as a basis for generating text. The answer to how a widget can be enabled is generated using a simple planner to derive a series of actions which, when chained together, satisfies the unsatisfied pre-conditions.

Context-sensitive animated presentation of how to perform an action was implemented utilizing the Smalltak-80 version of the knowledge model [21, 22]. The help system, Cartoonist, synthesized the current context and the procedural descriptions of the knowledge model to come up with animation scenarios which explained how to perform actions. A planner was also used to derive scenarios involving additional actions which satisfied conditions which were not true in the current context.

We are currently working on integrating animated help with textual help generation. We are working on three major improvements over our previous prototypes. First, the textual WHY help generation is being revised to work with the full knowledge model. Pre- and post-conditions are now inferred from interface and application actions associated with a disabled object. Second, the textual help on HOW to make an object enabled is now derived from actions in both the application model and the interface model. Animated help is integrated with this help for showing interaction techniques on interface objects, while textual explanations are derived from associated interface and application actions. And lastly, help on HOW to perform an action is derived in the same fashion similar to Cartoonist, with a better planner and a more efficient predicate representation. All these 3 kinds of help generation are automatically generated and will be made available in the same environment.

Recording of User Task Usage Information

Another piece of work currently being developed is using the knowledge model as a framework to record user task usage patterns within UIDE. As described in Section 5 on the way input information is processed in the current architecture, it is easily seen that each time an interaction technique is completed, each time an interface action is completed, and each time an application action is completed, UIDE is discretely aware of these occasions. We take advantage of the fact to record user completions of actions at different levels of abstractions. Following are different kinds of information available through the task recording mechanism.

Statistical History of Interactions: Upon a successful completion of each application action, the counter for this action is incremented. All interface actions and interaction techniques which are used for this particular invocation are considered completed and their corresponding counters are incremented.

Two kinds of statistics could be recorded with regards to interface actions and interaction techniques. One is their usage in relation to the actions to which they are binded. This recording allows the system to have information such as "has this technique been used for this action by this user?" or "which interface action is used more often to invoke this command?" The other statistic is global across actions in an application. At any point in time, this statistic would provide information such as how many times this interaction techniques have been used in this application, or whether most commands are invoked through accelerator keys, etc. Since interface actions and interaction techniques can be used in various applications, this information could be useful to understand how well users will handle a new application with interface elements they are already familiar with.

Chronological History of Interactions: Chronological history can be recorded rather straightforwardly in UIDE. Each action, once performed, can be recorded with associated parameter values chosen. Timestamps can be recorded. Interface actions and interaction techniques can also be recorded. The chronological recording would feed in research prototypes for intent recognition work such as Edwards [1], EAGER [4], SINIX [15] and LEXITUS [13].

Chronological recording of help requests can be recorded separately. It is merely a list of actions for which help is requested and timestamps when the requests occur.

6. IMPLEMENTATION

The current runtime architecture UIDE is implemented in C++ using the OLIT widget set as a front-end interface and runs on Sun's X Server. 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 ported to C++. Generation of WHY explanations has been developed separately in C++ and is now integrated with the current UIDE.

6. FUTURE DIRECTIONS OF THE ARCHITECTURE

Creating the model in UIDE using textual input is still very cumbersome, and the connection from the model to the actual interface are not easily seen. Visual interface has its strength in caching some of the information which would otherwise have to be memorized on the screen. Direct manipulation user interface builders such as DevGuide [24] and Prototyper [3] make designing an interface one step easier. Some direct manipulation tools such as DialogEditor [2] and Peridot [16] 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. These tools are not model-based tools and provide no assistance beyond laying out interface objects on the screen. VisualWorks [26] is close to what we would like to see; it partially creates the underlying model when an interface object is created. This makes creating an interface in the MVC paradigm possible without having to understand it first.

A necessary future direction is to come up with visual tools which would make the tie between the application model to an interface much easier to visualize, and help designers come up with a new application model with ease. Work in this direction has started and is reported in [9]. Figure 8 shows where various graphical tools can be used to enhance the design process. Ideally, if these tools are active at runtime as well, they perhaps can be use as debugging tool for both the model and the interface itself.

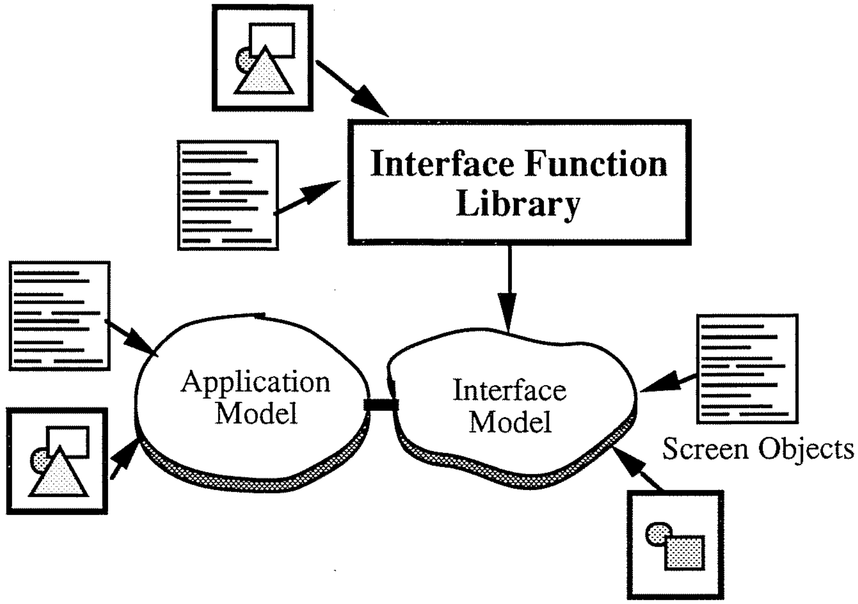


Figure 8 Places Where Visual Interface Design Tools can Help Create the Model

Currently, more advanced tools such as Siemens's S/X tools which provide sophisticated server-client user interface support have begun to realize the need for an underlying model for intelligent user interface support. Since the UIDE model is independent of toolkits, we realize a practical approach of separating the front-end interface from the application model. This will allow the same application to have interfaces implemented in different toolkits. We plan to add a protocol handler as an communication intermediary between running interfaces and the model. Messages from the model would be those which control the sequencing of an interface such as displaying, undisplaying, or disabling objects. Messages from the front-end interface would be those which inform the model what the user has done, i.e., an interaction technique is completed on an object. With the current model structure, multiple interface models are also possible to support different interface paradigms for different groups of users. Figure 9 illustrates how the UIDE architecture should look like in the near future.

7. CONCLUSIONS

A model-based user interface architecture embeds meta-information about an application and its interface in the runtime architecture. Taking advantage of this declarative knowledge, the system can provide intelligent user interface support. UIDE is an example of a model-based user interface architecture. We have described in this paper how we model the application, the interface, and how they connect to form an executable interface. We also described how declarative model information enabled intelligent support such as automatic generation of context-sensitive help and recording of user command usage pattern. These two benefits open doors for future research such as adaptive interfaces, adaptive help, and perhaps others as well.

In addition to these benefits, the modeling components in UIDE can be used as a set of primitives to program an interface. These primitives take away the burden of programming an interface; they allow designers to think in terms of, but not necessarily restricted to, abstract functions in the application domain. A visual interface tool without a model often makes designers think too much of physical details and layout of an interface, and get distracted from the actual tasks an interface is designed for. A model-based user interface design seems to bring out a better, process-oriented design process, while the benefits which it enables at runtime are more than convincing that a model should be part of the user interface archiecture in the future.

ACKNOWLEDGEMENTS

This work has been supported by the Siemens Corporate R&D System Ergonomics and Interaction group of Siemens Central Research Laboratory, Munich, Germany, the Human Interface Technology Group of Sun Microsystems through their Collaborative Research Program, and Digital Corporation. 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, Mark Gray, Srdjan Kovacevic, and Ray Johnson, and our visiting researcher from Siemens, Thomas Kuhme. 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.

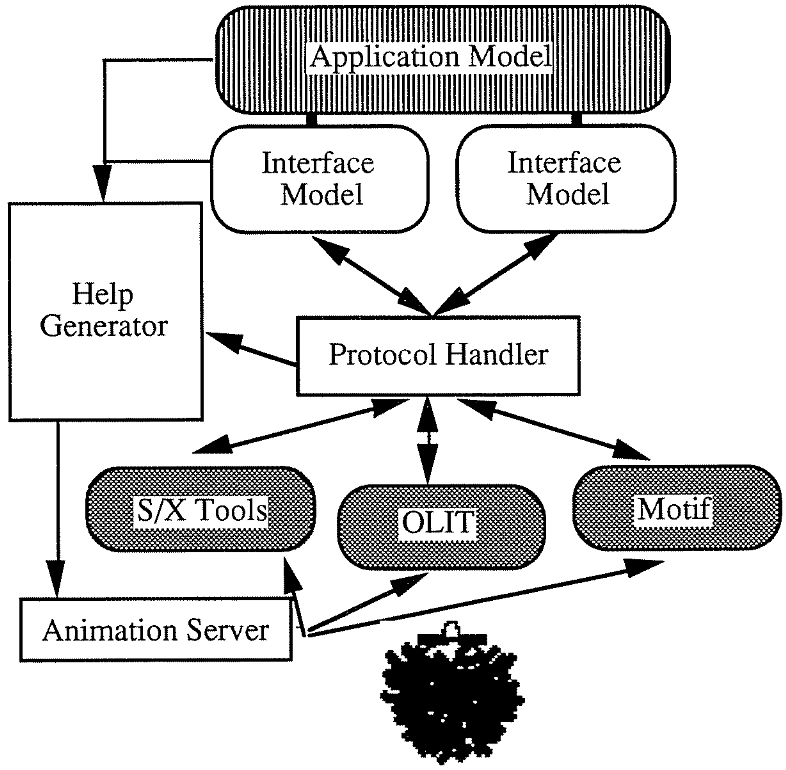


Figure 9 Future UIDE Architecture where Various Toolkits Could be Used as UIDE Front-Ends

REFERENCES

1. Bos, E. Some Virtues and Limitations of Action Inferring Interfaces. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*. November 1992, 79-88.
2. 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.
3. Cossey, G. *Prototyper*. SmetherBarnes, Portland, Oregon, 1989.
4. Cypher, A. Eager: Programming Repetitive Tasks by Example. In *Proceedings of Human Factors in Computing Systems, CHI'91*. 1991, 33-39.
5. 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.
6. de Graaff, J.J.; P. Sukaviriya, and A.P.G. van der Mast. Automatic Generation of Context-sensitive Textual Help. Submitted to UIST'93.
7. 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.
8. 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.
9. Frank, M. and J,D. Foley. Model-based User Interface Design By Example and By Interview. Submitted to UIST'93.
10. 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.
11. Goldberg, A. Information Models, Views, and Controllers. *Dr. Dobb's Journal of Software Tools*, 15(7) : 54-61, July 1990.
12. 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.
13. Hoppe, H. U. Intelligent User Support Based on Task Models. *Adaptive User Interfaces: Principles and Practice*. Amsterdam : North Holland Elsevier, 1993.
14. Inference Corporation. *ART Reference Manual*. Inference Corporation, Los Angeles, CA, 1987.

15. Kemke, C. Representation of Domain Knowledge in an Intelligent Help System. In *Proceedings INTERACT'87, 2nd IFIP Conference on Human-Computer Interaction*. 1987 215-220.
16. Myers, B. Creating Interaction Techniques by Demonstrations. *IEEE Transactions on Computer Graphics and Applications* 7 (September 1987): 51-60.
17. Neches, R.; J.D. Foley; P. Szekeley; P. Sukaviriya; P. Luo; S. Kovacevic; and S. Hudson. Knowledgeable Development Environments Using Shared Design Models. *Proceedings of Intelligent Interfaces Workshop*, Orlando, Florida, January 4-7, 1993.
18. Olsen, D. MIKE: The Menu Interaction Kontrol Environment. *ACM Transactions on Graphics* 5,4 (1986): 318-344.
19. Olsen, D. A Programming Language Basis for User Interface Management. In *Proceedings of Human Factors in Computing Systems, CHI'89*. May 1989, 171-176.
20. 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.
21. 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.
22. Sukaviriya, P. *Automatic Generation of Context-sensitive Animated Help*. A Ds.C. Dissertation, George Washington University, 1991.
23. Sukaviriya, P.; J.D. Foley, and T. Griffith. A Second Generation User Interface Design Environment: The Model and the Runtime Architecture. *Proceedings of Human Factors in Computing Systems, INTERCHI'93*. April 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. 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.
26. VisualWorks. ParcPlace Systems. Mountain View, California