[Bur92b] David Burgess. "Low Cost Sound Spatialization." In *Proceedings of ACM Symposium on User Interface Software and Technology, UIST'92*, November 1992.

[Bux86] William Buxton."Human Interface Design and the Handicapped User." In *Proceedings of ACM Conference on Computer-Human Interaction, CHI'86*, pp. 291-297, 1986.

[Edwa89] Alistair D. N. Edwards. "Modeling Blind User's Interactions with an Auditory Computer Interface." International Journal of Man-Machine Studies, pp 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 *Proceedings of ACM Conference on Computer-Human Interaction, CHI'88*, pp. 219-222, 1988.

[LC91] Lester F. Ludwig and Michael Cohen. "Multidimensional Audio Window Management." International Journal of Man-Machine Studies, 34:3, pp. 319-336, March 1991.

[LPC90] Lester F. Ludwig, Natalio Pincever, and Michael Cohen. "Extending the Notion of a Window System to Audio." Computer, pp. 66-72, August 1990.

[ME92a] Elizabeth Mynatt and W. Keith Edwards. "The Mercator Environment: A Nonvisual Interface to X Windows and Unix Workstations." GVU Technical Report GIT-GVU-92-05. February 1992.

[ME92b] Elizabeth Mynatt and W. Keith Edwards. "Mapping GUIs to Auditory Interfaces." In *Proceedings of ACM Symposium on User Interface Software and Technology, UIST'92*, 1992.

[Pet91] Chris D. Peterson. "Editres-A Graphical Resource Editor for X Toolkit Applications." In *Proceedings of the Fifth Annual X Technical Conference.* Boston, MA, January 1991.

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

We currently support two separate synthesized speech servers. One supports the DECtalk hardware, the other provides software-based synthesis. Each supports multiple user-definable voices. The non-speech audio server controls access to the built-in workstation sound hardware (/dev/audio on the SPARCstation in our case), and provides prioritized access and on-the-fly mixing of audio data. The spatialized sound server currently runs on either a NeXT workstation or an Ariel DSP-equipped Sun SPARCstation (both systems are based on the Motorola 56001 digital signal processor). The SPARC-based system supports spatialization of multiple channels in real-time [Bur92a][Bur92b].

All of the Application Manager components except for the various sound servers execute as a single thread of control in the same address space currently. We are investigating whether a multi-threaded approach would yield significant performance benefits and better code structuring.

Due to the widget set dependencies discussed in the Caveats section, above, our current implementation supports only the Athena widget set. We are investigating support for OLIT and Motif and possibly other non-Intrinsics-based X toolkits.

Acknowledgments

The Mercator project is a joint effort by the Multimedia Computing Group (a part of the Graphics, Visualization, & 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 Microsystems 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 substantial amount of time and energy to the development of Mercator; without their hard work this system would not have been possible. We would like to thank Dave Burgess and Ian Smith in particular.

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*, pp. 496-502, December 1990.

[BGB91] Bill Buxton, Bill Gaver, and Sara Bly. “The Use of Non-speech Audio at the Interface.” Tutorial presented at the ACM Conference on Computer-Human Interaction. New Orleans, 1991.

[Bur92a] David Burgess. "Real-Time Audio Spatialization with Inexpensive Hardware." In *Proceedings of International Conference on Signal Processing Applications and Technology.* Boston, MA, 1992.

We are exploring the possibility of creating an externalized representation of any widget set-specific information that would be used by the Mercator rules. This would allow us to more easily support different widget sets and customize support for our current widget sets.

Xt-Based Protocol

One of the goals of our system was to explore the quality of non-visual interfaces which could be produced from X applications using a completely external (outside the application) approach. While our system works, it is quite complex and has some weaknesses (which we have described in the Caveats section of this paper). The major limitations of our current system result from two basic properties of our information capture approach: (1) the X protocol is so low-level that it is both difficult to monitor, and difficult to create rules which have to express constructs in terms of the protocol, and (2) Editres was simply not designed for the task we are applying it to.

While we do get fairly good results from our current implementation, we are beginning to explore modifications to the Xt Intrinsics layer which would allow an external process to present an alternative interface to any Xt-based application, without requiring a complex protocol monitoring system. The modifications would consist of hooks into Xt which would allow an external process to communicate directly with the Xt layer of an application. This new Xt-based protocol would differ from Editres in several respects. Most importantly, it would be able to provide resource values to clients, and would support asynchronous notification of events in the interface (to obviate the need to constantly poll the application). Unlike Editres, the new protocol would also allow controlling input to be delivered to the application.

We feel that such a protocol, specifically designed for the task of providing alternative interfaces to X applications, would provide a greater degree of semantic information about the workings and structure of application interfaces than our current solution, and would be much easier to construct and modify.

The mechanisms built for such a system would have a great number of other applications as well. For example, since the protocol would allow another process to drive an X application, it would be possible to construct harnesses which would allow scripting or automated testing of graphical applications, much as it is possible to drive character-based applications programmatically in many systems now. A broadening of the goals of Mercator would allow “retargeting” of application interfaces to virtually any new modality, including speech (perhaps to allow interactive application use over a telephone), or even to provide a different look-and-feel in the 2-D graphical domain (for example, to translate a Motif application to an OPEN LOOK application “on-the-fly”).

Status

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

Often, a rule must be split across several chained callbacks because the rule initiates several round-trip requests or Editres queries.

Limitations of Rules

Currently it is difficult to modify or create new rules in our system. Since the rules are actually coded into the system, a knowledge of C++ is required at a minimum. Perhaps even more constraining is the fact that since rules must work with information stored in other parts of the Application Manager (for example, the Model Manager), generate Editres requests, and perhaps even modify the X protocol stream, a significant amount of knowledge of the implementation of the Application Manager and X itself is required to create new rules. There is a great need for a more high-level, abstract representation of rules and the facilities available to rules to allow for easier experimentation and customization of interfaces.

Future Directions

This section details our future plans for the Mercator system.

Generalize Input and Output

In the current implementation of Mercator, user input and system output are directly specified in the system's translation rules. That is, the particular input syntax and modalities which are required to control either the Application Manager or applications are hardcoded into the Application Manager itself. Similarly, the syntax and modalities of output are hardcoded into the Application Manager itself.

We are investigating a more flexible I/O system in which the translation rules would operate on abstract input and output tokens. Separate input and output management systems would govern mapping these abstract tokens into whatever particular syntax and modality the user desires.

Such a system would allow much more flexible customization of the non-visual interface according to user preferences. For example, users could "remap" the output mechanics to change the presentation of a dialog box to speech output, non-speech audio output, or even tactile output on a Braille printer. Similarly, users could customize the system to support different, perhaps overlapping, modalities for input. Speech, keyboard, and trackball input could all be used as application input according to users' specifications.

Externalization of Rules and Widget Descriptions

Our specification of translation rules and widget set-dependent class and resource names are currently hardcoded into the Application Manager. The rules in our system are expressed in a stylized C++ predicate/action notation by the system implementors. We are working on an externalized rules representation which would be read in and parsed at runtime by the Application Manager. This system will allow far easier customization of the interface by both the developers and users of the system.

creates problems when used to gather information for an interactive application like Mercator. The selection mechanism is a synchronous, multi-stage protocol, which unnecessarily increases the implementation complexity of Mercator and degrades interactive performance because of the transmission overhead. We were unable to address this problem without significantly modifying EditRes.

Widget Set Dependencies

There are a number of widget set dependencies in Mercator. Such dependencies exist because Editres returns widget class names and resource names which are particular to the widget set the application is written with. The rules in Mercator must look for and recognize this widget set-specific information to know how to treat particular widgets. Unfortunately such dependencies will probably exist in any system similar to Mercator.

Deadlock Potential

There are some interesting potential deadlock conditions which we 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 behavior of blocking the protocol stream can cause several problems. The most common problem relates to the use of Editres. 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 (specifically, the selection mechanism), and thus Editres communication must proceed through the PIM like all X traffic.

A deadlock condition will arise if a rule instructs the Editres Manager to send a request and then blocks waiting on the reply, rather than returning control to the PIM. If control is not returned to the PIM, then the connection between the client and server is blocked and the Editres request will never be sent. Obviously in this case a reply will never be generated so the Application Manager will hang forever waiting on a reply that will never come.

This situation is an example of a general problem in which various components of the Application Manager need to generate X traffic which will produce a response (the selection protocol is one example of this, as are all round-trip requests generated by the Application Manager). 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.

We have constructed a callback mechanism in the Application Manager which can be used by rules when they would otherwise block waiting on communication. One unfortunate consequence of the wide use of callbacks is that the control flow in the Application Manager is rather complex.

In order to bypass widgets which the user does not need to be aware of, various classes of widgets are marked as not navigable by the Model Manager. Current marked classes are Core, Viewport, Grip and Transient. As navigation input from the user is processed, the user's position is moved one step in the tree (left, right, up, down). If the new position is at a class marked as not navigable, then the navigation request is propagated another step. The direction for the next step is dependent on whether the current object has siblings and/or children in the tree structure.

This approach does not allow us to bypass all the widgets we would like to avoid. Since our navigation is still based on the full widget hierarchy, there may be widgets which are critical to the overall tree structure which cannot be ignored. One example is a widget which has both siblings and children. Currently, it is not possible to navigate around such a widget in a systematic manner and not miss other widgets. We are exploring new algorithms to allow more flexibility in navigating these structures.

Caveats

There are a number of weaknesses in our current system. This section explores some of the more important of these weaknesses and provides some insight into various solutions we are attempting.

Editres

The Editres protocol was designed to allow easy customization of X applications. As such, it provides requests to get the widget hierarchy of an application, query the names and types of resources in widget classes, and change the values of resources in particular widgets. While these requests may support simple customization of applications, they are insufficient for Mercator because they do not provide a way to query the value of resources.

Widgets are sometimes used to implement several different interface objects and are configured based on the values of resources. For example, the XmRowColumn widget in the Motif toolkit may be used as a popup menu, a pulldown menu, an option menu, a menu bar or a generic work area. This information is contained in the XmNrowColumnType resource. While we might be able to infer the capacity the XmRowColumn was acting in, the information is directly available in the widgets resources. Because of the potential power of a GetValues request, we decided to extend Editres to support it, with the hope that the MIT-supplied Editres will eventually support a similar request.

Another limitation of Editres that is related to its original design as a customization protocol is that it only supports polling. The information that Editres returns is only valid at the moment that it is sent to the Editres client. Changes in the widget hierarchy are not reported back to the Editres clients as they happen. Due to this limitation, we are required to watch for CreateWindow requests in the protocol stream and then request Editres to resend the widget hierarchy. This scenario exacerbates our flow control problems (see Deadlock Potential) and results in extraneous requests since there is no way to determine when the interface has stabilized.

Additionally, Editres uses ClientMessages and the selection mechanism to transmit requests and replies. While this transport mechanism is acceptable for a protocol intended for customization, it

Keyboard Input

Currently all input to the system is through the keyboard. Hence, since X "owns" the keyboard, all input to the system comes through the X server. It is the task of the Application Manager to translate keyboard input into appropriate events (including possibly mouse events) and routing those new events to the appropriate applications as a means of control. The Application Manager must also detect "meta controls;" that is, input which is designated for the Application Manager itself, rather than one of the applications running on the system.

There are some potential problems with input to the Application Manager which are related to the fact that all input to the system comes through the X server. The primary problem is that not all widgets solicit keyboard-related events and, in general, if a particular widget does not solicit a certain event type, the server will never generate that event. This implies that some widgets will not be "navigatable." That is, once the current context has changed to a widget that is not soliciting keyboard input, no further keyboard events will be generated from the server and it will be impossible to move out of that widget.

To work around this problem we have a set of rules which instruct the protocol interest manager to modify the X protocol stream in certain ways. First, whenever a CreateWindow request is generated, we intercept and modify the request to ensure that the newly-created window has an event mask attribute which will cause keyboard events to be generated. Second, we monitor the protocol for ChangeWindowAttribute requests, and modify these requests if need be to ensure that a client does not "turn off" solicitation for keyboard events at some later point in time. The Application Manager stores the "natural" state of event solicitation in the Model Manager, so that extraneous keypresses will never be sent to windows which are not supposed to receive keyboard input. This approach seems to work quite well in practice.

Of course, not all input is directed to the Application Manager. Some input is intended for the applications themselves. Since the PIM has full control over the X protocol stream, we can simply remove events which are intended for the Application Manager. Thus, the "reserved" keys which are used for navigation are never seen by applications. Other keyboard events are passed through to the applications unmodified. Note that under our approach any standard application keyboard accelerators should continue to work.

We are currently beginning work on using other input streams (such as voice) which do not have to travel through the X server. We feel that the use of multiple input sources will greatly improve the human-computer bandwidth of the interface. We are also beginning to investigate the use of the X Input Extension as a means of accepting input from novel input devices.

Bypassing Unwanted Objects

The concept of the user's context is maintained by keeping a pointer to the user's position in the application's widget hierarchy which is stored in the off-screen model by the Model Manager. The user navigates the application interface by moving throughout this tree structure. But the granularity of the user's movements is in terms of interface objects not X widgets. For this reason, many of the X widgets in the tree structure are skipped during the traversal process. The strategy is to place the user in the most specific widget which corresponds to the current interface object.

Our second assumption is that all text in a window is drawn using the same font. This constraint eliminates some of the same bookkeeping needed for proportional fonts and appears to be a fairly valid assumption in general, though it is not always true.

Using these constraints we are able to model text similarly to text on a terminal. Each window containing text is divided into a grid based on the text's font and is stored and manipulated based on this grid. This model supports CopyArea and ClearArea requests since we can easily map the rectangular coordinates supplied in these requests onto the grid.

The assumptions we made for handling text were acceptable for our prototype system but the use of proportional fonts and the mixing of fonts are important issues which must be resolved in future systems.

Sound Servers

As mentioned earlier, output to the user is generated by the specific rules fired by the Rules Engine. When fired, rules can invoke methods on the Sound Manager object (see Figure 1) which then generates the specified auditory output. The Sound Manager provides a single programmatic interface to the various sound resources available on our platform.

Currently, we use three types of sound output: speech, simple non-speech audio, and high-quality spatialized (3-D) sound. Each type of sound output is generated by different hardware, and thus there is a separate server process for each which regulates access to the sound hardware. These servers support access via multiple processes to workstation sound resources and allow sound hardware to reside on other machines across a network. See the Status section for more details on the particulars of these servers.

User Input

In addition to presenting application interfaces in a non-visual modality, the Application Manager is responsible for processing user input to both the applications and to the Application Manager itself. In essence, the Application Manager must *run* the existing application based on user input in the new interface. This section presents the mechanisms used for user input.

The Notion of Context

The Application Manager maintains the notion of the user's *current context* in the environment as a construct to support navigation. The current context is defined as the application the user is currently working in, and a particular widget within that application where the user is currently "located." Users navigate through applications by changing their current context and by operating widgets at their current location. The Application Manager effects changes in context by actually warping the mouse pointer to the desired location in response to user input. We warp the pointer to ensure that applications in our environment behave exactly as they would in a visual environment if the user moved the mouse pointer to a new widget.

Templates

To obviate the need to install a large number of rules to deal with the different semantics of each individual widget in the interface, 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 in the original interface. Templates provide consistency in the “hear and feel” of objects in Mercator interfaces.

For example, the rule template for widgets of the PushButton class may provide a set of rules to be fired whenever the PushButton instance is selected, 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 that particular PushButton instance which governs its interaction in the new interface. Rule templates provide a mechanism for ensuring standard behavior across all instances of a particular widget class with little work.

Text Handling

Computer interfaces are still mostly comprised of text. Early screen-reader systems for text-based interfaces (such as command line interfaces) accessed the screen contents by reading the ASCII codes stored in the display’s framebuffer. In these systems, the spatial relationships between pieces of text on the screen was well-defined since the text was laid out in a grid. Since the basic display unit of GUIs is the pixel, capturing text in a GUI presents a different set of problems.

First, text drawn in a GUI is stored as pixels in the framebuffer, which would be difficult to interpret as characters. Therefore the text must be captured before it is drawn. In Mercator this is accomplished by intercepting the text drawing requests, ImageText and PolyText. These drawing requests are the only means of drawing text under X unless the application renders text itself a pixel at a time. The text contained in these requests does not correspond directly to the text that will appear on the screen. All text under X is drawn using a font, with the characters acting as indices into the font. Thus the character “A” may be drawn as “A” in most fonts, but may appear as a special symbol in others. Thus we must not only keep track of text, but what font it was drawn in.

Another problem with handling text in a GUI is that everything is drawn at pixel locations, so it is difficult to determine the effects of overwriting text with new text or graphics. Likewise, graphics requests are used to copy and erase text, so we need to accurately model the location and movement of text at the pixel level.

To solve these problems, our present implementation operates under two assumptions concerning the display of text. First, applications are not allowed to use proportional fonts to draw text. This constraint simplifies the bookkeeping for handling text since we can assume that each character in a font takes up a fixed amount of space. This constraint is also easily configurable through the use of the resource database and enforceable by intercepting OpenFont requests.

Rules Engine

The heart of the Mercator Applications Manager is the Rules Engine (see Figure 1). The Rules Engine takes as input the current state and structure of the interface, and any user input, and produces auditory output according to a set of translation rules. From a high-level standpoint, the Rules Engine has two responsibilities: to present the application interface to the user, and to respond to input from the user and generate controlling events to the application and possibly the Application Manager itself. This section deals with the first goal, presenting the interface. See "User Input" below for a description of how we handle user input.

The Rules Engine is driven asynchronously by the Protocol Interest Manager, which taps the connection between the X client and the server. The Rules Engine informs the PIM of patterns in the X protocol which should cause control to be passed to the Rules Engine (that is, the Rules Engine expresses a "protocol interest," hence the name of the Protocol Interest Manager). When the Rules Engine is "awakened" by the PIM, it examines the current state of the protocol and may fire one or more rules, which may take any of a number of actions, including of course the production of auditory output.

The facilities available to translation rules are quite complex. For example, rules can stall the X protocol stream (that is, queue requests and events at the PIM for later delivery). Further, rules can actually insert new requests or events into the protocol stream, change existing requests or events before they are delivered, or delete requests or events. When packets are inserted or deleted the PIM will rewrite later packets to ensure that sequence numbers remain consistent.

Rules can examine or update the interface representation stored in the Model Manager. They may also query applications and collect replies via Editres. Information returned via Editres is used to update the off-screen model of the interface's state.

Rules perform a number of functions in our system. Some rules are used only for "maintenance" of the interface representation. For example, when new windows are created, a rule fires which updates the window hierarchy representation in the Model Manager. If the request to create the window came from a client which understands the Editres protocol (an attribute which is stored in the Client objects in the Model Manager) then a second rule will fire to retrieve an updated copy of the application's widget hierarchy in an attempt to match the newly created window to an actual widget.

Other rules actually create the auditory presentation of the interface. In the window creation example above, a rule would also fire which would examine the state of the interface and the newly created object and generate some auditory output to inform the user that, for example, a dialog box has just been created.

Our rules are currently implemented in a stylized C++ predicate/action notation. Rule list traversal is quite fast: predicates are preprocessed and organized into hash tables based on protocol interest patterns (which form the basis of rule predicates). Thus, when a particular protocol sequence occurs, the system does not need to test the predicates of all rules, just the ones that have the specified protocol pattern as a predicate.

ciated with the object. When XtObj instances are created, they also install “back-pointers” into the Model Manager’s Win objects so that it is easy to locate a particular widget based on a window ID. As a convenience, Client objects also maintain a list of the top-level windows of their application.

There are a number of benefits to our approach to interface modeling. Because of the way our data structures are keyed, it is easy to determine which widget or widgets correspond to a window ID in the protocol stream. As an example, when we see a window being mapped in the protocol, we can use the window ID to determine the widget in the interface which has just been popped up. The information stored in the Model Manager allows us to interpret the X protocol to associate higher-level semantic meaning to occurrences in the interface.

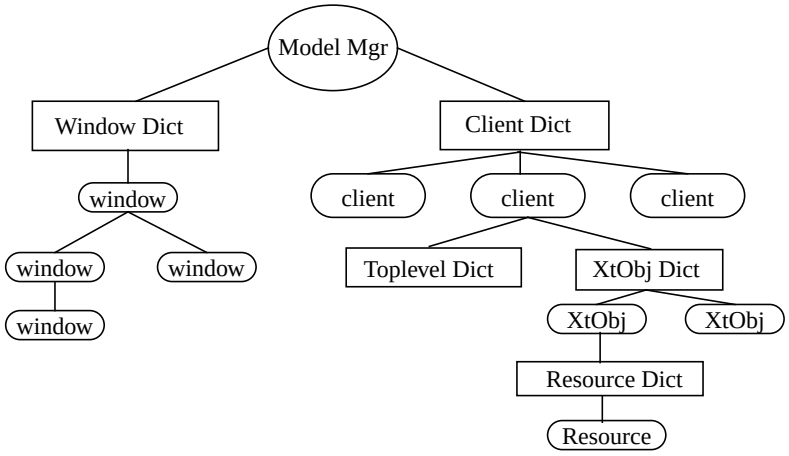

FIGURE 2: MODEL MANAGER DATA STRUCTURES

Application Output

Once we have captured information about the interface of a running application, and stored that information in our off-screen model, we still must present that interface structure to the user in a meaningful manner. To create an auditory presentation of the interface representation stored in our off-screen model, we apply a set of translations to the interface representation. These translations produce a consistent auditory interface which is semantically similar to our stored representation of the application’s original graphical interface. This section describes the particular mechanisms we use to accomplish this translation.

are notified of changes in the state of the application via the X protocol, and can then use Editres to understand and interpret the meaning of the protocol in terms of high-level constructs such as buttons, menus, and scrollbars.

Modeling Applications

Based on the information retrieved from an application, we must be able to synthesize a coherent model of the application's interface. We need to keep this interface representation locally so that we can have information about the interface available to us at all times without having to constantly poll the application to retrieve needed data. Also, we can store the interface information locally in a convenient format which can provide easy access to particular data as needed.

This model must provide a representation for not only the structure of the interface, but also appearance attributes, behavioral attributes, and semantic attributes. Our modeling techniques must be sufficiently flexible to support representing these (and possibly other) attributes of the applications which run in our environment.

Figure 2 provides a structural overview of the data structures we use to model application interfaces. The data structures shown here are controlled by the Model Manager component of the Application Manager. The Model Manager provides the programmatic interface to the interface representation for the rest of the Application Manager.

Since many clients may not implement Editres, the only information which is always *guaranteed* to be available to us is information present in the X protocol stream. The most important structural attribute of interfaces which is present in the protocol is the window hierarchy of the application. Thus our data structures provide a representation of this lowest common denominator of all X applications, the window. The Model Manager maintains a dictionary (key-value mapping) of Win objects which mirror the actual window hierarchy present on the display server. Win objects are our internal representation of relevant window attributes. This dictionary is updated automatically by the Protocol Interest Manager as windows are created, destroyed, or modified on the display server.

Of course the problem with modeling windows alone is that, by themselves, windows are too low-level to be meaningful when trying to understand the semantics of an application's interface. To overcome some of the limitations of representing the window hierarchy alone, we also maintain a representation of the interface objects (widgets or gadgets) in the interface on a per-client basis. Note that this information will only be available if the application in question understands the Editres protocol. The Model Manager keeps a list of Client objects, each of which represents one application running in the system. Client objects have unique identifiers based on the connection number of the application's protocol stream to the X server.

Client objects maintain the name and class of the application (sent as properties through the X protocol), and keep a dictionary which represents the interface object hierarchy of the application. Our internal representation for these interface objects is the XtObj class. Each XtObj instance models one widget or gadget in the application interface, and maintains information about parent and children object identifiers, the window ID of the object (if applicable), and any resources asso-

the window, the border of the window is changed (highlighted). Based solely on the low-level information present in the X protocol it would be extremely difficult if not impossible to derive the fact that the created object is a push button.

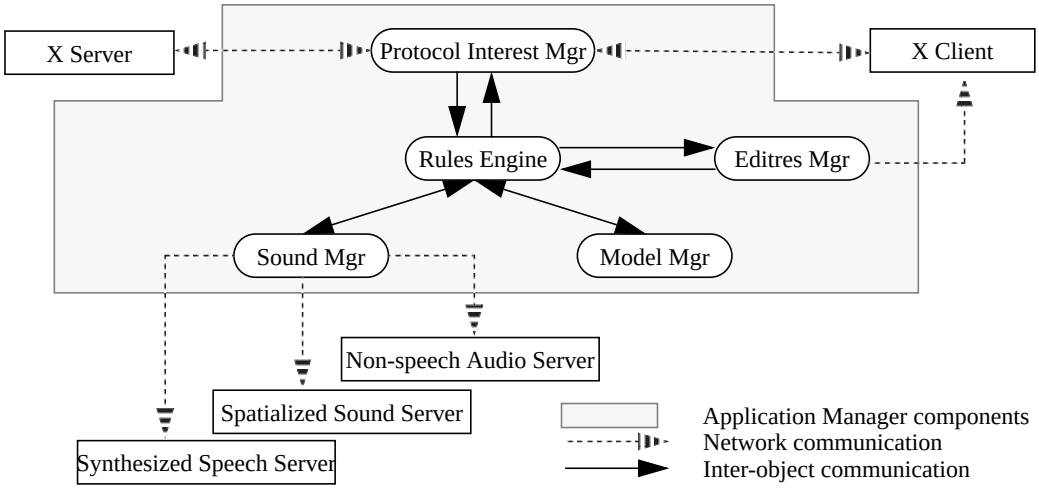

FIGURE 1: OVERVIEW OF THE APPLICATION MANAGER

Editres

To solve many of the problems which would be present if we relied on protocol monitoring alone, the Mercator Application Manager also makes use of the Editres protocol [Pet91]. The Editres protocol, introduced in X11R5, was designed to allow easy, interactive customization of applications. Editres provides facilities for querying running applications about their widget hierarchies, widget names and classes, and resources. This information is maintained by the Xt Intrinsics layer and is resident in the client itself; other than Editres there is no way for an external process to access this type of information in another client. The component of the Application Manager which implements the Editres protocol is called the Editres Manager (see Figure 1).

We use the information which can be obtained via Editres to interpret the X protocol stream. In our push button example above, when we detect via the protocol that a new window has been created and mapped, we generate an Editres query to the application to retrieve the widget hierarchy of the application. Contained in the reply is the name, class, widget ID, and window ID of all of the widgets and gadgets in the application. By matching the returned window IDs to the ID of the new window in the protocol, we can determine that indeed a push button has just been created.

Combining the semantically high-level information from Editres with the all-inclusive protocol-based information gives us a good picture of what is going on inside the application interface. We

There are a number of possible approaches to providing this level of information about application interfaces. One class of approaches involves changing the applications themselves, or the toolkits the applications are built with. By making direct modifications to application or toolkit code it is possible to introduce the mechanisms for non-visual interfaces directly and efficiently into applications; the code which produces the non-visual interface has available to it high-level information about the original interface and its behavior since it is "inside" the application. Of course, the drawback to this approach is that it requires modifications to applications and toolkits, and thus is not a transparent solution: it will not work with existing systems without recompilation (or at the least, relinking).

We have taken a different approach to the problem of information capture. Our system provides a non-visual interface to existing, unmodified X applications by using an external process to capture information about the interface at run-time. Our approach has the advantage that it is completely transparent: client applications and the window server do not (and indeed *cannot*) know that our alternative interface system is in use. The chief drawback to our approach is that since our system exists outside of the applications and window system, it has only a limited amount of information available about the original graphical interfaces it is attempting to translate.

This section presents the two mechanisms we are using to retrieve interface information from running applications. Both of these mechanisms operate transparently to both the applications and the window system.

Protocol Monitoring

The most basic technique we use to capture interface information is to monitor the exchange of X protocol packets between the X server and clients. The component of the Mercator Application Manager which accomplishes this protocol monitoring is called the Protocol Interest Manager. See Figure 1 for a structural overview of the Application Manager.

The Protocol Interest Manager, or PIM, is in essence a *pseudo-server*. That is, it is a separate process which, to clients, appears to be an X Window System server. To the server itself, the PIM appears to be a client. Clients connect directly to the PIM in our system and then the PIM establishes a surrogate connection to the "true" X server. Applications cannot distinguish the PIM from a real X server.

By tapping the connection between X clients and the server, the PIM is in a position to be notified of all changes in the interface: all window creations, maps, and unmaps, text and graphics rendering, and size changes are represented in the protocol. The X protocol is the only mechanism for drawing to the screen (since the screen is owned by the X server itself in most cases), and thus it is essentially impossible to circumvent the X protocol to draw to the screen. The PIM is aware of any and all application output to the screen.

But even though monitoring the protocol can tell us in complete detail *what* has happened in the interface, it cannot tell us *why* something has happened. The information in the X protocol is extremely low-level and provides virtually no semantic clues as to the meaning of events in the protocol. For example, when a push button is created, a sequence of protocol will be generated which creates and maps a window, and draws some text into it. When the mouse is moved into

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.

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 control and navigation allows the user to safely scan the interface without activating interface controls.

Architectural Overview

The previous sections of the paper discussed the motivation and design for a non-visual interface which is based on the semantic constructs in the original graphical interface of an application . We have also presented a keyboard-based navigation paradigm which allows users to scan and operate the interface. Now we shall discuss the architecture we have constructed to provide this alternative interface presentation and navigation. The component of the Mercator project which performs this translation of X-based graphical interfaces to auditory interfaces is called the Application Manager.

The Application Manager itself is composed of several components, and we shall discuss each of those components in turn. The remainder of this paper presents the tasks which must be performed by the Application Manager and the components of the system which accomplish these tasks.

Capturing Interface Information

The most important technical obstacle which must be overcome in order to provide an alternative interface to existing applications is the problem of how to capture or retrieve information about the interfaces of applications as they are running. A system which provides an alternative interface to existing applications must ideally have access to the semantic properties of the original interface (for example, the types of objects in the interface and the structure of the interface), as well as the behavioral attributes of the interface ("clicking this button causes a dialog box to pop up"). Lower-level "syntactic" information may also be needed ("this text is in boldface, this button is red").

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 can be used to convey some 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

TABLE 1: USING FILTEARS TO CONVEY AIC ATTRIBUTES

Navigation Paradigms

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

so many items) would need to be based on the auditory presentation of the scrollable object not its visual presentation.

Essentially, we have chosen a compromise between the two extremes outlined above. To ensure compatibility between visual and nonvisual interfaces, we are translating the interface at the level of the interface objects which form the user's model of the application interface. 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.

Another design consideration is which nonvisual interface modality to use. The obvious choices are auditory and tactile. We are currently basing our design on previous work in auditory interfaces which has demonstrated that complex auditory interfaces are usable [BGB91][Edwa89]. Another factor that we considered is that a significant portion of people who are blind also suffer from diabetes which may cause a reduction in their sensitivity to tactile stimuli [HTAP90]. Nevertheless our system will eventually have tactile components as well. For example, a braille terminal provides an alternate means for conveying textual information which may be preferred to speech synthesis.

Interface Components

As stated in the last section, the auditory presentation will be based on the user's model of the application interface. This model will be composed of the objects that make up the graphical presentation. The constructs of the user's model can be thought of in terms of common interface objects such as menus, buttons, and dialog boxes or in terms of the functions afforded by these objects such as "selection from a set" or "containing heterogenous types of information". In X Windows applications, these objects 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.

In Mercator, we call the objects in our auditory presentation Auditory Interface Components, or AICs. The translation from graphical interface components to AICs occurs at the widget level. The Mercator Model Manager stores the widget hierarchy for the application interface and any attribute information associated with the widgets. Using a set of rules and widget templates, these widgets are combined to form abstract interface objects.

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.

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

ent access to applications, we need to build a framework which will allow us to monitor, model and translate graphical interfaces of X Window System applications without modifying the applications. Second, given these application models, we need to develop a methodology for translating graphical interfaces into nonvisual interfaces.

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.

Next we describe the architecture we have constructed to provide this interface transparently for X 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.

Auditory Interfaces

The primary human-interface 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 Audio GUIs. We then describe the presentation of common interface objects such as buttons, windows, and menus, and detail the navigation paradigm for Mercator interfaces.

Design Considerations

There are several design decisions we had to make when constructing our nonvisual interface. A 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. The user could move a mouse over a screen and hear the objects announced as the cursor touched the object. In this type of system the user 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, the system could 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.

The primary question here is determining what aspects of the graphical interface contribute to the user's model of the application and what aspects of the graphical interface are simply visual artifacts of its graphical presentation. Retaining the user's model of the application interface across presentations is necessary to support collaboration between sighted and non-sighted users. It is also important to remove visual artifacts of the graphical presentation which do not make sense in the auditory domain. For example, scrollbars serve two purposes in graphical interfaces. First they conserve limited screen real estate by presenting only a portion of a list. Second, they provide a mechanism for the user to quickly search the list. The first use of the scrollbar most likely does not need to be conveyed in the auditory interface since there is no limited display real estate. The second use does need to be transferred to the auditory domain. But the movement of the scrollbar (up or down

Introduction

A common design principle for building computer applications is to separate the design and implementation of the application functionality from the application interface. The reasons for this separation are clear. If the application interface, or more precisely, the presentation of the application interface is independent of the application behavior then the interface can be easily modified to suit the needs of a wide range of users.

The graphical user interface is at this time an extremely common vehicle for presenting a human-computer interface. There are many times, however, when a graphical user interface is inappropriate or unusable. One example is when the task requires that the user's visual attention is directed somewhere other than the computer screen. Another example is when the computer user is blind or visually-impaired. Unfortunately, graphical user interfaces, or GUIs, have disenfranchised this portion 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]. The motivation for this legislation is clear. As more organizations move to a standard graphical environment, computer users who are visually-impaired may lose employment. Although this legislation has yet to be tested in the courts, both vendors and consumers of computer equipment and software are beginning to seriously address accessibility concerns.

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† Project, is to provide *transparent* access to X Windows applications for computer users who are blind or severely visually-impaired [ME92a][ME92b]. In order to achieve this goal, we need to solve two major problems. First, in order to provide transpar-

---

† 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 graphical display to an auditory display of the same user interface.

Published in *The X Resource*, #7, Summer, 1993. O'Reilly Publishers, Sebastopol, CA.

THE MERCATOR PROJECT

A NON-VISUAL INTERFACE TO THE X WINDOW SYSTEM

W. Keith Edwards, Elizabeth D. Mynatt, and Tom Rodriguez

ABSTRACT

This paper describes work to provide mappings between X-based graphical user 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 user interfaces for users who are blind or severely visually impaired. We describe the features of an auditory interface which simulates many of the characteristics of graphical interfaces. We then describe the architecture we have built to model and transform graphical interfaces. We present some of the difficulties encountered in building such a system on top of X. Finally, we conclude with some indications of future work.

Keith Edwards is a Research Assistant at the Georgia Tech Multimedia Computing Group. His research interests focus on computer-supported cooperative work. His email address is keith@cc.gatech.edu.

Beth Mynatt is a Research Scientist at Georgia Tech where she manages the Multimedia Computing Group. Her interests include auditory interfaces and novel interaction techniques. Her email address is beth@cc.gatech.edu.

Tom Rodriguez is a Research Assistant at the Multimedia Computing Group. His interests include window systems and digital video. His email address is jack@cc.gatech.edu.

[Bur92b] David Burgess. "Low Cost Sound Spatialization." In *Proceedings of ACM Symposium on User Interface Software and Technology, UIST'92*, November 1992.

[Bux86] William Buxton."Human Interface Design and the Handicapped User." In *Proceedings of ACM Conference on Computer-Human Interaction, CHI'86*, pp. 291-297, 1986.

[Edwa89] Alistair D. N. Edwards. "Modeling Blind User's Interactions with an Auditory Computer Interface." International Journal of Man-Machine Studies, pp 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 *Proceedings of ACM Conference on Computer-Human Interaction, CHI'88*, pp. 219-222, 1988.

[LC91] Lester F. Ludwig and Michael Cohen. "Multidimensional Audio Window Management." International Journal of Man-Machine Studies, 34:3, pp. 319-336, March 1991.

[LPC90] Lester F. Ludwig, Natalio Pincever, and Michael Cohen. "Extending the Notion of a Window System to Audio." Computer, pp. 66-72, August 1990.

[ME92a] Elizabeth Mynatt and W. Keith Edwards. "The Mercator Environment: A Nonvisual Interface to X Windows and Unix Workstations." GVU Technical Report GIT-GVU-92-05. February 1992.

[ME92b] Elizabeth Mynatt and W. Keith Edwards. "Mapping GUIs to Auditory Interfaces." In *Proceedings of ACM Symposium on User Interface Software and Technology, UIST'92*, 1992.

[Pet91] Chris D. Peterson. "Editres-A Graphical Resource Editor for X Toolkit Applications." In *Proceedings of the Fifth Annual X Technical Conference.* Boston, MA, January 1991.

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

We currently support two separate synthesized speech servers. One supports the DECtalk hardware, the other provides software-based synthesis. Each supports multiple user-definable voices. The non-speech audio server controls access to the built-in workstation sound hardware (/dev/audio on the SPARCstation in our case), and provides prioritized access and on-the-fly mixing of audio data. The spatialized sound server currently runs on either a NeXT workstation or an Ariel DSP-equipped Sun SPARCstation (both systems are based on the Motorola 56001 digital signal processor). The SPARC-based system supports spatialization of multiple channels in real-time [Bur92a][Bur92b].

All of the Application Manager components except for the various sound servers execute as a single thread of control in the same address space currently. We are investigating whether a multi-threaded approach would yield significant performance benefits and better code structuring.

Due to the widget set dependencies discussed in the Caveats section, above, our current implementation supports only the Athena widget set. We are investigating support for OLIT and Motif and possibly other non-Intrinsics-based X toolkits.

Acknowledgments

The Mercator project is a joint effort by the Multimedia Computing Group (a part of the Graphics, Visualization, & 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 Microsystems 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 substantial amount of time and energy to the development of Mercator; without their hard work this system would not have been possible. We would like to thank Dave Burgess and Ian Smith in particular.

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*, pp. 496-502, December 1990.

[BGB91] Bill Buxton, Bill Gaver, and Sara Bly. “The Use of Non-speech Audio at the Interface.” Tutorial presented at the ACM Conference on Computer-Human Interaction. New Orleans, 1991.

[Bur92a] David Burgess. "Real-Time Audio Spatialization with Inexpensive Hardware." In *Proceedings of International Conference on Signal Processing Applications and Technology.* Boston, MA, 1992.

We are exploring the possibility of creating an externalized representation of any widget set-specific information that would be used by the Mercator rules. This would allow us to more easily support different widget sets and customize support for our current widget sets.

Xt-Based Protocol

One of the goals of our system was to explore the quality of non-visual interfaces which could be produced from X applications using a completely external (outside the application) approach. While our system works, it is quite complex and has some weaknesses (which we have described in the Caveats section of this paper). The major limitations of our current system result from two basic properties of our information capture approach: (1) the X protocol is so low-level that it is both difficult to monitor, and difficult to create rules which have to express constructs in terms of the protocol, and (2) Editres was simply not designed for the task we are applying it to.

While we do get fairly good results from our current implementation, we are beginning to explore modifications to the Xt Intrinsics layer which would allow an external process to present an alternative interface to any Xt-based application, without requiring a complex protocol monitoring system. The modifications would consist of hooks into Xt which would allow an external process to communicate directly with the Xt layer of an application. This new Xt-based protocol would differ from Editres in several respects. Most importantly, it would be able to provide resource values to clients, and would support asynchronous notification of events in the interface (to obviate the need to constantly poll the application). Unlike Editres, the new protocol would also allow controlling input to be delivered to the application.

We feel that such a protocol, specifically designed for the task of providing alternative interfaces to X applications, would provide a greater degree of semantic information about the workings and structure of application interfaces than our current solution, and would be much easier to construct and modify.

The mechanisms built for such a system would have a great number of other applications as well. For example, since the protocol would allow another process to drive an X application, it would be possible to construct harnesses which would allow scripting or automated testing of graphical applications, much as it is possible to drive character-based applications programmatically in many systems now. A broadening of the goals of Mercator would allow “retargeting” of application interfaces to virtually any new modality, including speech (perhaps to allow interactive application use over a telephone), or even to provide a different look-and-feel in the 2-D graphical domain (for example, to translate a Motif application to an OPEN LOOK application “on-the-fly”).

Status

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

Often, a rule must be split across several chained callbacks because the rule initiates several round-trip requests or Editres queries.

Limitations of Rules

Currently it is difficult to modify or create new rules in our system. Since the rules are actually coded into the system, a knowledge of C++ is required at a minimum. Perhaps even more constraining is the fact that since rules must work with information stored in other parts of the Application Manager (for example, the Model Manager), generate Editres requests, and perhaps even modify the X protocol stream, a significant amount of knowledge of the implementation of the Application Manager and X itself is required to create new rules. There is a great need for a more high-level, abstract representation of rules and the facilities available to rules to allow for easier experimentation and customization of interfaces.

Future Directions

This section details our future plans for the Mercator system.

Generalize Input and Output

In the current implementation of Mercator, user input and system output are directly specified in the system's translation rules. That is, the particular input syntax and modalities which are required to control either the Application Manager or applications are hardcoded into the Application Manager itself. Similarly, the syntax and modalities of output are hardcoded into the Application Manager itself.

We are investigating a more flexible I/O system in which the translation rules would operate on abstract input and output tokens. Separate input and output management systems would govern mapping these abstract tokens into whatever particular syntax and modality the user desires.

Such a system would allow much more flexible customization of the non-visual interface according to user preferences. For example, users could "remap" the output mechanics to change the presentation of a dialog box to speech output, non-speech audio output, or even tactile output on a Braille printer. Similarly, users could customize the system to support different, perhaps overlapping, modalities for input. Speech, keyboard, and trackball input could all be used as application input according to users' specifications.

Externalization of Rules and Widget Descriptions

Our specification of translation rules and widget set-dependent class and resource names are currently hardcoded into the Application Manager. The rules in our system are expressed in a stylized C++ predicate/action notation by the system implementors. We are working on an externalized rules representation which would be read in and parsed at runtime by the Application Manager. This system will allow far easier customization of the interface by both the developers and users of the system.

creates problems when used to gather information for an interactive application like Mercator. The selection mechanism is a synchronous, multi-stage protocol, which unnecessarily increases the implementation complexity of Mercator and degrades interactive performance because of the transmission overhead. We were unable to address this problem without significantly modifying EditRes.

Widget Set Dependencies

There are a number of widget set dependencies in Mercator. Such dependencies exist because Editres returns widget class names and resource names which are particular to the widget set the application is written with. The rules in Mercator must look for and recognize this widget set-specific information to know how to treat particular widgets. Unfortunately such dependencies will probably exist in any system similar to Mercator.

Deadlock Potential

There are some interesting potential deadlock conditions which we 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 behavior of blocking the protocol stream can cause several problems. The most common problem relates to the use of Editres. 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 (specifically, the selection mechanism), and thus Editres communication must proceed through the PIM like all X traffic.

A deadlock condition will arise if a rule instructs the Editres Manager to send a request and then blocks waiting on the reply, rather than returning control to the PIM. If control is not returned to the PIM, then the connection between the client and server is blocked and the Editres request will never be sent. Obviously in this case a reply will never be generated so the Application Manager will hang forever waiting on a reply that will never come.

This situation is an example of a general problem in which various components of the Application Manager need to generate X traffic which will produce a response (the selection protocol is one example of this, as are all round-trip requests generated by the Application Manager). 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.

We have constructed a callback mechanism in the Application Manager which can be used by rules when they would otherwise block waiting on communication. One unfortunate consequence of the wide use of callbacks is that the control flow in the Application Manager is rather complex.

In order to bypass widgets which the user does not need to be aware of, various classes of widgets are marked as not navigable by the Model Manager. Current marked classes are Core, Viewport, Grip and Transient. As navigation input from the user is processed, the user's position is moved one step in the tree (left, right, up, down). If the new position is at a class marked as not navigable, then the navigation request is propagated another step. The direction for the next step is dependent on whether the current object has siblings and/or children in the tree structure.

This approach does not allow us to bypass all the widgets we would like to avoid. Since our navigation is still based on the full widget hierarchy, there may be widgets which are critical to the overall tree structure which cannot be ignored. One example is a widget which has both siblings and children. Currently, it is not possible to navigate around such a widget in a systematic manner and not miss other widgets. We are exploring new algorithms to allow more flexibility in navigating these structures.

Caveats

There are a number of weaknesses in our current system. This section explores some of the more important of these weaknesses and provides some insight into various solutions we are attempting.

Editres

The Editres protocol was designed to allow easy customization of X applications. As such, it provides requests to get the widget hierarchy of an application, query the names and types of resources in widget classes, and change the values of resources in particular widgets. While these requests may support simple customization of applications, they are insufficient for Mercator because they do not provide a way to query the value of resources.

Widgets are sometimes used to implement several different interface objects and are configured based on the values of resources. For example, the XmRowColumn widget in the Motif toolkit may be used as a popup menu, a pulldown menu, an option menu, a menu bar or a generic work area. This information is contained in the XmNrowColumnType resource. While we might be able to infer the capacity the XmRowColumn was acting in, the information is directly available in the widgets resources. Because of the potential power of a GetValues request, we decided to extend Editres to support it, with the hope that the MIT-supplied Editres will eventually support a similar request.

Another limitation of Editres that is related to its original design as a customization protocol is that it only supports polling. The information that Editres returns is only valid at the moment that it is sent to the Editres client. Changes in the widget hierarchy are not reported back to the Editres clients as they happen. Due to this limitation, we are required to watch for CreateWindow requests in the protocol stream and then request Editres to resend the widget hierarchy. This scenario exacerbates our flow control problems (see Deadlock Potential) and results in extraneous requests since there is no way to determine when the interface has stabilized.

Additionally, Editres uses ClientMessages and the selection mechanism to transmit requests and replies. While this transport mechanism is acceptable for a protocol intended for customization, it

Keyboard Input

Currently all input to the system is through the keyboard. Hence, since X "owns" the keyboard, all input to the system comes through the X server. It is the task of the Application Manager to translate keyboard input into appropriate events (including possibly mouse events) and routing those new events to the appropriate applications as a means of control. The Application Manager must also detect "meta controls;" that is, input which is designated for the Application Manager itself, rather than one of the applications running on the system.

There are some potential problems with input to the Application Manager which are related to the fact that all input to the system comes through the X server. The primary problem is that not all widgets solicit keyboard-related events and, in general, if a particular widget does not solicit a certain event type, the server will never generate that event. This implies that some widgets will not be "navigatable." That is, once the current context has changed to a widget that is not soliciting keyboard input, no further keyboard events will be generated from the server and it will be impossible to move out of that widget.

To work around this problem we have a set of rules which instruct the protocol interest manager to modify the X protocol stream in certain ways. First, whenever a CreateWindow request is generated, we intercept and modify the request to ensure that the newly-created window has an event mask attribute which will cause keyboard events to be generated. Second, we monitor the protocol for ChangeWindowAttribute requests, and modify these requests if need be to ensure that a client does not "turn off" solicitation for keyboard events at some later point in time. The Application Manager stores the "natural" state of event solicitation in the Model Manager, so that extraneous keypresses will never be sent to windows which are not supposed to receive keyboard input. This approach seems to work quite well in practice.

Of course, not all input is directed to the Application Manager. Some input is intended for the applications themselves. Since the PIM has full control over the X protocol stream, we can simply remove events which are intended for the Application Manager. Thus, the "reserved" keys which are used for navigation are never seen by applications. Other keyboard events are passed through to the applications unmodified. Note that under our approach any standard application keyboard accelerators should continue to work.

We are currently beginning work on using other input streams (such as voice) which do not have to travel through the X server. We feel that the use of multiple input sources will greatly improve the human-computer bandwidth of the interface. We are also beginning to investigate the use of the X Input Extension as a means of accepting input from novel input devices.

Bypassing Unwanted Objects

The concept of the user's context is maintained by keeping a pointer to the user's position in the application's widget hierarchy which is stored in the off-screen model by the Model Manager. The user navigates the application interface by moving throughout this tree structure. But the granularity of the user's movements is in terms of interface objects not X widgets. For this reason, many of the X widgets in the tree structure are skipped during the traversal process. The strategy is to place the user in the most specific widget which corresponds to the current interface object.

Our second assumption is that all text in a window is drawn using the same font. This constraint eliminates some of the same bookkeeping needed for proportional fonts and appears to be a fairly valid assumption in general, though it is not always true.

Using these constraints we are able to model text similarly to text on a terminal. Each window containing text is divided into a grid based on the text's font and is stored and manipulated based on this grid. This model supports CopyArea and ClearArea requests since we can easily map the rectangular coordinates supplied in these requests onto the grid.

The assumptions we made for handling text were acceptable for our prototype system but the use of proportional fonts and the mixing of fonts are important issues which must be resolved in future systems.

Sound Servers

As mentioned earlier, output to the user is generated by the specific rules fired by the Rules Engine. When fired, rules can invoke methods on the Sound Manager object (see Figure 1) which then generates the specified auditory output. The Sound Manager provides a single programmatic interface to the various sound resources available on our platform.

Currently, we use three types of sound output: speech, simple non-speech audio, and high-quality spatialized (3-D) sound. Each type of sound output is generated by different hardware, and thus there is a separate server process for each which regulates access to the sound hardware. These servers support access via multiple processes to workstation sound resources and allow sound hardware to reside on other machines across a network. See the Status section for more details on the particulars of these servers.

User Input

In addition to presenting application interfaces in a non-visual modality, the Application Manager is responsible for processing user input to both the applications and to the Application Manager itself. In essence, the Application Manager must *run* the existing application based on user input in the new interface. This section presents the mechanisms used for user input.

The Notion of Context

The Application Manager maintains the notion of the user's *current context* in the environment as a construct to support navigation. The current context is defined as the application the user is currently working in, and a particular widget within that application where the user is currently "located." Users navigate through applications by changing their current context and by operating widgets at their current location. The Application Manager effects changes in context by actually warping the mouse pointer to the desired location in response to user input. We warp the pointer to ensure that applications in our environment behave exactly as they would in a visual environment if the user moved the mouse pointer to a new widget.

Templates

To obviate the need to install a large number of rules to deal with the different semantics of each individual widget in the interface, 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 in the original interface. Templates provide consistency in the “hear and feel” of objects in Mercator interfaces.

For example, the rule template for widgets of the PushButton class may provide a set of rules to be fired whenever the PushButton instance is selected, 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 that particular PushButton instance which governs its interaction in the new interface. Rule templates provide a mechanism for ensuring standard behavior across all instances of a particular widget class with little work.

Text Handling

Computer interfaces are still mostly comprised of text. Early screen-reader systems for text-based interfaces (such as command line interfaces) accessed the screen contents by reading the ASCII codes stored in the display’s framebuffer. In these systems, the spatial relationships between pieces of text on the screen was well-defined since the text was laid out in a grid. Since the basic display unit of GUIs is the pixel, capturing text in a GUI presents a different set of problems.

First, text drawn in a GUI is stored as pixels in the framebuffer, which would be difficult to interpret as characters. Therefore the text must be captured before it is drawn. In Mercator this is accomplished by intercepting the text drawing requests, ImageText and PolyText. These drawing requests are the only means of drawing text under X unless the application renders text itself a pixel at a time. The text contained in these requests does not correspond directly to the text that will appear on the screen. All text under X is drawn using a font, with the characters acting as indices into the font. Thus the character “A” may be drawn as “A” in most fonts, but may appear as a special symbol in others. Thus we must not only keep track of text, but what font it was drawn in.

Another problem with handling text in a GUI is that everything is drawn at pixel locations, so it is difficult to determine the effects of overwriting text with new text or graphics. Likewise, graphics requests are used to copy and erase text, so we need to accurately model the location and movement of text at the pixel level.

To solve these problems, our present implementation operates under two assumptions concerning the display of text. First, applications are not allowed to use proportional fonts to draw text. This constraint simplifies the bookkeeping for handling text since we can assume that each character in a font takes up a fixed amount of space. This constraint is also easily configurable through the use of the resource database and enforceable by intercepting OpenFont requests.

Rules Engine

The heart of the Mercator Applications Manager is the Rules Engine (see Figure 1). The Rules Engine takes as input the current state and structure of the interface, and any user input, and produces auditory output according to a set of translation rules. From a high-level standpoint, the Rules Engine has two responsibilities: to present the application interface to the user, and to respond to input from the user and generate controlling events to the application and possibly the Application Manager itself. This section deals with the first goal, presenting the interface. See "User Input" below for a description of how we handle user input.

The Rules Engine is driven asynchronously by the Protocol Interest Manager, which taps the connection between the X client and the server. The Rules Engine informs the PIM of patterns in the X protocol which should cause control to be passed to the Rules Engine (that is, the Rules Engine expresses a "protocol interest," hence the name of the Protocol Interest Manager). When the Rules Engine is "awakened" by the PIM, it examines the current state of the protocol and may fire one or more rules, which may take any of a number of actions, including of course the production of auditory output.

The facilities available to translation rules are quite complex. For example, rules can stall the X protocol stream (that is, queue requests and events at the PIM for later delivery). Further, rules can actually insert new requests or events into the protocol stream, change existing requests or events before they are delivered, or delete requests or events. When packets are inserted or deleted the PIM will rewrite later packets to ensure that sequence numbers remain consistent.

Rules can examine or update the interface representation stored in the Model Manager. They may also query applications and collect replies via Editres. Information returned via Editres is used to update the off-screen model of the interface's state.

Rules perform a number of functions in our system. Some rules are used only for "maintenance" of the interface representation. For example, when new windows are created, a rule fires which updates the window hierarchy representation in the Model Manager. If the request to create the window came from a client which understands the Editres protocol (an attribute which is stored in the Client objects in the Model Manager) then a second rule will fire to retrieve an updated copy of the application's widget hierarchy in an attempt to match the newly created window to an actual widget.

Other rules actually create the auditory presentation of the interface. In the window creation example above, a rule would also fire which would examine the state of the interface and the newly created object and generate some auditory output to inform the user that, for example, a dialog box has just been created.

Our rules are currently implemented in a stylized C++ predicate/action notation. Rule list traversal is quite fast: predicates are preprocessed and organized into hash tables based on protocol interest patterns (which form the basis of rule predicates). Thus, when a particular protocol sequence occurs, the system does not need to test the predicates of all rules, just the ones that have the specified protocol pattern as a predicate.

ciated with the object. When XtObj instances are created, they also install “back-pointers” into the Model Manager’s Win objects so that it is easy to locate a particular widget based on a window ID. As a convenience, Client objects also maintain a list of the top-level windows of their application.

There are a number of benefits to our approach to interface modeling. Because of the way our data structures are keyed, it is easy to determine which widget or widgets correspond to a window ID in the protocol stream. As an example, when we see a window being mapped in the protocol, we can use the window ID to determine the widget in the interface which has just been popped up. The information stored in the Model Manager allows us to interpret the X protocol to associate higher-level semantic meaning to occurrences in the interface.

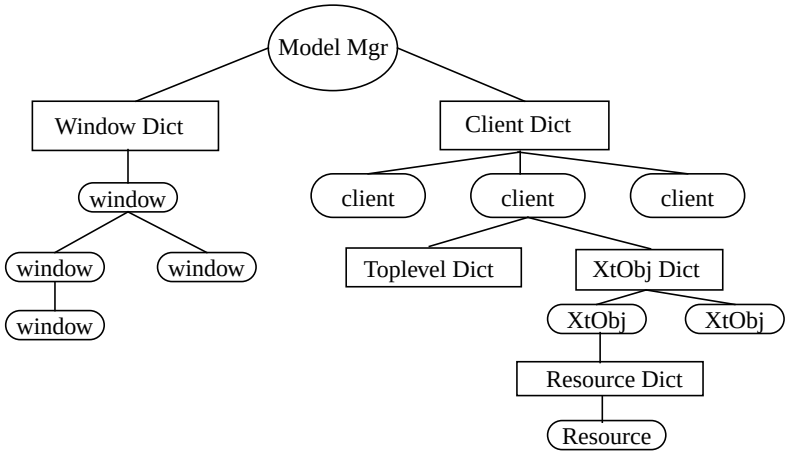

FIGURE 2: MODEL MANAGER DATA STRUCTURES

Application Output

Once we have captured information about the interface of a running application, and stored that information in our off-screen model, we still must present that interface structure to the user in a meaningful manner. To create an auditory presentation of the interface representation stored in our off-screen model, we apply a set of translations to the interface representation. These translations produce a consistent auditory interface which is semantically similar to our stored representation of the application’s original graphical interface. This section describes the particular mechanisms we use to accomplish this translation.

are notified of changes in the state of the application via the X protocol, and can then use Editres to understand and interpret the meaning of the protocol in terms of high-level constructs such as buttons, menus, and scrollbars.

Modeling Applications

Based on the information retrieved from an application, we must be able to synthesize a coherent model of the application's interface. We need to keep this interface representation locally so that we can have information about the interface available to us at all times without having to constantly poll the application to retrieve needed data. Also, we can store the interface information locally in a convenient format which can provide easy access to particular data as needed.

This model must provide a representation for not only the structure of the interface, but also appearance attributes, behavioral attributes, and semantic attributes. Our modeling techniques must be sufficiently flexible to support representing these (and possibly other) attributes of the applications which run in our environment.

Figure 2 provides a structural overview of the data structures we use to model application interfaces. The data structures shown here are controlled by the Model Manager component of the Application Manager. The Model Manager provides the programmatic interface to the interface representation for the rest of the Application Manager.

Since many clients may not implement Editres, the only information which is always *guaranteed* to be available to us is information present in the X protocol stream. The most important structural attribute of interfaces which is present in the protocol is the window hierarchy of the application. Thus our data structures provide a representation of this lowest common denominator of all X applications, the window. The Model Manager maintains a dictionary (key-value mapping) of Win objects which mirror the actual window hierarchy present on the display server. Win objects are our internal representation of relevant window attributes. This dictionary is updated automatically by the Protocol Interest Manager as windows are created, destroyed, or modified on the display server.

Of course the problem with modeling windows alone is that, by themselves, windows are too low-level to be meaningful when trying to understand the semantics of an application's interface. To overcome some of the limitations of representing the window hierarchy alone, we also maintain a representation of the interface objects (widgets or gadgets) in the interface on a per-client basis. Note that this information will only be available if the application in question understands the Editres protocol. The Model Manager keeps a list of Client objects, each of which represents one application running in the system. Client objects have unique identifiers based on the connection number of the application's protocol stream to the X server.

Client objects maintain the name and class of the application (sent as properties through the X protocol), and keep a dictionary which represents the interface object hierarchy of the application. Our internal representation for these interface objects is the XtObj class. Each XtObj instance models one widget or gadget in the application interface, and maintains information about parent and children object identifiers, the window ID of the object (if applicable), and any resources asso-

the window, the border of the window is changed (highlighted). Based solely on the low-level information present in the X protocol it would be extremely difficult if not impossible to derive the fact that the created object is a push button.

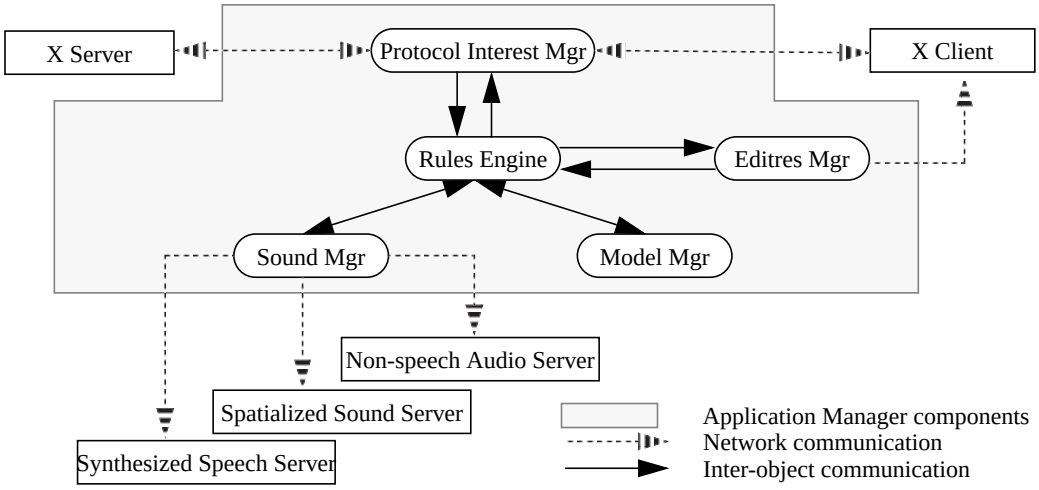

FIGURE 1: OVERVIEW OF THE APPLICATION MANAGER

Editres

To solve many of the problems which would be present if we relied on protocol monitoring alone, the Mercator Application Manager also makes use of the Editres protocol [Pet91]. The Editres protocol, introduced in X11R5, was designed to allow easy, interactive customization of applications. Editres provides facilities for querying running applications about their widget hierarchies, widget names and classes, and resources. This information is maintained by the Xt Intrinsics layer and is resident in the client itself; other than Editres there is no way for an external process to access this type of information in another client. The component of the Application Manager which implements the Editres protocol is called the Editres Manager (see Figure 1).

We use the information which can be obtained via Editres to interpret the X protocol stream. In our push button example above, when we detect via the protocol that a new window has been created and mapped, we generate an Editres query to the application to retrieve the widget hierarchy of the application. Contained in the reply is the name, class, widget ID, and window ID of all of the widgets and gadgets in the application. By matching the returned window IDs to the ID of the new window in the protocol, we can determine that indeed a push button has just been created.

Combining the semantically high-level information from Editres with the all-inclusive protocol-based information gives us a good picture of what is going on inside the application interface. We

There are a number of possible approaches to providing this level of information about application interfaces. One class of approaches involves changing the applications themselves, or the toolkits the applications are built with. By making direct modifications to application or toolkit code it is possible to introduce the mechanisms for non-visual interfaces directly and efficiently into applications; the code which produces the non-visual interface has available to it high-level information about the original interface and its behavior since it is "inside" the application. Of course, the drawback to this approach is that it requires modifications to applications and toolkits, and thus is not a transparent solution: it will not work with existing systems without recompilation (or at the least, relinking).

We have taken a different approach to the problem of information capture. Our system provides a non-visual interface to existing, unmodified X applications by using an external process to capture information about the interface at run-time. Our approach has the advantage that it is completely transparent: client applications and the window server do not (and indeed *cannot*) know that our alternative interface system is in use. The chief drawback to our approach is that since our system exists outside of the applications and window system, it has only a limited amount of information available about the original graphical interfaces it is attempting to translate.

This section presents the two mechanisms we are using to retrieve interface information from running applications. Both of these mechanisms operate transparently to both the applications and the window system.

Protocol Monitoring

The most basic technique we use to capture interface information is to monitor the exchange of X protocol packets between the X server and clients. The component of the Mercator Application Manager which accomplishes this protocol monitoring is called the Protocol Interest Manager. See Figure 1 for a structural overview of the Application Manager.

The Protocol Interest Manager, or PIM, is in essence a *pseudo-server*. That is, it is a separate process which, to clients, appears to be an X Window System server. To the server itself, the PIM appears to be a client. Clients connect directly to the PIM in our system and then the PIM establishes a surrogate connection to the "true" X server. Applications cannot distinguish the PIM from a real X server.

By tapping the connection between X clients and the server, the PIM is in a position to be notified of all changes in the interface: all window creations, maps, and unmaps, text and graphics rendering, and size changes are represented in the protocol. The X protocol is the only mechanism for drawing to the screen (since the screen is owned by the X server itself in most cases), and thus it is essentially impossible to circumvent the X protocol to draw to the screen. The PIM is aware of any and all application output to the screen.

But even though monitoring the protocol can tell us in complete detail *what* has happened in the interface, it cannot tell us *why* something has happened. The information in the X protocol is extremely low-level and provides virtually no semantic clues as to the meaning of events in the protocol. For example, when a push button is created, a sequence of protocol will be generated which creates and maps a window, and draws some text into it. When the mouse is moved into

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.

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 control and navigation allows the user to safely scan the interface without activating interface controls.

Architectural Overview

The previous sections of the paper discussed the motivation and design for a non-visual interface which is based on the semantic constructs in the original graphical interface of an application . We have also presented a keyboard-based navigation paradigm which allows users to scan and operate the interface. Now we shall discuss the architecture we have constructed to provide this alternative interface presentation and navigation. The component of the Mercator project which performs this translation of X-based graphical interfaces to auditory interfaces is called the Application Manager.

The Application Manager itself is composed of several components, and we shall discuss each of those components in turn. The remainder of this paper presents the tasks which must be performed by the Application Manager and the components of the system which accomplish these tasks.

Capturing Interface Information

The most important technical obstacle which must be overcome in order to provide an alternative interface to existing applications is the problem of how to capture or retrieve information about the interfaces of applications as they are running. A system which provides an alternative interface to existing applications must ideally have access to the semantic properties of the original interface (for example, the types of objects in the interface and the structure of the interface), as well as the behavioral attributes of the interface ("clicking this button causes a dialog box to pop up"). Lower-level "syntactic" information may also be needed ("this text is in boldface, this button is red").

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 can be used to convey some 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

TABLE 1: USING FILTEARS TO CONVEY AIC ATTRIBUTES

Navigation Paradigms

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

so many items) would need to be based on the auditory presentation of the scrollable object not its visual presentation.

Essentially, we have chosen a compromise between the two extremes outlined above. To ensure compatibility between visual and nonvisual interfaces, we are translating the interface at the level of the interface objects which form the user's model of the application interface. 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.

Another design consideration is which nonvisual interface modality to use. The obvious choices are auditory and tactile. We are currently basing our design on previous work in auditory interfaces which has demonstrated that complex auditory interfaces are usable [BGB91][Edwa89]. Another factor that we considered is that a significant portion of people who are blind also suffer from diabetes which may cause a reduction in their sensitivity to tactile stimuli [HTAP90]. Nevertheless our system will eventually have tactile components as well. For example, a braille terminal provides an alternate means for conveying textual information which may be preferred to speech synthesis.

Interface Components

As stated in the last section, the auditory presentation will be based on the user's model of the application interface. This model will be composed of the objects that make up the graphical presentation. The constructs of the user's model can be thought of in terms of common interface objects such as menus, buttons, and dialog boxes or in terms of the functions afforded by these objects such as "selection from a set" or "containing heterogenous types of information". In X Windows applications, these objects 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.

In Mercator, we call the objects in our auditory presentation Auditory Interface Components, or AICs. The translation from graphical interface components to AICs occurs at the widget level. The Mercator Model Manager stores the widget hierarchy for the application interface and any attribute information associated with the widgets. Using a set of rules and widget templates, these widgets are combined to form abstract interface objects.

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.

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

ent access to applications, we need to build a framework which will allow us to monitor, model and translate graphical interfaces of X Window System applications without modifying the applications. Second, given these application models, we need to develop a methodology for translating graphical interfaces into nonvisual interfaces.

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.

Next we describe the architecture we have constructed to provide this interface transparently for X 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.

Auditory Interfaces

The primary human-interface 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 Audio GUIs. We then describe the presentation of common interface objects such as buttons, windows, and menus, and detail the navigation paradigm for Mercator interfaces.

Design Considerations

There are several design decisions we had to make when constructing our nonvisual interface. A 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. The user could move a mouse over a screen and hear the objects announced as the cursor touched the object. In this type of system the user 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, the system could 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.

The primary question here is determining what aspects of the graphical interface contribute to the user's model of the application and what aspects of the graphical interface are simply visual artifacts of its graphical presentation. Retaining the user's model of the application interface across presentations is necessary to support collaboration between sighted and non-sighted users. It is also important to remove visual artifacts of the graphical presentation which do not make sense in the auditory domain. For example, scrollbars serve two purposes in graphical interfaces. First they conserve limited screen real estate by presenting only a portion of a list. Second, they provide a mechanism for the user to quickly search the list. The first use of the scrollbar most likely does not need to be conveyed in the auditory interface since there is no limited display real estate. The second use does need to be transferred to the auditory domain. But the movement of the scrollbar (up or down

Introduction

A common design principle for building computer applications is to separate the design and implementation of the application functionality from the application interface. The reasons for this separation are clear. If the application interface, or more precisely, the presentation of the application interface is independent of the application behavior then the interface can be easily modified to suit the needs of a wide range of users.

The graphical user interface is at this time an extremely common vehicle for presenting a human-computer interface. There are many times, however, when a graphical user interface is inappropriate or unusable. One example is when the task requires that the user's visual attention is directed somewhere other than the computer screen. Another example is when the computer user is blind or visually-impaired. Unfortunately, graphical user interfaces, or GUIs, have disenfranchised this portion 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]. The motivation for this legislation is clear. As more organizations move to a standard graphical environment, computer users who are visually-impaired may lose employment. Although this legislation has yet to be tested in the courts, both vendors and consumers of computer equipment and software are beginning to seriously address accessibility concerns.

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† Project, is to provide *transparent* access to X Windows applications for computer users who are blind or severely visually-impaired [ME92a][ME92b]. In order to achieve this goal, we need to solve two major problems. First, in order to provide transpar-

---

† 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 graphical display to an auditory display of the same user interface.

Published in *The X Resource*, #7, Summer, 1993. O'Reilly Publishers, Sebastopol, CA.

THE MERCATOR PROJECT

A NON-VISUAL INTERFACE TO THE X WINDOW SYSTEM

W. Keith Edwards, Elizabeth D. Mynatt, and Tom Rodriguez

ABSTRACT

This paper describes work to provide mappings between X-based graphical user 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 user interfaces for users who are blind or severely visually impaired. We describe the features of an auditory interface which simulates many of the characteristics of graphical interfaces. We then describe the architecture we have built to model and transform graphical interfaces. We present some of the difficulties encountered in building such a system on top of X. Finally, we conclude with some indications of future work.

Keith Edwards is a Research Assistant at the Georgia Tech Multimedia Computing Group. His research interests focus on computer-supported cooperative work. His email address is keith@cc.gatech.edu.

Beth Mynatt is a Research Scientist at Georgia Tech where she manages the Multimedia Computing Group. Her interests include auditory interfaces and novel interaction techniques. Her email address is beth@cc.gatech.edu.

Tom Rodriguez is a Research Assistant at the Multimedia Computing Group. His interests include window systems and digital video. His email address is jack@cc.gatech.edu.