Mapping GUIs to Auditory Interfaces

Elizabeth D. Mynatt and W. Keith Edwards

Graphics, Visualization, and Usability Center
College of Computing
Georgia Institute of Technology
Atlanta, GA 3032-0280
beth@cc.gatech.edu, keith@cc.gatech.edu

ABSTRACT

This paper describes work to provide mappings between X-based graphical interfaces and auditory interfaces. In our system, dubbed Mercator, this mapping is transparent to applications. The primary motivation for this work is to provide accessibility to graphical applications for users who are blind or visually impaired. We describe the design of an auditory interface which simulates many of the features of graphical interfaces. We then describe the architecture we have built to model and transform graphical interfaces. Finally, we conclude with some indications of future research for improving our translation mechanisms and for creating an auditory "desktop" environment.

KEYWORDS: Auditory interfaces, GUIs, X, visual impairment, multimodal interfaces.

INTRODUCTION

The goal of human-computer interfaces is to provide a communication pathway between computer software and human users. The history of human-computer interfaces can be interpreted as the struggle to provide more meaningful and efficient communication between computers and humans. One important breakthrough in HCI was the development of graphical user interfaces. These interfaces provide graphical representations for system objects such as disks and files, interface objects such as buttons and scrollbars, and computing concepts such as multi-tasking. Unfortunately, these graphical user interfaces, or GUIs, have disenfranchised a percentage of the computing population. Presently, graphical user interfaces are all but completely inaccessible for computer users who are blind or severely visually-disabled [BBV90][Bux86][Yor89]. This critical problem has been recognized and addressed in recent legislation (Title 508 of the Rehabilitation Act of 1986, 1990 Americans with Disabilities Act) which mandates that computer suppliers ensure the accessibility of their systems and that employers must provide accessible equipment [Lad88].

Our work on this project began with a simple question, how could we provide access to X Windows applications for blind computer users. Historically, blind computer users had little trouble accessing standard ASCII terminals. The line-oriented textual output displayed on the screen was stored in the computer's framebuffer. An access program could simply copy the contents of the framebuffer to a speech synthesizer, a Braille terminal or a Braille printer. Conversely, the contents of the framebuffer for a graphical interface are simple pixel values. To provide access to GUIs, it is necessary to intercept application output before it reaches the screen. This intercepted application output becomes the basis for an off-screen model of the application interface.The information in the off-screen model is then used to create alternative, accessible interfaces.

The goal of this work, called the Mercator[1] Project, is to provide *transparent* access to X Windows applications for computer users who are blind or severely visually-impaired [ME92]. In order to achieve this goal, we needed to solve two major problems. First, in order to provide transparent access to applications, we needed to build a framework which would allow us to monitor, model and translate graphical interfaces of X Windows applications without modifying the applications. Second, given these application models, we needed to develop a methodology for translating graphical interfaces into nonvisual interfaces. This methodology is essentially the implementation of a *hear-and-feel* standard for Mercator interfaces. Like a look-and-feel standard for graphical interfaces, a hear-and feel standard provides a systematic presentation of nonvisual interfaces across applications.

In this paper, we describe the steps we have taken to solve these two problems. In the following section, we describe the design for the Mercator interface. We introduce the concept of audio GUIs and the abstract components of auditory interfaces. We also detail some of the techniques we are using to convey a range of interface attribute information via the auditory channel.

The second half of the paper describes the architecture we have constructed to provide this interface transparently for X

1. Named for Gerhardus Mercator, a cartographer who devised a way of projecting the spherical Earth's surface onto a flat surface with straight-line bearings. The Mercator Projection is a mapping between a three-dimensional presentation and a two-dimensional presentation of the same information. The Mercator Environment provides a mapping from a two-dimensional graphical display to a three-dimensional auditory display of the same user interface.

applications. We detail the requirements and goals of the system, the individual components of the architecture, and how those components interoperate to provide a translation of a graphical interface into an auditory interface

AUDIO GUIs

The primary design question to be addressed in this work is, given a model for a graphical application interface, what corresponding interface do we present for blind computer users. In this portion of the paper, we discuss the major design considerations for these interfaces. We then describe the presentation of common interface objects such as buttons, windows, and menus, and detail the navigation paradigm for Mercator interfaces. We conclude by discussing the inherent advantages of a hear-and-feel standard.

Design Considerations

There are several design decisions we had to make when constructing our nonvisual interface. One consideration is which nonvisual interface modality to use. The obvious choices are auditory and tactile. We are currently focusing on auditory interfaces for several reasons:

- Research in auditory interfaces (Gaver, Bly, Blattner et al) has proven that complex auditory interfaces are usable [BGB91].
- Due to people's ability to monitor multiple auditory signals (the cocktail party effect [Che53]) and through the use of spatialized sound [WWF88], we can support many of the advantages of GUIs such as spatial organization and access to multiple sources of information.
- Tactile output devices are generally more passive than auditory output. It is more difficult to get the users' attention with tactile output.
- For the most part, audio hardware is more common and cheaper than hardware to support tactile displays.
- A significant portion of people who are blind also suffer from diabetes which reduces their sensitivity to tactile stimuli [HTAP90].

Nevertheless our system will eventually have a tactile component as well. For examplle, since speech synthesizers are notoriously bad at reading source code, we will provide a Braille terminal as an alternate means for presenting textual information.

A second major design question for building access systems for visually-impaired users is deciding the degree to which the new system will mimic the existing visual interface. At one extreme the system can model every aspect of the visual interface. For example, in Berkeley System's Outspoken,™ which provides access to the Macintosh, visually-impaired users use a mouse to search the Macintosh screen [Van89]. When the mouse moves over an interface object, Outspoken reads the label for the object. In these systems, visually-impaired users must contend with several characteristics of graphical systems which may be undesirable in an auditory presentation, such as mouse navigation and occluded windows.

At the other extreme, access systems can provide a completely different interface which bears little to no resemblance to the existing visual interface. For example, a menu-based graphical interface could be transformed into an auditory command line interface.

Both approaches have advantages and disadvantages. The goal of the first approach is to ensure compatibility between different interfaces for the same application. This compatibility is necessary to support collaboration between sighted and non-sighted users. Yet if these systems are too visually-based they often fail to model the inherent advantages of graphical user interfaces such as the ability to work with multiple sources of information simultaneously. The second approach attempts to produce auditory interfaces which are best suited to their medium. [Edw89]

We believe that there are many features of graphical interfaces which *do not need* to be modeled in an auditory interface. Many of these features are artifacts of the relatively small two-dimensional display surfaces typically available to GUIs and do not add richness to the interaction in an auditory domain. If we consider the GUI screen to be in many regards a limitation, rather than something to be modeled exactly, then we are free to use the characteristics of auditory presentation which make it more desirable in some ways than graphical presentation.

We have chosen a compromise between the two approaches outlined above. To ensure compatibility between visual and nonvisual interfaces, we are translating the interface at the level of the interface components. For example, if the visual interface presents menus, dialog boxes, and push buttons, then the corresponding auditory interface will also present menus, dialog boxes and push buttons. Only the presentation of the interface objects will vary.

By performing the translation at the level of interface objects, rather than at a pixel-by-pixel level (like Outspoken) we can escape from some of the limitations of modeling the graphical interface exactly. We only model the structural features of the application interface, rather than its pixel representation on screen.

Interface Components

Graphical user interfaces are made up of a variety of interface components such as windows, buttons, menus and so on. In X Windows applications, these components roughly correspond to widgets. There does not always exist a one-to-one mapping between graphical interface components and X widgets. For example, a menu is made up of many widgets including lists, shells (a type of container), and several types of buttons.

Mercator provides auditory interface objects which mimic some of the attributes of graphical interface objects. In Mercator, we call the objects in our auditory presentation Audi-

TABLE 1. Using Filtears to convey AIC attributes.

Attribute	AIC	Filtear	Description
selected	all buttons	animation	Produces a more lively sound by accenting frequency variations
unavailable	all buttons	muffled	A low pass filter produces a duller sound
has sub-menu	menu buttons	inflection	Adding an upward inflection at the end of an auditory icon suggests more information
relative location	lists, menus	pitch	Map frequency (pitch) to relative location (high to low)
complexity	containers	pitch, reverberation	Map frequency and reverberation to complexity. Low to large, complex AICs and high to small, simple AICs

tory Interface Components, or AICs. The translation from graphical interface components to AICs occurs at the widget level. As with graphical interface components, there is not always a one-to-one mapping between X widgets and AICs. AICs may also be composed of many widgets. Additionally, many visual aspects of widgets need not be modeled in AICs. For example, many widgets serve only to control screen layout of sub-widgets. In an environment where there is no screen, there is no reason to model a widget which performs screen layout. For many widgets there *will* be a one-to-one mapping to AICs. As an example, push buttons (interface objects which perform some single function when activated) exist in both interface domains. In other cases, many widgets may map to a single AIC. For example, a text window with scrollbars may map to one text display AIC. Scrollbars exist largely because of the need to display a large amount of text in a limited area. A text display AIC may have its own interaction techniques for scanning text.

There are two types of information to convey for each AIC: the type of the AIC and the various attributes associated with the AIC. In our system, the type of the AIC is conveyed with an auditory icon. Auditory icons are sounds which are designed to trigger associations with everyday objects, just as graphical icons resemble everyday objects [Gav89]. This mapping is easy for interface components such as trashcan icons but is less straight-forward for components such as menus and dialog boxes, which are abstract notions and have no innate sound associated with them. As an example of some of our auditory icons, touching a window sounds like tapping on a glass pane, searching through a menu creates a series of shutter sounds, a variety of push button sounds are used for radio buttons, toggle buttons, and generic push button AICs, and a touching a text field sounds like a old fashioned typewriter.

AICs can have many defining attributes. Most AICs have text labels which can be read by a speech synthesizer upon request. Many attributes can be conveyed by employing so-called *filtears* to the auditory icon for that AIC. Filtears provide a just-noticeable, systematic manipulation of an auditory signal to convey information[LPC90][LC91]. Table 1 details how filtears are used to convey some AIC attributes.

Navigation

The navigation paradigm for Mercator interfaces must support two main activities. First, it must allow the user to quickly "scan" the interface in the same way as sighted users visually scan a graphical interface. Second, it must allow the user to operate on the interface objects, push buttons, enter text and so on.

In order to support both of these activities, the user must be able to quickly move through the interface in a structured manner. Standard mouse navigation is unsuitable since the granularity of the movement is in terms of graphic pixels. Auditory navigation should have a much larger granularity where each movement positions the user at a different auditory interface object. To support navigation from one AIC to another, we map the user interface into a tree structure which breaks the user interface down into smaller and smaller AICs. This tree structure is related to application's widget hierarchy but there is not a one-to-one mapping between the widget hierarchy and the interface tree structure. As discussed earlier, there is sometimes a many-to-one mapping between widgets and AICs. Additionally, an AIC may conceptually be a child of another AIC but the widgets corresponding to these AICs may be unrelated. For example, a push button may cause a dialog box to appear. These AICs are related (the dialog box is a child of the push button) but the widget structure does not reflect the same relationship. Figure 1 shows a screen-shot of the graphical interface for xmh, an X-based mail application. Figures 2a and 2b show a portion of the xmh widget hierarchy and the corresponding interface tree structure, respectively.

To navigate the user interface, the user simply traverses the interface tree structure. Currently the numeric keypad is used to control navigation. Small jumps in the tree structure are controlled with the arrow keys. Other keys can be mapped to make large jumps in the tree structure. For example, one key on the numeric keypad moves the user to the top of the tree structure. It is worth noting that existing application keyboard short-cuts should work within this structure as well.

Navigating the interface via these control mechanisms does not cause any actions to occur except making new AICs "visible." To cause a selection action to occur, the user must hit the Enter key while on that object. This separation of con-

Figure 1: An Example X Application (XMH)

trol and navigation allows the user to safely scan the interface without activating interface controls. Table 2 summarizes the navigation controls.

Hear and Feel

Just as graphical interfaces use "look-and-feel" standards to ensure that, for example, all buttons look and behave in similar manners, we are using what may be called "hear-and-feel" standards to ensure that consistency is maintained across the interface. AICs of the same type are presented consistently across application. This consistency promotes usability in our environment.

ARCHITECTURE

We shall now discuss the architecture we have developed for the Mercator Applications Manager, the system which implements the interface described above.

There are basically three goals which must be met to generate an auditory interface from an X application given our design constraints (application transparency, and so on). First, we must be able to capture high-level, semantically meaningful information from X applications. By semantically meaningful we mean that the information we get must reflect the structural organization of the interface, rather than just its pixelated representation on-screen. Second, we must create a semantic model based on the information retrieved. And third, once we have retrieved this information from the application and stored it in our off-screen model, we must be able to present the structure of the application in some meaningful way, and allow the user to interact with our new representation of the interface.

It should be noted that while our current interface uses an auditory presentation, the facilities we have built could just as easily present a different 2D graphical interface for a given application, map it into a 3D interface, or so on, given the proper mapping rules.

We shall discuss the design and implementation of the various system components in the next several sections. Figure 3 gives a bird's-eye overview of the architecture, to which we will refer during our discussion.

Information Retrieval

To be able to translate a GUI into another representation we must first be able to retrieve information about the interface from the application. Since one of the requirements of the

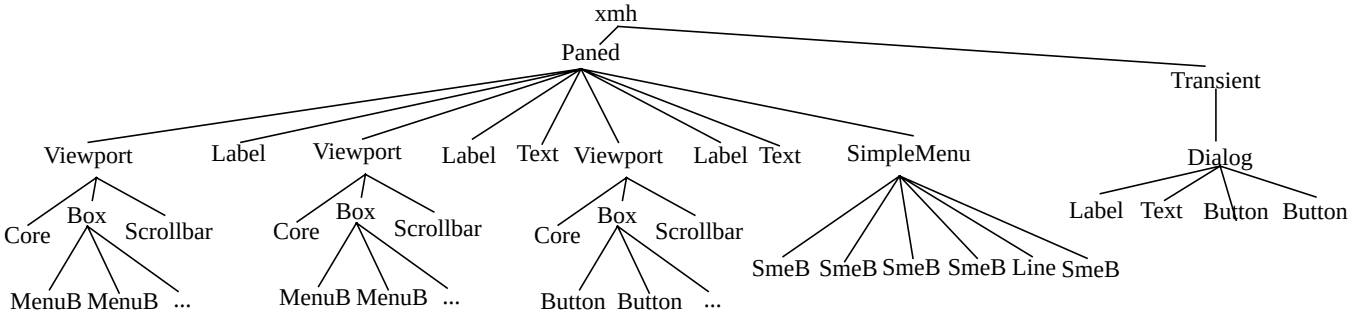

Figure 2a: XMH Widget Hierarchy

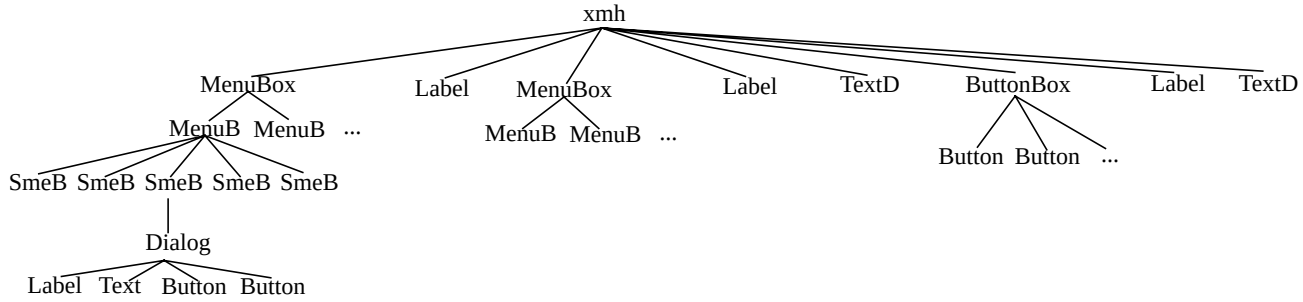

Figure 2b: Corresponding XMH Auditory Interface Component Hierarchy

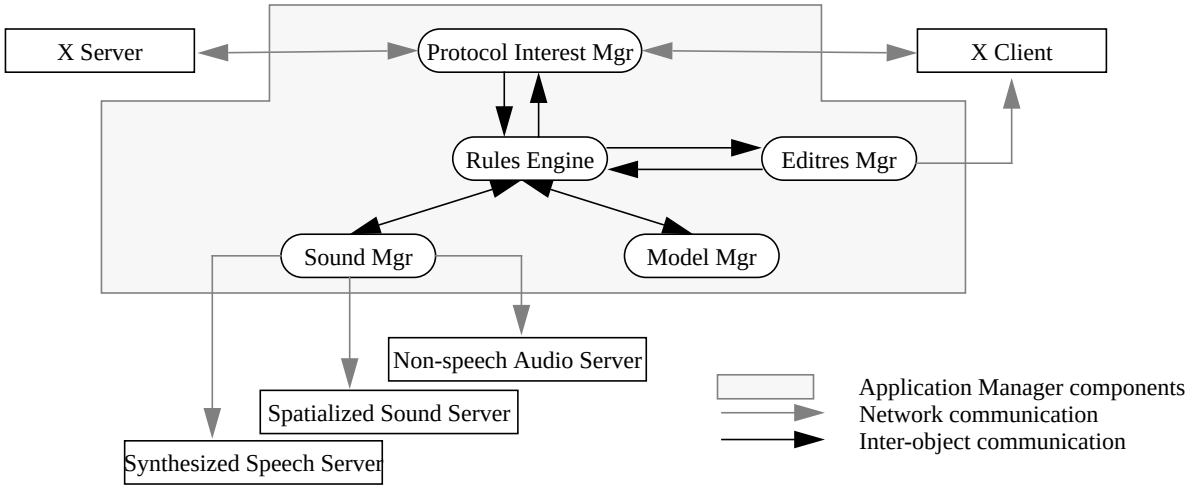

Figure 3: Application Manager Overview

Mercator project is that the system must be able to provide access to existing applications, the Application Manager must be able to retrieve information from unmodified X applications *as they are running*.

Thus, while simply modifying an X toolkit would have given us a testbed for research into providing auditory output from X applications, it would not have been a general solution to the problem of providing access to already-existing X applications. Our system uses two approaches to retrieve information from running applications: client-server monitoring, and the Editres protocol, discussed below.

Client-Server Monitoring. The architecture of X allows us to "tap" the client-server connection and eavesdrop on the communication between the application and X [Sch87]. Thus, we can know when text or graphics are drawn, and when events (such as keypresses) take place. The Application Manager component which accomplishes this monitoring is called the Protocol Interest Manager, or PIM. The PIM lies between the client and server and processes the communication between them (see Figure 3).

Monitoring the connection between the client and the server allows us to access the low-level information describing *what* is on the user's screen. Unfortunately, it does not tell us *why* something is on a user's screen. This lack of information is because the constructs in the X protocol are very low-level and do not embody much information about application structure or behavior.

For example, the protocol does not express notions such as "draw a button with the label 'Quit.'" Instead, the protocol passes messages on the level of "draw a rectangle of the following dimensions at the following X,Y coordinates," and "render the text 'Quit' at the following X,Y coordinates." It is difficult for any system to know from simply monitoring the protocol stream whether the rectangle specified in the protocol is meant to be a button, a graphic image on a drawing canvas, a menu, or any other interface component.

Thus, simply reading the X protocol as it is passed between a client and the server is insufficient to provide a good semantically-meaningful translation of a graphical interface. For this reason, we make use of the Editres protocol described below.

Editres. Release 5 of the X Window System presented a new protocol called Editres designed primarily to allow easy customization of X applications [Pet91]. In the programming model of the X Toolkit Intrinsics layer, applications are built out of user interface objects called *widgets* (such as scrollbars and push buttons). Widgets have named variable data called *resources* associated with them. Resources govern the appearance and behavior of the widgets in an application. Examples of data stored in resources include the currently-displayed text in a widget, the color of a widget, information about whether the data displayed in the widget is editable, layout policies, and so on.

The Editres protocol defines a method for querying a running application and retrieving from it information about the hierarchy of widgets which make up the application. Additionally, this protocol makes it possible to query individual widgets about information such as their geometry (X,Y coordinates, width, and height), retrieve resource values (with a slight modification, see *Caveats*), and to set the values of individual resources associated with each widget. In our sys-

TABLE 2. Summary of Navigation Controls

Key	Action
8/up arrow	Move to parent
2/down arrow	Move to first child
6/right arrow	Move to right sibling
4/left arrow	Move to left sibling
0/Ins	Move to top of tree
Enter	Activate selection action
= / * -	Can be mapped to other movements

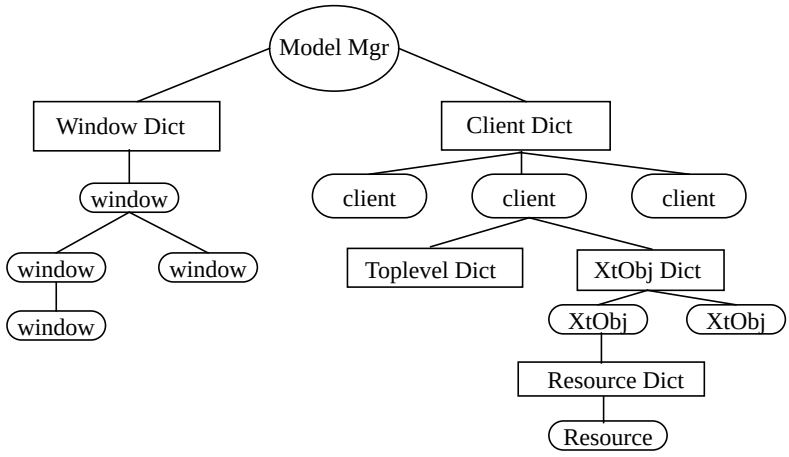

Figure 4: Interface Modeling Data Structures

tem, the Editres Manager sends and receives protocol messages to and from client applications (see Figure 3).

The Editres protocol allows us to determine the structural components which make up an application. By combining the information retrieved from Editres and the information retrieved from client-server communication monitoring, we can build a detailed picture of an application running in our system. To refer back to the previous example of the button, when we detect a request to draw "Quit" on the screen, we can use Editres-derived information to determine which interface component is being rendered.

Sufficiency of Techniques. These two techniques, client-server monitoring and Editres, give us access to a great deal of information about X applications. We have available to us all of the user-generated input events, all of the low-level application-generated output requests, and a great deal of structural and high-level semantic information about the application interface. While this amount of information is significant, it is not complete: we do not have available to us anything that is not expressed in terms of either the X protocol, or widgets and resources. Thus, information and behavior which is directly coded into an application and not built in terms of widgets or resources is unavailable. The degree to which this lack of total knowledge will affect application accessibility is unclear at this point. Based on our initial results, it seems promising that we can effectively model enough aspects of a wide range of applications to the point of making them accessible.

Application Modeling

Based on the information we have retrieved from an application, we must be able to synthesize a coherent model of the application interface. This model must include structural information (such as the hierarchy of objects which comprise the application's interface, and the relations between objects), semantic information (the types of different objects in the interface), appearance attributes ("the text in this window is in boldface"), and behavioral attributes ("clicking this button causes a dialog box to pop up").

The modeling techniques must be sufficient to represent any application which could be run in our environment. The only way to satisfy this requirement is to have our off-screen model mimic many of the structural attributes inherent in X applications. The notion of the window is the lowest common structural denominator for X applications. Windows are represented in the X protocol and thus we are guaranteed that at a minimum we can represent an application's window structure.

The problem with using this approach alone is that, like much of the X protocol, windows themselves are too low-level to be very meaningful. Thus, we also need to maintain information retrieved via Editres: information on the structural components of the application, and the attributes of those components.

Figure 4 is a diagram of the data structure we are using to model the interfaces of programs running under the Application Manager. This data structure maintains both the low-level protocol-related information, and the higher-level Editres-related information. The lines in the diagram represent structural connections, not communication pathways. Note that the Model Manager object is the same object shown in Figure 3; here we have expanded the detail to show the sub-objects associated with the Model Manager.

The Model Manager is responsible for maintaining the off-screen models of all applications running in the environment. The Model Manager keeps two dictionaries to track application information. The first is a dictionary of all windows present in the system. This dictionary maintains a one-to-one mapping of the current X window hierarchy. The dictionary values are Mercator Window objects, which store known attributes about particular windows.

The second dictionary is the Client Dictionary. This dictionary contains Mercator Client objects which represent per-application information. Every time a new application is started, a new Client object is instantiated and placed into the Client Dictionary.

Client objects maintain information about running applications: the application name, current state, and other information. In addition, the Client objects maintain a representation of the structural layout of the application. Each Client object maintains a tree of Xt Objects (widgets) which reflect the widget organization in the actual application. Each Xt Object keeps information about its name and class (such as MenuBox or PushButton), and also keeps a dictionary of resource information.

The other major data structure maintained by Client objects is a dictionary of all toplevel windows in the application. Many applications create several toplevel windows and it is often useful to be able to quickly determine what the top nodes are in an application's window hierarchy. This information is maintained by the toplevel dictionary.

All of the data structures maintained by the Model Manager are multiply keyed, so it is easy to determine the window (or windows) which correspond to a given widget. Similarly, given a window it is easy to determine the widget which corresponds to the window.

Keeping information cached in the Application Manager itself reduces the amount of Editres and X protocol traffic we must generate to retrieve information about the interface and thus can provide performance improvements.

This technique of application modeling gives us the power to represent any X interface in several forms. Our modeling scheme provides us with a means to quickly determine the structural objects which are referred to by X protocol requests. Based on the structural model of the interface maintained by the Model Manager we are able to translate that model to an auditory representation.

Interface Presentation and User Input

Once we have retrieved information from the running application, organized it into a coherent structure, and stored it in the off-screen model, we must still present that interface structure to the user in a meaningful way. Further, to facilitate experimentation and user customization, the system must allow easy modification of the interface.

Rules Engine. We are using a translation system which takes as input the state and structure of the interface we are modeling, and produces auditory output. This translation takes place in the Rules Engine (see Figure 3), which is the heart of the Application Manager. It is the Rules Engine which conceptually implements AICs. From a high-level standpoint, it has two primary functions: presenting the interface as described by the Model Manager to the user, and processing user input to both the application and the Application Manager itself.

The Rules Engine is driven asynchronously by the Protocol Interest Manager, which taps the connection to and from the X server. The Rules Engine informs the PIM of patterns in the X protocol which should cause control be passed to the Rules Engine (that is, the Rules Engine expresses a "protocol interest," hence the name of the Protocol Interest Manager). When protocol events or requests occur which match a specified pattern, control is passed to the Rules Engine which is notified of the condition which caused it to awaken. It may then examine the state of the protocol, generate Editres traffic, or query the Model Manager for current client status. From these input sources, the Rules Engine may decide to generate output to the user.

The facilities available to the Rules Engine are quite complex. The Engine can specify that the X protocol stream be effectively stalled--X protocol packets are queued by the Protocol Interest Manager and not delivered until some later point. This facility can be useful in preventing deadlocks (see the section, *Inter-Object Communication*). Furthermore, the Rules Engine can actually cause new protocol packets to be introduced into the protocol stream. When packets are inserted, the Protocol Interest Manager will rewrite later packets to ensure that protocol sequence numbers are kept consistent. Insertion of protocol packets is done to generate controlling events to the original application based on actions taken in the new interface. Basically, through the use of the Protocol Interest Manager, the Rules Engine has complete control over the X protocol stream.

The Rules Engine can also direct the Editres Manager to query applications and collect replies via Editres. These actions are taken to update the off-screen model of the interface's state. The information returned from applications is stored in the Model Manager and also returned to the Rules Engine. The Rules Engine can query the Model Manager and update any information present there.

Translation Rules and Templates. All of the actions taken by the Rules Engine are dictated by a set of translation rules. It is the translation rules which actually create the notion of AICs from the information available to the Rules Engine. Currently these rules are expressed in a stylized C++ predicate/action notation by the system implementors. In the future we will provide an externalized rules representation which will be read in and parsed by the Application Manager at start-up time. This representation will allow both the developers and users of the system to easily customize the interface.

To obviate the need to install a large number of rules to deal with different output semantics for widgets on a per-instance basis, we have developed the notion of *rule templates*. Rule templates are sets of rules which are generated and installed automatically whenever a widget of a given class is created by the original interface.

For example, the rule template for widgets of the PushButton class may provide a set of rules to be fired whenever the PushButton is activated, desensitized, destroyed, and so forth. When the Rules Engine detects that a PushButton widget has been created, it generates and installs the rules specified in the rule template for this widget class. Thus, a set of routines will automatically be associated with the push button which govern its interaction in the new interface. Rule

templates provide a mechanism for ensuring standard behavior across classes of widgets with little work.

Rule list traversal is quite fast: predicates are preprocessed and organized into hash tables based on the event types which can cause the predicates to return True. Thus, when a particular event occurs, the system does not need to test the predicates of all rules, just the ones that have the event as a condition.

Output. Output to the user is generated via the Rules Engine. When fired, rules can invoke methods on the Sound Manager object (see Figure 3) which generate auditory output. The Sound Manager provides a single programmatic interface to the various sound servers which run in our environment. Each of these servers regulates access to the workstation sound resources and allows access to sound hardware located on other machines across the network.

The first server is responsible for speech synthesis. The second provides digitized (mostly non-speech) audio, and controls access to workstation sound resources. The third server is responsible for producing high-quality spatialized sound. This server architecture allows multiple workstations on a network running Mercator to share audio hardware. The *Status* section gives some more details on these servers.

Input. In addition to representing the interface of the application in the new modality, the Application Manager also has the task of processing user input to both the application and to the Application Manager itself. In essence, the Application Manager must *run* the existing application from user input to the new interface. Given user input into the system and the user's current context, the Application Manager must generate X events to the application which correspond to input to the new interface to cause the application to perform the desired actions.

Our current implementation supports only the mouse and the keyboard as input devices, although we do not use the mouse because it does not seem to be an appropriate navigational device for nonsighted users. Since under X the keyboard normally causes KeyPress events to be generated from the server to the client, the PIM is already in a position to intercept any keystrokes made by the user and process them.

The PIM, under the direction of the Rules Engine, decides whether keystrokes should be routed to the application (that is, passed through the PIM unmodified), or whether they are meant as control input to the Application Manager itself. Currently, the Application Manager grabs the numeric keypad and assumes keystrokes made on it are Application Manager controls. Other keystrokes are routed to the widget designated as the current input receptor.

The current input receptor is basically maintained as a pointer into the Model Manager data structures which defines where the user currently "is." Any typed data will be sent to this widget. When other actions are made by the user (such as a selection), the Rules Engine will generate mouse events to the current widget to initiate the action in the application.

Inter-Object Communication

There are some interesting potential deadlock conditions which we have had to take care to avoid in the Application Manager architecture. Since the entire system is driven by the Protocol Interest Manager, the thread of control must reside within the PIM when the Application Manager is "inactive."

Thus, whenever rule execution terminates, control is passed back to the PIM where the system blocks until either (1) some X request or event traffic is generated, or (2) some other user input takes place which the PIM has been instructed to monitor. Control must be returned to the PIM because when the PIM is not executing, the X protocol stream is effectively stalled.

This blocking behavior can cause several problems. The most common problem is in Editres handling. Editres requests are asynchronous. This means that the Application Manager transmits an Editres query and then, at some unspecified point in the future, the application returns the result of the query. The problem is that the Editres protocol is based on the X protocol, and thus must proceed through the PIM. While in Figure 3 we have shown the Editres Manager communicating directly with the client, this link is a conceptual connection only. In reality, the Editres Manager must establish its own connection with the X server and transmit Editres queries to the client through the server.

A deadlock condition will arise if the Editres Manager sends a request and then does not return control to the PIM. If control is not returned to the PIM, then the server-client connection is blocked and the Editres request cannot be sent to the client (and obviously, a reply will not be generated). This will cause the Editres Manager to hang until a timeout expires.

This situation is an example of a general problem in which various portions of the Application Manager need to generate X traffic which will produce a response (a so-called round trip request). Care must be taken that the operation is separated into two phases: an initiating phase, and an action phase which is invoked when the reply returns. For this reason we have built a callback mechanism into the PIM through which other Application Manager components can initiate a round trip request and then have a callback routine automatically executed when the reply is received.

CAVEATS

Editres Weaknesses

The implementation of Editres which comes with X11R5 is quite powerful, but has what is in our opinion a glaring omission. While the system supports queries to determine an application's widget hierarchy, determine geometry, and set resource values, it does not provide a means for retrieving resource values. The reason for this deficiency is that the

MIT-supplied toolkits do not support conversion from resource internal types to strings (although they do support the reverse operation). This capability is necessary for resource value retrieval so that resource values could be displayed.

We made a simple addition to Editres to support this behavior. This addition retains complete compatibility with the older version of Editres but allows the retrieval of resource values. The code modification is made in a shared library so none of the applications in the system need to be relinked. Dependence on shared libraries is not a generally acceptable solution however, so we plan to submit the code modifications back to the X Consortium for possible inclusion in future X releases.

Widget Set Dependencies

The names of widget classes and the resources associated with widgets vary from widget set to widget set. For example, push button widgets have different names and resources in the Athena widget set and the Motif widget set. These differences mean that there is a degree of widget set dependence in the Application Manager. Currently, support for a given widget set must be "hard wired" into the Rules Engine. Eventually, we hope to externalize widget set descriptions out of the system's code so that support for new widgets and widget sets can be added easily without recompiling.

STATUS

The components of the Application Manager are C++ objects; the current implementation is approximately 12,000 lines of code. Our implementation is for the Sun SPARCstation. The three audio servers discussed in this paper have been implemented as RPC services, with C++ wrappers around the RPC interfaces.

The synthesized speech server supports the DECtalk hardware and provides multiple user-definable voices. The non-speech audio server controls access to the build-in workstation sound hardware (/dev/audio on the SPARCstation in our case), and provides prioritized access and on-the-fly mixing. The spatialized sound server currently runs on either a NeXT workstation or an Ariel DSP-equipped Sun SPARCstation and supports the spatialization of up to 5 channels in real-time [Bur92].

In the current implementation, all of the Application Manager components except for the various sound servers execute as a single thread of control in the same address space. We are investigating whether a multithreaded approach would yield significant performance benefits and better code structuring.

Due to widget set dependencies inherent in using Editres, our current implementation only supports the Athena widget set. We are investigating building support for Motif and possibly other non-Intrinsics-based X toolkits.

Currently our rule set is somewhat limited. Creating rules in the present system is difficult because rule writers must have some degree of familiarity with both X and the Application Manager as a whole. For this reason there are several interface conditions which we do not handle well at the present. For example, dialog boxes which appear asynchronously (for example, to report an error message) are not brought to the attention of the user.

FUTURE DIRECTIONS

As we mentioned, translation rules are currently implemented as a set of C++ routines which evaluate predicates and fire actions. In the future we will move away from a code-based implementation of rules. We plan on supporting a textual rules language in which translation rules may be expressed. Further, we foresee using a multi-stage translation sequence which would support input event and output request translation, as well as the current semantic translation. A more flexible rules scheme will allow for greater user customization and experimentation.

We believe that the Application Manager provides an architecture for performing some interesting experiments in modifying application interfaces. Although we are currently working only with auditory output, there is no reason that the Application Manager could not be reconfigured to support general *retargeting* of interfaces. That is, the automatic translation of an application interface to a different interface, perhaps in a completely different modality. With a sufficient set of translation rules, application GUIs could theoretically be retargeted to virtually any desired interface including another (different) two-dimensional windowed interface, a three-dimensional visual interface, and so on.

In order to validate our auditory interface designs, we plan to conduct considerable user testing. At this time, we are compiling a list of visually-impaired computer users who have volunteered to participate in our experiments.

While the Application Manager can provide access to graphical applications, we feel that there is still a need for a more complete auditory environment in which applications and data may be grouped, controlled, and organized. Such an environment would be analogous to the desktop environments found on many graphical systems, which provide common file metaphors and inter-application operations such as drag-and-drop. We are designing such a system to serve as a higher level of abstraction for dealing with applications. [ME91] Our model is the Xerox PARC Rooms metaphor [HC86], implemented solely with auditory cues. This portion of the system will more fully utilize the spatialized sound capabilities we have built [Bur92].

ACKNOWLEGEMENTS

The Mercator project is a joint effort by the Georgia Tech Multimedia Computing Group (a part of the Graphics, Visualization, and Usability Center) and the Center for Rehabilitation Technology. We would like to thank John Goldthwaite of the CRT.

This work has been sponsored by the NASA Marshall Space Flight Center (Research Grant NAG8-194) and Sun Micro-

systems Laboratories. We would like to thank our technical contacts at these organizations, Gerry Higgins and Earl Johnson, respectively.

Also, a number of people have contributed a great deal of time and energy to the development of the Mercator Application Manager; without their hard work this system would not have been possible. We would like to thank Dave Burgess for his work on spatialized sound, Tom Rodriguez for his work on the application modeling system, and Ian Smith for his work at the lowest levels of the X protocol.

REFERENCES

[BBV90] L.H. Boyd, W.L. Boyd, and G.C. Vanderheiden. The graphical user interface: Crisis, danger and opportunity. *Journal of Visual Impairment and Blindness*, pages 496–502, December 1990.

[BGB91] Bill Buxton, Bill Gaver, and Sara Bly. The Use of Non-SpeechAudio at the Interface. *Tutorial Presented at CHI'91*. April 1991.

[Bur92] David Burgess. Low Cost Sound Spatilization. To appear in *UIST '92: The Fifth Annual Symposium on User Interface Software and Technology and Technolgy,* November 1992.

[Bux86] William Buxton. Human interface design and the handicapped user. In *CHI'86 Conference Proceedings*, pages 291–297, 1986.

[Che53] E.C. Cherry. Some experiments on the recognition of speech with one and two ears. J*ournal of the Acoustical Society of America*, 22, pages 61-62.

[HC86] Jr. D. Austin Henderson and Stuart K. Card. Rooms: The use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. *ACM Transactions on Graphics*, pages 211–243, July 1986.

[Edw89] Allstair D. N. Edwards. Modeling blind users' interactions with an auditory computer interface. *International Journal of Man-Machine Studies*, pages 575–589, 1989.

[Gav89] William W. Gaver. The sonicfinder: An interface that uses auditory icons. *Human Computer Interaction*, 4:67–94, 1989.

[HTAP90] HumanWare, Artic Technologies, ADHOC, and The Reader Project. Making good decisions on technology: Access solutions for blindness and low vision. In *Closing the Gap Conference*, October 1990. Industry Experts Panel Discussion.

[Lad88] Richard E. Ladner. Public law 99–506, section 508, electronic equipment accessibility for disabled workers. In *CHI'88 Conference Proceedings*, pages 219–222, 1988.

[LC91] Lester F. Ludwig and Michael Cohen. Multidimensional audio window management. *International Journal of Man-Machine Studies*, Volume 34, Number 3, pages 319-336, March 1991.

[LPC90] Lester F. Ludwig, Natalio Pincever, and Michael Cohen. Extending the notion of a window system to audio. *Computer*, pages 66–72, August 1990.

[ME91] Elizabeth Mynatt and Keith Edwards. New metaphors for nonvisual interfaces. In *Extraordinary Human-Computer Interaction*, 1991. Draft chapter accepted for upcoming book.

[ME92] Elizabeth Mynatt and W. Keith Edwards. The Mercator Environment: A Nonvisual Interface to X Windows and Unix Workstations. *GVU Tech Report GIT-GVU-92-05*. February 1992.

[Pet91] Chris D. Peterson. Editres-a graphical resource editor for x toolkit applications. In *Conference Proceedings, Fifth Annual X Technical Conferenc*e, Boston, Massachusetts, January, 1991.

[Sch87] Robert W. Scheifler. X window system protocol specification, version 11. Massachusetts Institute of Technology, Cambridge, Massachusetts, and Digital Equipment Corporation, Maynard, Massachusetts, 1987.

[Van89] G.C. Vanderheiden. Nonvisual alternative display techniques for output from graphics-based computers. *Journal of Visual Impairment and Blindness*, 1989.

[WWF88] E.M. Wenzel, F.L. Wightman, and S.H. Foster. Development of a Three-Dimensional Auditory Display System. *SIGCHI Bulletin*, 20, pages 557-564, 1988.

[Yor89] Bryant W. York, editor. Final Report of the Boston University Workshop on Computers and Persons with Disabilities, 1989.