Access

for Blind Users

The crisis was imminent. Graphical user interfaces were quickly adopted by the sighted community as a more intuitive interface. Ironically, these interfaces were deemed more accessible by the sighted population because they seemed approachable for novice computer users. The danger was tangible in the forms of lost jobs, barriers to education, and the simple frustration of being left behind as the computer industry charged ahead.

About the Authors:

KEITH EDWARDS *is a research assistant and Ph.D. candidate in the Graphics, Visualization, and Usability Center at Georgia Tech.*

ELIZABETH MYNATT *is a Research Scientist in the Graphics, Visualization and Usability Center at Georgia Tech. She directs the Multimedia Computing Group, which focuses on research in auditory and collaborative interfaces.*

KATHRYN STOCKTON *graduated in 1994 from Georgia Tech with a Master's in Computer Science and is currently working in Portland, Oregon for Metheus Corporation.*

Much has changed since that article was published. Commercial screen reader interfaces now exist for two of the three main graphical environments. Some feel that the crisis has been adverted, that the danger is now diminished. But what about the opportunity? Have graphical user interfaces improved the lives of blind computer users? The simple answer is not very much.

This opportunity has not been realized because current screen reader technology provides access to graphical screens, not graphical interfaces. In this paper, we discuss the historical reasons for this mismatch as well as analyze the contents of graphical user interfaces. Next, we describe one possible way for a blind user to interact with a graphical user interface, independent of its presentation on the screen. We conclude by describing the components of a software architecture which can capture and model a graphical user interface for presentation to a blind computer user.

Accessing Interfaces

The design of screen readers for graphical interfaces is centered around one goal: allowing a blind user to work with a graphical application in an efficient and intuitive manner. There are a number of practical constraints which must be addressed in the design. First, collaboration between blind and sighted users must be supported. Blind users do not work in isolation and therefore their interaction with the computer must closely model the interaction which sighted users experience. A second, and sometime competing, goal is that the blind user's interaction be intuitive and efficient. Both social and pragmatic pressures require that blind users not be viewed as second class citizens based on their effectiveness with computers.

The careful balance between these two goals is often violated by screen readers which provide a blind user with a representation of the computer interface which is too visually-based. Essentially these systems provide access to the screen contents, not the application interface. The distinction between these two terms will be discussed at length later in this section. Suffice to say that the application interface is a collection of objects which are related to each other in different ways, and which allow a variety of operations to be performed by the user. The screen contents are merely a snapshot of the presentation of that interface which has been optimized for a visual, two dimensional display. Providing access to a graphical interface in terms of its screen contents forces the blind user to first understand how the interface has been visually displayed, and then translate that understanding into a mental model of the actual interface.

In this section, we will briefly describe graphical user interfaces, focusing on their potential benefits for sighted and nonsighted users. Next we will examine three historical reasons why screen reader technology has not adapted sufficiently to the challenge of providing access to graphical user interfaces. We will complete our argument by exploring the levels of abstraction which make up a graphical user interface.

The Power of GUIs

For much of their history, computers have been capable of presenting only textual and numeric data to users. Users reciprocated by specifying commands and data to computers in the form of text and numbers, which were usually typed into a keyboard. This method of interaction with computers was only adequate at best.

More recently, advances in computer power and display screen technology have brought about a revolution in methods of human-computer interaction for a large portion of the user population. The advent of so-called Graphical User Interfaces (or GUIs) has been usually well-received. In this section we examine some of the

defining characteristics of GUIs, and explore some of the traits that make them useful to the sighted population. This examination will motivate our design of a powerful interface for users with visual impairments.

As implemented today, most GUIs have several characteristics in common:

- The screen is divided into (possibly overlapping) regions called windows. These windows group related information together.
- An on-screen cursor is used to select and manipulate items on the display. This on-screen cursor is controlled by a physical pointing device, usually a mouse.
- Small pictographs, called icons, represent objects in the user's environment which may be manipulated by the user. A snapshot of a typical graphical user interface is shown in Figure 1.

GUIs are quite powerful for sighted users for a number of reasons. Perhaps, most importantly, there is a direct correlation between the objects and actions which the GUI supports and the user's mental model of what is actually taking place in the computer system. Such a system is often called a direct manipulation interface, since to effect changes in the computer's state, the user manipulates the on-screen objects to achieve the desired result. Contrast this design to textual interfaces in which there are often arbitrary mappings between commands, command syntax, and actual results. Direct manipulation interfaces are usually intuitive and easy to learn because they provide abstractions which are easy for users to understand. For example, in a direct manipulation system, users may copy a file by dragging an icon which "looks" like a file to it's destination "folder." Contrast this approach to a textual interface in which one may accomplish the same task via a command line such as "cp mydoc.tex ~keith/tex/docs." Of course, the syntax for the command line interface may vary widely from system to system.

In addition to direct manipulation, GUIs provide several other important benefits:

- They allow the user to see and work with different pieces of information at one time. Since windows group related information, it is easy for users to lay out their workspaces in a way that provides good

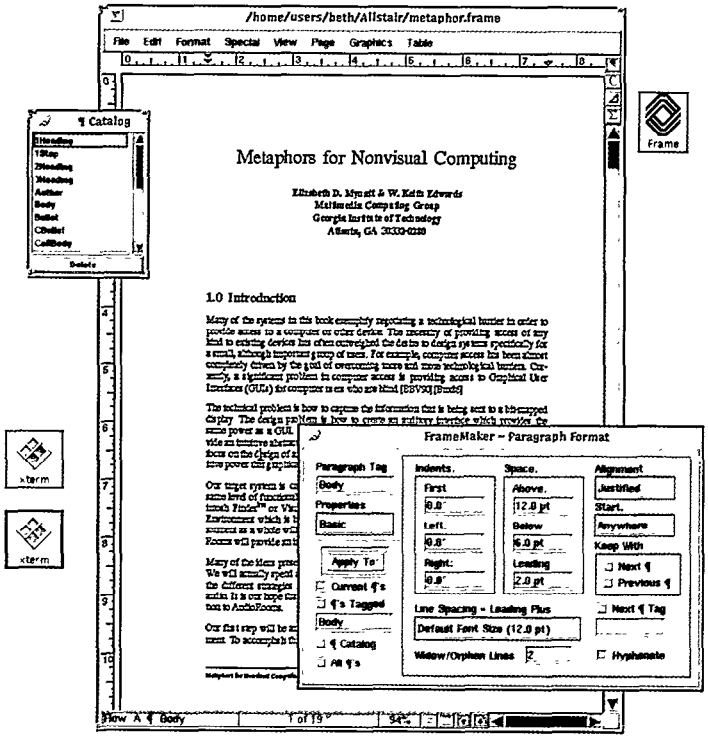

Figure 1
A typical graphical user interface

access to all needed information.

- An interface to multitasking is easily supported on most GUI-based systems. Each window provides a separate input/output point of control for each process which is running in the system. Processes continue running and users attend to the windows they choose.
- The graphical images used in GUIs lend themselves to the easy implementation of interface metaphors. The graphics support the metaphor by providing a natural mapping between metaphor and on-screen representation of the metaphor.

It is important to note that the power of graphical user interfaces lies not in their visual presentation, but in their ability to provide symbolic representations of objects which the user can manipulate in interesting ways.

Historical Reasons for Screen-Based Access

There are three major trends which help explain screen-based designs for accessing graphical interfaces. First, at one point in time, the screen contents closely equaled the application interface. The precursor to graphical interfaces were ASCII-based command-line interfaces. These interfaces presented output to the user one row at a time. Input to the interface was transmitted solely through the keyboard, again in a line-by-line manner. Screen reader systems for command line interfaces simply presented the contents of the screen in the same line by line manner, displaying the output via speech or braille. Input to the interface was the same for sighted and nonsighted users. In this scheme, both sighted and nonsighted users worked with the same interface - only the presentation of the interface varied. These strategies were sufficient as long as visual interfaces were constrained to 80 columns and 24 rows. However, the advent of the graphical user interface has made these strategies obsolete.

Second, reliance on translating the screen contents is caused, in part, by distrust of screen reader interfaces and concern about blind users not being able to use the same tools as sighted users. The general sentiment is that "I want to know what is on the screen because that is what my sighted colleague is working with." As concepts in graphical user interfaces became industry buzzwords, it was not uncommon to hear that blind users required screen readers that allowed them to use the mouse, drag and drop icons, and shuffle through overlapping windows. Although a popular notion in human-computer interface design is that the user is always right, it is interesting to compare these requirements with the requirements of sighted users who want auditory access to their computer. Current work in telephone-based interaction with computers allows a user to work with their desktop applications over the phone [10]. These interfaces perform many of the same functions that screen readers do - they allow the user to work with an auditory presentation of a graphical interface. Yet these system do not translate the contents of a graphical screen. Instead they provide an auditory interface to the same concepts conveyed in the graphical interfaces.

Third, limitations in software technology have driven the use of screen-based access systems. The typical scenario to providing access to a graphical application is that while the unmodified graphical application is running, an external program (or screen reader) collects information about the graphical interface by monitoring drawing requests sent to the screen. Typically these drawing requests contain only low-level information about the contents of the graphical interface. This information is generally limited to the visual presentation of the interface and does not represent the objects which are responsible for creating the interface and initiating the drawing requests.

Modeling Application Interfaces

At one level, an application interface can be thought of as a collection of lines, dots, and text on a computer screen. This level is the lexical interpretation of an interface: the underlying primitive tokens from which more meaningful constructs are assembled.

At a higher level, we can group these primitives into constructs such as buttons, text entry fields, scrollbars, and so forth. This level is the syntactic level of the interface. Lexical constructs (lines, text, dots) are combined into symbols which carry with them some meaning. While a line in itself may convey no information, a group of lines combined to form a push

button conveys the information, "I am pushable. If you push me some action will occur."

There is a still higher level though. At the highest level, we can describe an interface in terms of the operations it allows us to perform in an application. We might describe an interface in terms of the affordances [3] of the on-screen objects. For example, buttons simply provide a means to execute some command in the application; menus provide a list of possible commands, grouped together along some organizing construct; radio buttons provide a means to select from a group of settings which control some aspect of the application's behavior. It is the operators which the on-screen objects allow us to perform, not the objects themselves, which are important. This level is the semantic interpretation of the interface. At this level, we are dealing with what the syntactic constructs actually represent in a given context: these objects imply that the application will allow the user to take some action.

Seen from this standpoint, the most important characteristics of an application's interface are the set of actions the interface allows us to take, rather than how those actions are actually presented to the user on screen. Certainly we can imagine a number of different ways to capture the notion of "execute a command" rather than a simple push button metaphor represented graphically on a screen. In linguistic terms, the same semantic construct can be represented in a number of different syntactic ways.

This concept is the central notion behind providing access to graphical interfaces: rather than working with an application interface at the level of dots and lines, or even at the higher level of buttons and scrollbars, our goal is to work with the abstract operations which the application allows us to perform.

By divorcing ourselves from the low-level graphical presentation of the interface, we no longer constrain ourselves to presenting the individual graphical elements of the interface. By separating ourselves from the notion of graphical buttons and graphical scrollbars, we do away with interface objects which are merely artifacts of the graphical medium.

Does it make sense to translate application interfaces at the semantic level? Lines and dots on a screen, and even buttons and scrollbars on a screen, are simply one manifestation of the application's abstract interface. By translating the interface at the semantic level, we are free to choose presentations of application semantics which make the most sense in a nonvisual presentation.

...the most important characteristics of an application's interface are the set of actions the interface allows us to take, rather than how those actions are actually presented to the user on screen.

Certainly we could build a system which conveyed every single low-level lexical detail: "There is a line on the screen with endpoints and ." The utility of such an approach is questionable, although some commercial screen readers do construct interfaces in a similar manner.

Alternatively, we could apply some heuristics to search out the syntactic constructs on the screen: "There is a push button on the screen at location ." Certainly this method is better approach than conveying lexical information, although it is not ideal. Screen readers which use this method are taking the syntactic constructs of a graphical interface (themselves produced from the internal, abstract semantics of the actions the application affords), and mapping them directly into a nonvisual modality. Along with useful information comes much baggage that may not even make sense in a nonvisual presentation (occluded windows, scrollbars, and so forth, which are artifacts of the visual presentation). Certainly interacting with

such an interface is not as efficient as interacting directly with a presentation explicitly designed for the nonvisual medium.

We believe that transforming the application interface at the semantic level is the best approach for creating usable and efficient nonvisual interfaces. We can take the operations allowed by the application and present them directly in a non-visual form.

The question at this point is: are sighted and blind users working (and thinking) in terms of the same constructs? It is clear that they are if we translate the interface at the syntactic level. We argue that by constraining our semantic translation so that we produce "similar" objects in our non-visual presentation that the native application produces in its default graphical presentation, we maintain the user's model of the application interface. By giving things the same names (buttons, menus, windows), sighted and non-sighted users will have the same lexicon of terminology for referring to interface constructs.

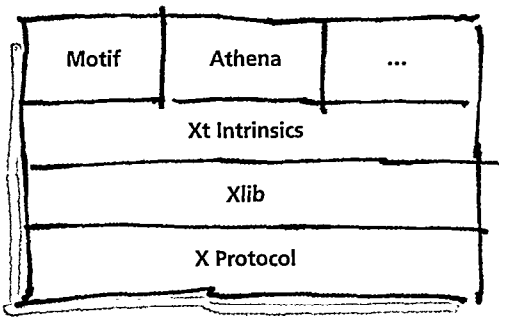

Figure 2
Layers in a typical X Window system application

Nonvisual Interaction With Graphical Interfaces

This section presents a set of implications for designers of nonvisual interfaces driven by our philosophy of translation at the semantic level. This discussion is presented in the context of the design of a particular nonvisual interface to provide access to graphical applications.

Auditory and Tactile Output of Symbolic Information

The first step in transforming a semantic model of a graphical interface into a nonvisual interface is to convey information about the individual objects which make up the interface. It is necessary to convey the type of the object (e.g. menu, push button), its attributes (e.g. highlighted, greyed out, size), and the operations it supports. Since the presentation of the objects is independent of its behavior, auditory and tactile output can be used as separate or complementary avenues for conveying information to the users. Our design focuses exclusively on the use of auditory output as a common denominator for North American users. Braille users will require additional, redundant braille output for textual information in the interface.

The objects in an application interface can be conveyed through the use of speech and nonspeech audio. Nonspeech audio, in the form of auditory icons [3] and filters [4], convey the type of an object and its attributes. For example, a text-entry field is represented by the sound of an old-fashioned typewriter, while a text field which is not editable (such as a error message bar) is represented by the sound of a printer. Likewise a toggle button is represented by the sound of a chain-pull light switch while a low pass (muffling) filter applied to that auditory icon can convey that the button is unavailable; that is, grayed out in the graphical interface. The auditory icons can also be modified to convey aspects of the interface which are presented spatially in the graphical interface such as the size of a menu or list. For example, all menus can be presented as a set of buttons which are evenly distributed along a set pitch range (such as 5 octaves on a piano). As the user moves from one menu button to another, the change in pitch will convey the relative size and current location in the menu. Finally, the labels on buttons, and any other textual information, can be read by the speech synthesizer.

In most screen reading systems, the screen reader will not have adequate access to the semantics of the application. To offset this problem, the screen reader must incorporate sematic information in the way that is models, and eventually presents, the graphical interface. The important concept is that symbolic information in the interface should be conveyed through symbolic representations which are intuitive for the user. By layering information in auditory cues, blind users interact with interface objects in the same way that sighted users interact with graphical objects.

Spatial versus Hierarchical Modeling of Object Relationships

The next step is to model the relationships between the objects which make up the application interface. Two principal types of relationships need to be conveyed to the users. First, parent-child relationships are common in

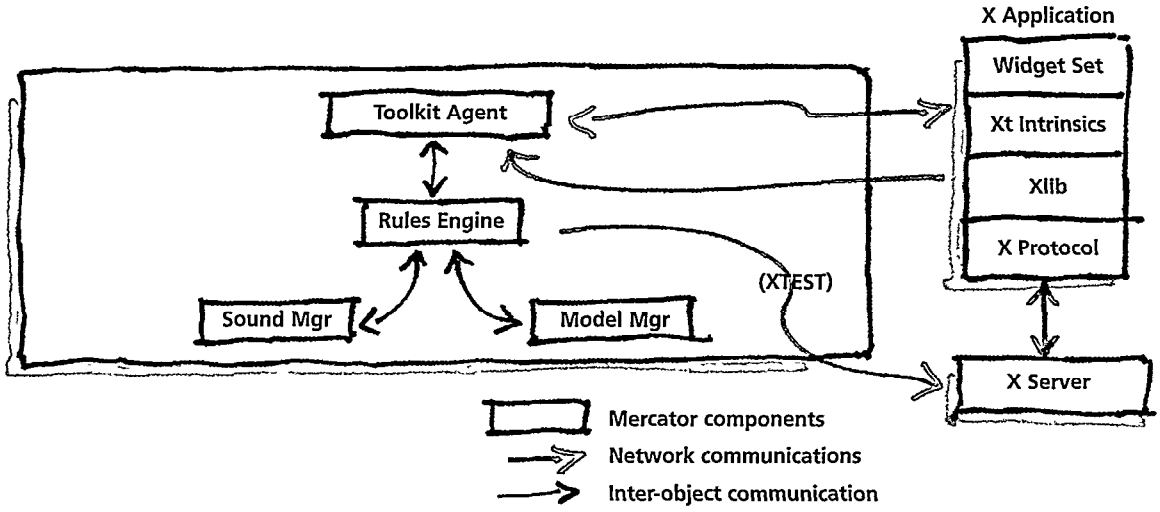

Figure 3
The architecture of Mercator

graphical interfaces. An object is a child of another object if that object is contained by the parent object, such as menu buttons which make up a menu, or a collection of objects which form the contents of a dialog box. In graphical interfaces these relationships are often conveyed by the spatial presentation of the graphical objects. Second, cause-effect relationships represent the dynamic portions of the graphical interface. For example, pushing a button makes a dialog box appear.

These relationships form the basis for navigating the application interface. Both of these relationships can be modeled with hierarchical structures. Parent-child relationships form the basis for the hierarchy, and cause and effect relationships are modeled by how they modify the parent-child object structure. Navigation is simply the act of moving from one object to another where the act of navigating the interface reinforces the mental model of the interface structure.

In short, information about the graphical interface is modeled in a tree-structure which represents the graphical objects in the interface (push buttons, menus, large text areas etc.) and the hierarchical relationships between those objects. The blind user's interaction is based on this hierarchical model. Therefore blind and sighted users share the same mental model of the application interface (interfaces are made up of objects which can be manipulated to perform actions) without contaminating the model with artifacts of the visual presentation such as occluded or iconified windows and other space saving techniques used by graphical interfaces. In general, the blind user is allowed to interact with the graphical interface independent of its spatial presentation.

At the simplest level, users navigate the interface by changing their position in the interface tree structure via keyboard input. Each movement (right, left, up or down arrow keys) positions the user at the corresponding object in the tree structure or informs the user, through an auditory cue, that there are no objects in the requested location. Additional keyboard commands allow the user to jump to different points in the tree structure. Likewise keyboard shortcuts native to the application as well as user-defined macros can be used to speed movement through the interface.

The hierarchical navigation model is extended to work in a multi-application environment. Essentially the user's desktop is a collection of tree structures. Users can quickly jump between applications while the system stores the focus for each application context. The user's current focus can also be used to control the presentation of changes to the application state. For example, a message window in an application interface may (minimally) use the following modes of operation:

- Always present new information via an auditory cue and synthesized speech.
- Signal new information via an auditory cue.
- Do not signal the presentation of new information.

These modes of operation can be combined in various ways depending on whether the application is the current focus. For example, an object can use one mode (always present via speech and/or nonspeech) when the application is the current focus and use another mode (signal via an auditory cue) when the application is not the current focus. Cues from applications which are not the current focus are preceded by a cue (speech or nonspeech) which identifies the sending applications.

Input Semantics and Syntax

We must also make a distinction, not only between the syntax and semantics of application output, but also between the syntax and semantics of application input. In a graphical interface, the semantic notion of "selection" (for example, activating a push button) may be accomplished by the syntactic input of double clicking the mouse on the on-screen push button. In the nonvisual medium we wish to preserve the input semantics (such as the notion of selection) while providing new input syntax which maps onto the semantics.

Our interfaces provide currently two input modalities: keyboard input and speech recognition. In the keyboard domain, the selection semantic is mapped to a keypress (currently the Enter key on the numeric keypad). Users who wish to perform selection via voice commands simply utter a keyword ("Select") which invokes the select action. The underlying mechanisms in the screen reader system take the input actions in the new modality and produce the syntactic input required to control the application.

An Architecture For X Window Access

We now present a system which implements the interface described above. This system, called Mercator, is designed to provide access to the X Window System [8]. The system is currently in its third major revision [5].

X is the de facto standard windowing system for Unix workstations. It is an open system controlled by the X Consortium, a vendor-neutral standards body. Figure 2 shows the layers of toolkits and libraries on which Xt-based applications are built. X is based on a client-server architecture, where X applications communicate with a display server via a network protocol. This protocol is the lowest layer of the X hierarchy. Xlib and the Xt Intrinsics provide two programming interfaces to the X protocol. Xlib provides the concept of events and provides support for drawing graphics and text. The Xt Intrinsics provide the concept of widgets (programmable interface objects) and provide a basic set of widgets. Most X applications are developed using libraries of widgets layered on top of the Intrinsics. Motif and Athena are two common widget sets.

At one extreme of the spectrum, it is possible to construct a system which is completely external to both the application and the window system.

The nonvisual interfaces produced by Mercator require high-level semantic information about the graphical interfaces of running applications. The system must be able to capture information from running (and unmodified) applications, maintain this information in a model of the application interface, and then transform the interface model to the new modality. Further, the system must be able to accept user input in new modalities and transform this input into the visually-oriented input expected by applications (mouse clicks, for example).

We now present a design space of potential solutions for information capture from running applications. Next, we discuss a set of modifications to the Xlib and Xt libraries which we have made and which have been accepted as a standard by the X Consortium. We describe how we store information about the application interface. Finally we describe how our system

implements input and output and maps from the graphical world into the nonvisual one.

A Spectrum of Solutions for Information Capture

How do we gather semantic information from running applications? How do we attain our goal of translating application interfaces at the semantic, rather than syntactic or lexical, level?

When we began our work we found that there is a spectrum of possible design choices for information capture. There are trade-offs between application transparency and the semantic level of the information available to us in this design space.

External Approaches.

At one extreme of the spectrum, it is possible to construct a system which is completely external to both the application and the window system. This point in the design space is essentially the approach taken by the initial version of Mercator: an external agent interposed itself between the client applications and the X Window System server. This approach has the advantage that it is completely transparent to both the application and to the window system. In the case of Mercator, the external agent appeared to the client to be an X server; to the "real" X server, Mercator appeared to be just another client application. There was no way for either to determine that they were being run in anything other than an "ordinary" environment.

This approach, while providing complete transparency, has a serious drawback however. Since we are interposing ourselves between the application and the window system, we can only access the information that would normally pass between these two entities. In the case of our target platform, the X Window System, this information is contained in the X Protocol which is exchanged between applications and the window server. While the X Protocol can describe any on-screen object (such as a button or a text area), it uses extremely low-level primitives to do so. Thus, while our system might detect that a sequence of lines was drawn to the screen, it was difficult to determine that these lines represented a button or some other on-screen object.

While the level of information captured by a system taking this approach depends on the particular platform, in general this method will provide only lexical information.

Our initial system did make use of another protocol called Editres [7] that allowed us to obtain some higher-level information about the actual structure of application interfaces. Thus, we could gain some information about interface syntax with which to interpret the lexical information available to us via the X Protocol. From our experiences, however, we determined that the level of information present in the X Protocol and Editres was insufficient to build a reliable and robust screen reader system.

Internal Approaches.

At the other extreme on the information capture spectrum, we can modify the internals of individual applications to produce non-visual interfaces. In this approach, the highest possible level of semantic information is available since in essence the application writer is building two complete interfaces (visual and non-visual) into his or her application. Of course the downside of this approach is that it is completely non-transparent: each application must be rewritten to produce a non-visual interface.

Obviously this approach is interesting as a reference point only. It is not practical for a "real world" solution.

Hybrid Approaches.

There is a third possible solution to the information capture problem which lies near the midpoint of the two alternatives discussed

At the other extreme on the information capture spectrum, we can modify the internals of individual applications to produce non-visual interfaces.

above however. In this solution, the underlying interface libraries and toolkits with which applications are written are modified to communicate information to an external agent which can implement the non-visual interface. This approach can potentially provide much more semantic information than the purely external approach: application programmers describe the semantics of the application interface in terms of the constructs provided by their interface toolkit. The interface toolkit then produces the actual on-screen syntax of these constructs.

The benefit of this strategy is that we do gain access to fairly high-level information. This approach cannot provide the level of semantic knowledge present in the purely internal strategy however, since the semantic level of information captured depends on the semantics provided by the toolkit library (and toolkits vary greatly in the semantic level of the constructs they provide). Still, for most platforms, toolkit modifications will provide access to enough useful information to accomplish a semantic translation of the interface.

The drawback of this approach is that, while it is transparent to the application programmer (that programmer just uses the interface toolkit as usual, unaware of the fact that the toolkit is providing information about the interface to some external agent), there must be a way to ensure that applications actually use the new library. Requiring all applications to be relinked against the new library is not feasible. Many systems support dynamic libraries, but this is not a practical solution for all platforms.

Rationale for Our Information Capture Strategy

During our use of the first version of Mercator it became clear that the protocol-level information we were intercepting was not sufficient to build a robust high-level model of application interfaces. Up until this point we had not seriously considered the hybrid approach of modifying the underlying X toolkits because of our stringent requirement for application transparency.

From our experiences with the initial prototype, we began to study a set of modifications to the Xt Intrinsics toolkit and the low-level Xlib library. These modifications could be used to pass interface information off to a variety of external agents, including not just agents to produce non-visual interfaces, but also testers, profilers, and dynamic application configuration tools.

Originally our intention was to build a modified Xt library which could be relinked into applications to provide access (either on a per-application basis, or on a system-wide basis for those platforms which support run-time linking). Through an exchange with the X Consortium, however, it became clear that the modifications we were proposing could be widely used by a number of applications. As a result, a somewhat modified version of our "hooks" into Xt and Xlib have become a part of the standard X11R6 release of the X Window System. A protocol, called RAP (Remote Access Protocol) uses these hooks to communicate changes in application state to the external agent. Figure 3 shows the architecture of the current system.

As a result of the adoption of our hooks by the X Consortium, our concerns with the transparency of this approach have been resolved. Essentially our hybrid approach has become an external approach: it is now possible to write non-visual interface agents which exist entirely externally to both the application and the window server, and only use the mechanisms provided by the platform.

Interface Modeling

Once Mercator has captured information about an application's interface, this information must be stored so that it is available for transformation to the nonvisual modality. Application interfaces are modeled in a data structure which maintains a tree for each client application. The nodes in this tree represent the individual widgets in the application. Widgets nodes store the attributes (or resources) associated with the widget (for example, foreground color, text in a label, currently selected item from a list).

There are three storage classes in Mercator: the Model Manager (whcih stores the state of the user's desktop in its entirety), Client (which stores the context associated with a single application), and XtObject (which stores the attrib-

Acknowledgements
This work has been sponsored by Sun Microsystems Laboratories and the NASA Marshall Space Flight Center. We are indebted to them for their support.

Graphics, Visualization, and Usability Center
College of Computing
Georgia Institute of Technology Atlanta, GA 30332-0280
keith@cc.gatech.edu,
beth@cc.gatech.edu,
kathryn@metheus.com

utes of an individual Xt widget). Each of these storage classes is stored in a hashed-access, in-core database for quick access. Each storage class has methods defined on it to dispatch events which arrive while the user's context is in that object. Thus it is possible to define bindings for events on a global, per-client, or per-object basis.

Other components of Mercator can access this data store at any time. A facility is provided to allow "conservative retrievals" from the data store. A data value marked as conservative indicates that an attempt to retrieve the value should result in the generation of a RAP message to the application to retrieve the most recent value as it is known to the application. This facility provides a "fail safe" in case certain widgets do not use the approved X Window System APIs to change their state.

Implementing Interfaces

The preceding sections of this paper described our strategies for information capture and storage from running X applications. Capturing and storing interface information is only a portion of the solution, however. A framework for coordinating input and output, and for presenting a consistent, usable, and compelling non-visual interface for applications is also required.

This section describes how our system creates effective non-visual interfaces based on the interface information captured using the techniques described above.

Rules for Translating Interfaces

We have designed our system to be as flexible as possible so that we can easily experiment with new non-visual interface paradigms. To this end, Mercator contains an embedded interpreter which dynamically constructs the non-visual interface as the graphical application runs. The auditory presentation of an application's graphical interface is generated on-the-fly by applying a set of transformation rules to the stored model of the application interface as the user interacts with the application.

These rules are expressed in an interpreted language and are solely responsible for creating the non-visual user interface. No interface code is located in the core of Mercator itself. This separation between the data capture and I/O mechanisms of the system from the interface rules makes it possible for us to easily tailor the system interface in response to user testing. The presence of rules in an easily-modifiable, human-readable form also makes customization of the system easy for users and administrators.

Our interpreted rules language is based on TCL (the Tool Command Language [6]), with extensions specific to Mercator. TCL is a lightweight language complete with data types such as lists and arrays, subroutines, and a variety of control flow primitives; Mercator rules have available to them all of the power of a general-purpose programming language.

When Mercator is first started, a base set of rules is loaded which provides some simple key-bindings, and the basic navigation paradigm. Each time a new application is started, Mercator detects the presence of the application, retrieves its name, and loads an applications-specific rule file if it exists. This allows an administrator or user to configure an interface according to their desires.

Event/Action Model

After start-up time, rules are fired in response to Mercator events. Mercator events represent either user input or a change in state of the application (as represented by a change in the interface model). Thus, we use a traditional event-processing structure, but extend the notion of the event to represent not just user-generated events, but also application-generated events. Events are bound to actions, which are interpreted procedures which are fired automatically whenever a particular event type occurs. Action lists are maintained at all levels of the storage hierarchy, so it is possible to change event-action bindings globally, on a per-client basis, or a per-widget basis.

As stated before, actions are fired due to either user input or a change in the state of the application. In the second case, we fire actions at the point the data model is changed, which ensures that the applications-generated actions are uniformly fired whenever Mercator is aware of the change. The call-out to actions occurs automatically whenever the data store is updat-

ed. This technique is reminiscent of access-oriented programming systems, in which changeing a system variable automatically triggers the execution of some code. [9].

System Output

All output to the user is generated through the interface fules. The "hard-coded" portions of Mercator do not implement any interface. This reliance on interpreted code to implement the interface makes it easy to experiment with new interface paradigms.

Interface rules generate output by invoking methods on the varous output objects in the system. Currently we support both speech and non-speech auditory output, and we are beginning to experiment with tactile output. The Speech object provides a "front-end" to a speech server which can be run on any machine on the network. This server is capable of converting text to speech using a number of user-definable voices. The Audio object provides a similar front-end to a non-speech audio server. The non-speech audio server is capable of mixing, filtering, and spatializing sound, in addition to a number of other effects. [2]

Both the Speech and the Audio objects are interruptible, which is a requirement in a highly interactive environment.

Simulating Input

Mercator provides new input modalities for users, just as it provides new output modalities. The mouse, the most commonly used input device for graphical applications, is inherently bound to the graphical display since it is a relative, rather than absolute positioning device (positioning requires spatial feedback, usually in the form on an on- screen cursor that tracks the mouse). Other devices may be more appropriate for users without the visual feedback channel. Our current interfaces favor keyboard and voice input over the mouse. We are also exploring other mechanisms for tactile input.

But while we provide new input devices to control applications, already existing applications expect to be controlled via mouse input. That is, applications are written to solicit events from the mouse device, and act accordingly whenever mouse input is received. To be able to drive existing applications we must map our new input modalities into the forms of input applications expect to receive.

User input handling can be conceptually divided into three stages. At the first stage, actual user input events are received by Mercator. These events may be X protocol events (in the case of key or button presses) or events from an external device or process (such as a braille keyboard or a speech recognition engine).

At the second stage, the low-level input events are passed up into the rules engine where they may cause action procedures to fire. The rules fired by the input may cause a variety of actions. Some of the rules may cause output to an external device or software process (for example, braille output or synthesized speech output), or a change in the internal state of Mercator itself (such as navigation). Some rules, however, will generate controlling input to the application. This input is passed through to the third stage.

At the third stage, Mercator synthesizes X protocol events to the application to control it. These events must be in an expected format for the given application. For example, to operate a menu widget, Mercator must generate a mouse button down event, mouse motion to the selected item, and a mouse button release when the cursor is over the desired item. Note that the actual event sequence which causes some action to take place in the application interface may be determined by user, application, and widget set defaults and preferences. Thus Mercator must be able to retrieve the event sequence each interface component expects to receive for a given action. This information is stored as a resource (called the translation table) in each widget and can be retrieved via the RAP protocol.

We currently use the XTEST X server extension to generate events to the application. This approach is robust and should work for all X applications.

Status

The hooks into the Xt and Xlib libraries have been implemented and are present in the X11R6 release from the X Consortium. The RAP protocol is currently not shipped with X11R6 pending a draft review process; we hope that in the near future RAP will ship with the

PERMISSION TO COPY WITHOUT FEE, ALL OR PART OF THIS MATERIAL IS GRANTED PROVIDED THAT THE COPIES ARE NOT MADE OR DISTRIBUTED FOR DIRECT COMMERCIAL ADVANTAGE, THE ACM COPYRIGHT NOTICE AND THE TITLE OF THE PUBLICATION AND ITS DATE APPEAR, AND NOTICE IS GIVEN THAT COPYING IS BY PERMISSION OF THE ASSOCIATION FOR COMPUTING MACHINERY. TO COPY OTHERWISE, OR PUBLISH, REQUIRES A FEE/AND OR SPECIFIC PERMISSION
© ACM 1072-5520/96/0100 $3.50

standard distribution of the X Window System.

The various components of Mercator are written in C++; the current core system is approximately 16,000 lines of code, not including I/O servers and device specific modules. Our implementation runs on Sun SPARCstations running either SunOS 4.1.3 or SunOS 5.3 (Solaris 2.3). Network-aware servers for both speech and non-speech audio have been implemented using Transport Independent Remote Procedure Calls (TI-RPC), with C++ wrappers around their interfaces.

The speech server supports the DECtalk hardware and the Centrigram TruVoice software-based text-to-speech system and provides multiple user-defined voices. The non-speech audio server controls access to the built-in workstation audio hardware and provides prioritized access, on-the-fly mixing, spatialization of multiple sound sources, room acoustics, and several filters and effects. The non-speech audio server will run on any SPARCstation, although a SPARCstation 10 or better is required for spatialization effects.

Speech input is based on the IN3 Voice Control System, from Command Corp, which is a software-only speech recognition system for Sun SPARCstations. The recognition server runs in conjunction with a tokenizer which generates input to the Mercator rules system based on recognized utterances.

Future Directions

There are several new directions we wish to pursue. These directions deal not only with the Mercator interface and implementation, but also with standards and commercialization issues.

From the interface standpoint, we will be performing more user studies to evaluate the non-visual interfaces produced by Mercator. Further testing is required to fully ensure that the interfaces produced by the system are usable, effective, and easy to learn.

Our implementation directions lie in the area of building a more efficient architecture for producing Mercator interfaces. Our current implementation is singly-threaded; we plan to investigate a multi-threaded architecture. We are also experimenting with a more refined I/O system in which input and output modalities can be more easily substituted for one another.

We are working with the X Consortium and the Disability Access Committee on X to ensure that the RAP protocol is adopted as a standard within the X community. It is our desire that any number of commercial screen reader products could be built on top of RAP.

We are exploring the possibilities of undertaking a commercialization effort of our own to bring our research prototype to market.

References

[1] Boyd L.H., Boyd W.L., and Vanderheiden G.C. *The Graphical Gser Interface: Crisis, Danger and Opportunity. Journal of Visual Impairment and Blindness* (December 1990) pages 496-502.

[2] Burgess, D. *Low Cost Sound Spatialization. In Proceedings of the 1992 ACM Symposium on User Interface Software and Technology* (November 1992). ACM, New York.

[3] Gaver, W.W. *The Sonicfinder: An Interface That Uses Auditory Icons. Human Computer Interaction* (1989), 4:67-94.

[4] Ludwig, L.F., and Cohen, M. *Multidimensional Audio Window Management. International Journal of Man-Machine Studies* (March 1991) 34:3, pages 319-336.

[5] Mynatt, E.D. and Edwards, W.K. *Mapping GUIs to Auditory Interfaces.* In *Proceedings of the 1992 ACM Symposium on User Interface Software and Technology* (November 1992). ACM, New York.

[6] Ousterhout, J.K. *TCL: An Embeddable Command Language.* In *the Proceedings of the 1990 Winter USENIX Conference,* pp. 133-146.

[7] Peterson, C.D. *Editres-A Graphical Resource Editor for X Toolkit Applications.* In *Conference Proceedings, Fifth Annual X Technical Conference* (January 1991), Boston, Massachusetts.

[8] Scheifler, R.W. *X Window System Protocol Specification, Version 11.* Massachusetts Institute of Technology, Cambridge, Massachusetts, and Digital Equipment Corporation, Maynard, Massachusetts (1987).

[9] Stefik, M.J., Bobrow, D.G., and Kahn, K.M. *Integrating Access-Oriented Programming into a Multiparadigm Environment. IEEE Software, 3, 1,* (January 1986), IEEE Press, pages 10-18.

[10] Yankelovich, N. *SpeechActs & The Design of Speech Interfaces.* In *the Adjunct Proceedings of the 1994 ACM Conference on Human Factors and Computing Systems,* Boston, MA (1994). ACM, New York.