An Architecture for Transforming Graphical Interfaces

W. Keith Edwards and Elizabeth D. Mynatt
Graphics, Visualization, and Usability Center
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
keith@cc.gatech.edu, beth@cc.gatech.edu

ABSTRACT
While graphical user interfaces have gained much popularity in recent years, there are situations when the need to use existing applications in a nonvisual modality is clear. Examples of such situations include the use of applications on hand-held devices with limited screen space (or even no screen space, as in the case of telephones), or users with visual impairments.

We have developed an architecture capable of transforming the graphical interfaces of existing applications into powerful and intuitive nonvisual interfaces. Our system, called Mercator, provides new input and output techniques for working in the nonvisual domain. Navigation is accomplished by traversing a hierarchical tree representation of the interface structure. Output is primarily auditory, although other output modalities (such as tactile) can be used as well. The mouse, an inherently visually-oriented device, is replaced by keyboard and voice interaction.

Our system is currently in its third major revision. We have gained insight into both the nonvisual interfaces presented by our system and the architecture necessary to construct such interfaces. This architecture uses several novel techniques to efficiently and flexibly map graphical interfaces into new modalities.

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

INTRODUCTION

The graphical user interface is, at this time, the most common vehicle for presenting a human-computer interface. There are times, however, when these interfaces are inappropriate. One example is when the task requires that the user's visual attention be directed somewhere other than the computer screen. Another example is when the computer user is blind or visually-impaired [BBV90][Bux86].

The goal of providing nonvisual access to graphical interfaces may sound like an oxymoron. The interface design issues of translating an interactive, spatially presented, visually-dense interface into an efficient, intuitive and non-intrusive nonvisual interface are numerous. Likewise, the software architecture issues of monitoring, modeling and translating unmodified graphical applications are equally complex.

The typical scenario to providing access to a graphical interface is as follows: While an unmodified graphical application is running, an outside agent (or screen reader) collects information about the application interface by watching objects drawn to the screen and by monitoring the application behavior. This screen reader then translates the graphical interface into a nonvisual interface, not only translating the graphical presentation into an nonvisual presentation, but providing different user input mechanisms as well.

During UIST 1992, we presented a set of strategies for mapping graphical interfaces into auditory interfaces primarily with the aim of providing access for blind users [ME92]. These strategies, implemented in a system called Mercator, demonstrated a scheme for monitoring X Windows [Sch87] applications transparently to both the applications and the X Windows environment. Guidelines for creating a complex auditory version of the graphical interface using auditory icons and hierarchical navigation were also introduced.

Much has happened since November 1992. Both formal and informal evaluations of the techniques used to create Mercator interfaces have offered new insights into the design of complex auditory interfaces. Since these interface techniques have generally been welcomed by the blind and sighted communities, they now form a set of requirements for screen reader systems.

More significantly, the entire architecture of Mercator has been replaced in response to experiences acquired in building the system as well as by the auditory interface requirements. The new architecture is based on *access hooks* located in the Xt Intrinsics and Xlib libraries. These hooks allow state changes in application interfaces to be communicated to outside agents such as screen readers, customization programs and testing programs. The Mercator project played a significant role in championing and designing these hooks which were accepted by the X

Consortium (a vendor-neutral body which controls the X standard) and released with X11R6.

In addition to modifying Mercator to use these new hooks, we have restructured Mercator to support a simplified event model which allows nonvisual interfaces to be loaded and customized in an extremely flexible manner. A litmus test of our work is that this architecture is sufficient to model and transform numerous X applications.

This paper is organized as follows. The following section summarizes the design of Mercator interfaces. It also briefly describes some of the modifications to the interfaces as last reported in this forum. The next section introduces the Mercator architecture and the design issues that have influenced the new implementation. We step through the construction of our system, highlighting the general applicability of our work to other user interface monitoring and manipulation tasks. We close by summarizing the status of our current system, introducing some of our future research goals and acknowledging the sponsors of our research.

MERCATOR INTERFACES

The design of Mercator interfaces is centered around one goal--allowing a blind user to work with a graphical application in an efficient and intuitive manner. Previous Mercator papers have discussed the preferred use of audio output for North American users, as well as the object model for the auditory interface [ME92]. 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 and so on) 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. This model is not contaminated 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.

The contents of the application interface are conveyed through the use of speech and nonspeech audio. The first Mercator system established the use of auditory icons [Gav89] and *filtears* [LC91] to 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 an 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 (this attribute may be conveyed by "graying out" in a graphical interface). Additional design work has led to the use of auditory cues to convey *hidden* information in the auditory interface, such as mapping a pitch range to the length of a menu [My94][MW94]. Finally, the label for that button, and any other textual information, can be read by a speech synthesizer.

At the simplest level, users navigate Mercator interfaces 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 or informs the user, through an auditory cue, that there are no objects at the requested location. Additional keyboard commands allow the user to jump directly 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 navigation model has been 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 application.

Mercator interfaces have undergone numerous informal evaluations by blind computer users. Generally, Mercator is demonstrated alongside commercial screen readers in the exhibit area of conferences devoted to technology and persons with disabilities. Feedback from blind users confirms that the hierarchical navigation and nonspeech auditory cues are intuitive and welcomed by the blind user community. This feedback is significant since Mercator is the only screen reader which uses these techniques, although versions of these techniques are now beginning to appear in other screen reader systems.

The capsule summary of the system requirements for the creation of Mercator interfaces cluster around three common themes.

- The construction of the interfaces must be based on the semantic organization of the application interface, not just its graphical presentation.

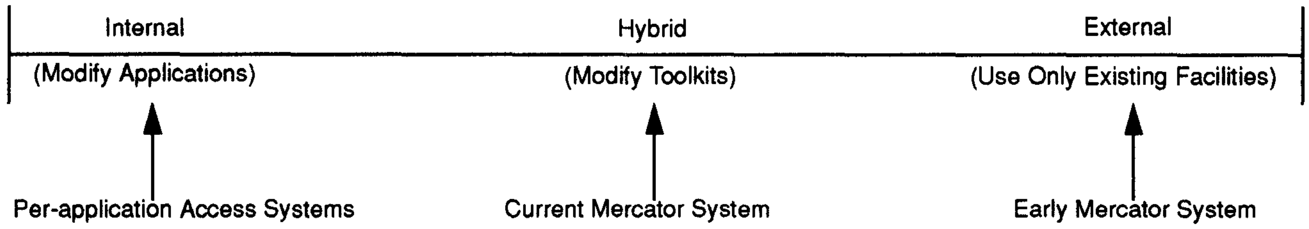


FIGURE 1. A Spectrum of Solutions for Information Capture

- The interfaces should be highly interactive and intuitive in the auditory space.
- The system must be able to generate efficient and compelling interfaces for a broad range of applications.

The remainder of this paper is devoted to describing the new architecture for Mercator, and how it is able to meet these requirements as well as providing a platform for other user interface monitoring and modeling tasks.

ARCHITECTURE

Given the user interface requirements described above, our goal was to build a software architecture capable of constructing these interfaces. Any architecture which is capable of producing such interfaces must address the following issues:

- The architecture must gather information with sufficient semantic content to support our object-based interaction model and yet be broadly applicable to a wide variety of applications.
- Our information capturing techniques must guarantee complete knowledge about the application's graphical interface.
- The architecture must support both a fine degree of control to configure the individual interfaces as well as coarse-grained control to explore the design space of nonvisual interfaces.
- Since we are providing new interaction techniques to control existing applications, we must support a separation between the semantic operations that the application provides and the syntactic grammar of user input.

In the following sections, we describe an architecture which addresses the aforementioned issues. This discussion explores the range of techniques for information capture that can provide varying degrees of semantic information about the graphical interface. Next, we detail the particular information capture strategy used by Mercator and discuss its applicability to other interface monitoring and modeling tasks.

Given the demands of fully monitoring interactive graphical applications, we describe the cooperation between the functional components of our system as it dispatches multiple forms of input from both the applications and the user. Next, we provide an overview of our modeling techniques for representing application interfaces and generic text, as well as strategies for handling potential inconsistencies in the data store.

We explain how our design allows us to provide highly flexible and dynamic nonvisual interfaces as well as a flexible and powerful event model which can represent user input as well as application output. Finally, we describe the input and output mechanisms in our system which support interactivity in the nonvisual domain, and provide a separation between syntax and semantics of application control.

Information Capture

When we began our work on this project, it became clear that there is, in fact, a spectrum of solutions for capturing information from applications. Along this spectrum we see essentially a trade-off between transparency and the semantic level of the information available to an external agent (see Figure 1).

At one extreme of the spectrum, we have the option of directly modifying every application so that it provides information about its state to the external agent. While this approach provides the highest possible degree of semantic information about what the application is doing, it is completely non-transparent: each application must be rewritten to be aware of the existence of the external agent. Obviously this end of the spectrum serves as a reference point only, and is not practical for a "real world" solution.

At the other extreme of the spectrum we can rely only on the facilities inherent in whatever platform the application was built on. In our design space, this platform meant the X Window System, specifically applications built using the Xlib and Xt libraries. This use of existing facilities is essentially the approach taken by the first version of Mercator. Our system interposed itself between the X server and the application and intercepted the low-level X protocol information on the client-server connection. This approach had the benefit that it was completely transparent to both the application and the X server (indeed, it was impossible for either to detect that they were not communicating with a "real" X client or server), but had a rather severe limitation: the information available using this approach was extremely low level. Essentially we had to construct a high-level structural model of the application from the low-level pixel-oriented information in the X protocol. Our first system also used the Editres widget customization protocol which appeared in X11R5 [Pet91], but we found that Editres was insufficient for all our needs. The use of these approaches was the only practical solution available to us in our first system, however, because of our requirement for application transparency.

There is a third possible solution strategy which lies near the middle point of these two extremes. In this strategy, the underlying libraries and toolkits with which the application is built are modified to communicate changes in application state to the external agent. This approach is not completely transparent--the libraries must be modified and applications relinked to use the new libraries--but all applications built with the modified libraries are accessible. The semantic level of information available to the external agent depends on the semantics provided by the toolkit library.

Modifications to Xt and Xlib

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. We began to study a set of changes to the Xt Intrinsics toolkit which could provide the information needed to support a variety of external agents, including not only auditory interface agents, 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. 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 are a part of the standard X11R6 release. A protocol, called RAP (Remote Access Protocol) uses these hooks to communicate changes in application state to an external agent. RAP also provides communication from the external agent to the application.

This section describes (in fairly X-specific terms) the design of the toolkit modifications which are present in X11R6. We feel that the modifications are fairly complete and can serve as a guideline for developers of other toolkits who wish to be able to communicate information about interface state changes to external agents. Further, these hooks can be used to implement all of the functionality of the Editres system used in our previous implementation; the new hooks and RAP subsume the configuration capabilities of Editres. Table 1 presents the basic messages in RAP.

Message	Description
GetResources	Retrieve the resources associated with a particular widget.
QueryTree	Retrieve the widget hierarchy of the application.
GetValues	Retrieve the values of a list of resources associated with a given widget.
SetValues	Change the values of a list of resources associated with a given widget.
AddCallback	"Turn on" a particular callback in the Hooks Object.
RemoveCallback	"Turn off" a particular callback in the Hooks Object.
ObjectToWindow	Map an object ID to a window ID.
WindowToObject	Map a window ID to an object ID.
LocateObject	Return the visible object that is under the specified X,Y location.
GetActions	Returns a list of actions for a widget.
DoAction	Invoke an action on a widget (may not be available in all implementations).
CloseConnection	Shut down the RAP connection.
Block	Stall the client so that an external agent can "catch up."
ObjectCreated	Inform an agent that a new widget has been created.
ObjectDestroyed	Inform an agent that a widget has been destroyed.
ValueChanged	Inform an agent that a resource has changed.
GeometryChanged	Inform an agent that a widget's geometry (size, position) has changed.
Configuration-Changed	Inform an agent that a widget's configuration (map/unmap state) has changed.

TABLE 1. Remote Access Protocol

The hooks consist of a new widget, called the Hook Object, which is private to Xt. The hook object maintains lists of callback procedures which will be called whenever widgets are created or destroyed, their attributes (resources) are changed, or their configuration or geometry is updated. Some bookkeeping data is also maintained in the Hook Object. A new API has been added to Xt which allows application programmers to retrieve the Hook Object associated with a connection to the X server.

All of the Xt Intrinsics routines which can create or destroy widgets, or modify widget state have been modified to call the appropriate callback functions that have been installed in the Hook Object. By default, no callbacks are installed in the Hook Object. Instead, a set of callbacks is installed in the Hook object when an application is initially contacted by an external agent such as Mercator. These callback routines implement the application-to-Mercator half of the RAP protocol which informs Mercator about changes in application state.

Protocol Setup. The Xaw Vendor Shell widget (essentially the "outer window" interface object) has been modified to support the initial "jump-start" phase of the connection setup protocol. External agents (including testers and profilers--

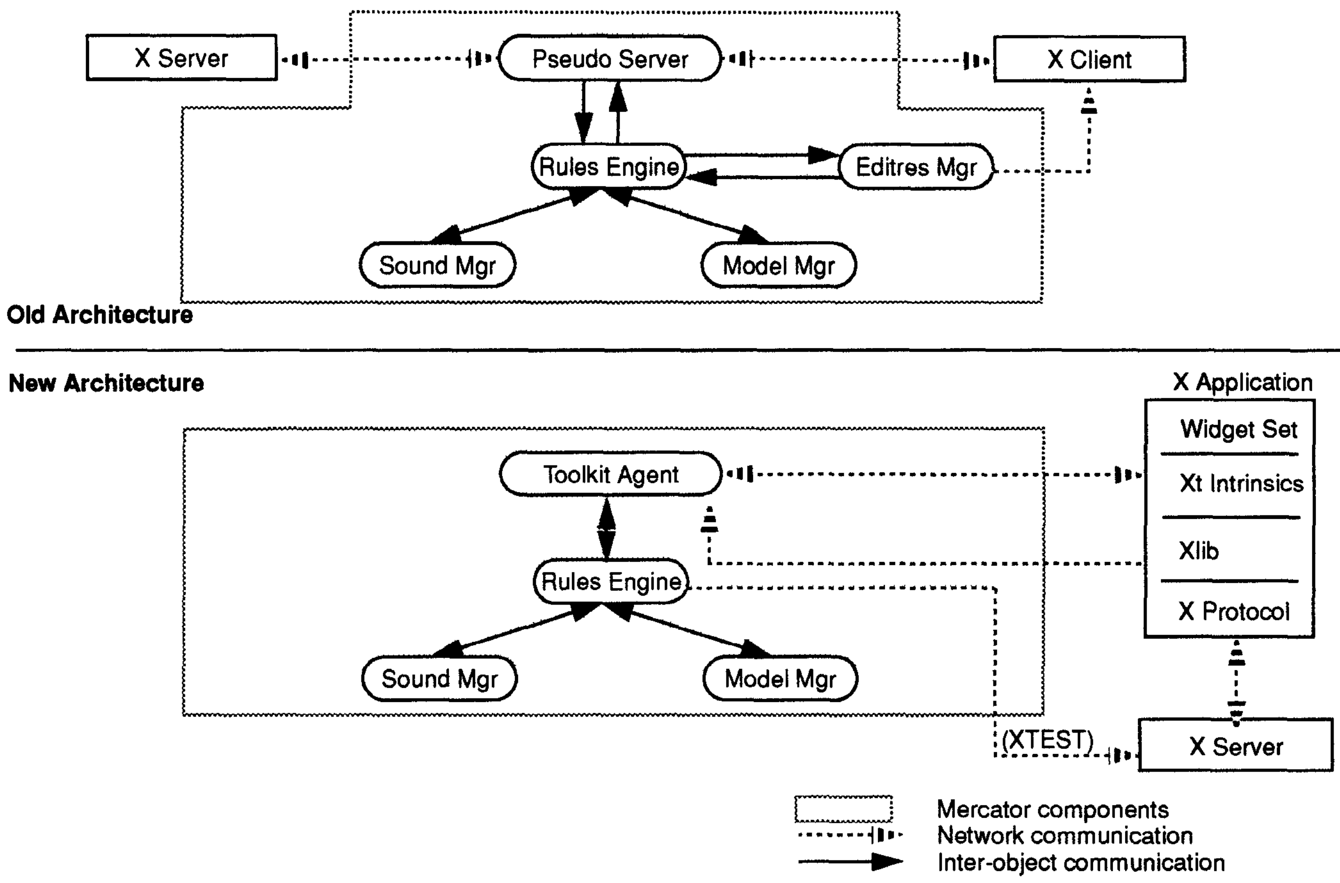


FIGURE 2. Old and New Mercator Architectures

not just Mercator) pass a message to an application via a selection. This message contains the name of the protocol the external agent wishes to speak. Code in the Vendor Shell catches the message, and searches a table for the named protocol. If found, an initializer routine will be called which will install the callback routines appropriate for that protocol in the Hooks Object. If the application needs to be able to receive messages from the external agent (in addition to simply passing information out via the hook callbacks), then it can create a communications port with an associated message handler for incoming messages from the external agent. This port is used for Mercator-to-application messages in the RAP protocol.

Replacing the Pseudoserver. We have also argued for a change to the lower-level Xlib library which has been adopted by the X Consortium for inclusion in X11R6. This change is an enhancement to the client-side extension mechanism in X which allows a function to optionally be called just before any X protocol packets are sent from the application to the server. A function can be installed in this "slot" by the protocol initializer which will pass to an external agent the actual X protocol information being generated by an application. This modification to Xlib serves as a "safety net" for catching any information which cannot be determined at the level of Xt. This hook in Xlib allows us to operate without the use of a pseudoserver system, unlike our previous implementation. Events from the server to the application are passed via a function installed in an already-extant client-side extension.

The modifications described above provide a general framework to allow a wide variety of external agents to cooperate closely with applications; these modifications consist of a number of small changes to Xt and Xlib which are, for the most part, invisible to applications which do not care about them. They allow protocols to be automatically installed and thus our goal of transparency has been achieved: any application built with the X11R6 libraries will be able to communicate with an external agent.

It is our conjecture that this foundation will be usable by other *generic* tools that will work across a wide array of applications. Although it may be possible to instrument a single application with, say, a profiling system, this infrastructure makes it possible to construct a generic profiler which will work across applications. Given the significant requirements that screen readers have for monitoring and interacting with graphical applications, it is reasonable to conclude that this architecture will be sufficient for less demanding tasks such as profiling, configuring or monitoring a graphical application. Since the system is built on standard X hooks which minimally impact the performance of a running application, this platform should be well-suited to other commercial and research endeavors.

Control Flow
Mercator must constantly monitor for changes in the states of the graphical applications as well as for user input. The new techniques for information capture mean that we no longer have to be slaved to the X protocol stream. In our earlier pseudoserver-based implementation, care had to constantly be taken to ensure that Mercator never blocked, since blocking would have stalled the X protocol stream and effectively deadlocked the system.

In our new system, using the Xlib hook, we have a more flexible control flow. We can generate protocol requests to the server at any time, and engage in potentially long computation without having to worry about deadlock. In our previous implementation a fairly complex callback system was constructed so that computation segments could be chained together to prevent potential deadlock situations. We avoid potential deadlocks in the new system because, unlike with the pseudoserver implementation, the client applications can continue running even if Mercator is blocked. The Xlib hook approach gives us the benefits of the pseudoserver approach--access to low-level information--without the costs associated with pseudoservers.

All components of Mercator which perform I/O are subclassed from a class called FDInterest. Each instance of this class represents a connection (over a file descriptor) to some external component of the system. For example, a separate FDInterest exists for each connection Mercator has to an application, each connection to an audio server, and so on. Each FDInterest is responsible for handling I/O to the entity it is connected to. This architecture makes the division of I/O responsibility much cleaner than in our older system.

Figure 2 shows the older pseudoserver-based Mercator architecture alongside the newer architecture which replaces the pseudoserver with the Xlib hook.

The overall system is driven by RAP messages to and from the client applications. For example, whenever an application changes state (pops up a dialog box, for example), a RAP message is generated from the application to Mercator. It is received in Mercator by the FDInterest connected to that client. The FDInterest determines the type of the message and dispatches it according to a hard-coded set of rules which keep our model of the application interface up-to-date. For example, if a new widget is created, the FDInterest generates commands to add the new widget, along with its attributes, to our model of the interface.

Interface Modeling
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. Widget 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 (which 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 attributes 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 objects in 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 provides a further safety feature in case certain widgets do not use approved APIs to change their state.

Text is stored in a special entity called a TextRep object. TextReps are created automatically whenever text is drawn to a window for the first time. TextReps are associated with objects in the data store and can be accessed by other components of Mercator to retrieve an up-to-date account of the text present in a given window. The Xlib hook keeps this information current; the text model maintains consistency over scrolling, font changes, and refreshes.

Embedded Computation to Dynamically Construct Interfaces
One of the more novel concepts in the Mercator implementation is its use of embedded interpreters to dynamically build the interface "on the fly" as the user is interacting with the application. Unlike graphical interfaces, where there is a constant, usually-static, presentation of the interface on the screen, auditory interfaces are much more dynamic. In Mercator, the auditory presentation for a given object is generated at run-time by applying a set of transformation rules to the application model. These rules are solely responsible for producing the user interface (play sounds, change the user's current context, and so on). No interface code is located in the core of Mercator itself.

In the earlier implementation, these rules were hard-coded into the system in a stylized predicate/action notation expressed in C++. In the current implementation, all of the interface rules are expressed in an interpreted language which is parsed and executed as users interact with the application. The interpreted approach has the benefit that we can quickly experiment with new auditory interfaces without having to recompile the system. It also allows easy customization of interfaces by users and administrators.

The interpreted language is based on TCL (the Tool Command Language [Ous90]), with extensions specific to Mercator. TCL is a light-weight language complete with data types such as lists and arrays, subroutines, and a variety of control flow primitives, so Mercator rules have available to them all of the power of a general-purpose programming language. Table 2 presents some of the Mercator-specific extensions to TCL.

Key Word	Description
currentobject	Get or set the current object.
currentclient	Get or set the current client.
callaction	Fires the named action procedure.
playsound	Plays a sound, allowing control over volume, muffling, rate, etc.
addaction	Make the named action procedure callable from C++.
setfocus	Moves the pointer and sets the focus to the named object.
button	Synthesize either a button press or release from the mouse.
speak	Send a string to the speech synthesizer. Control over voice and interruptibility is provided.
sreader	Invoke screen reader function on the specified object (word, line, paragraph reading, etc.)
widget	Retrieve information from the data store about widget hierarchy and state.
bindkey	Shortcut for bindevent which associates the named action procedure with a keypress.
bindevent	Associates an action with an event type.
key	Synthesize a key press or key release event to the object which currently has the focus.
resource	Get or set the value of a resource on the specified object.

TABLE 2. Mercator Language Extensions

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 will load an application-specific rule file if it exists. This allows an administrator or user to configure an interface for a particular application 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 application-generated actions are uniformly fired whenever Mercator is aware of the change. The call-out to actions occurs automatically whenever the data store is updated. This technique is reminiscent of access-oriented programming systems, in which changing a system variable causes some code to be run [Ste86].

Here is an example of an extremely simple action. This action is defined as a TCL procedure with four arguments: the name of the application, its class, the initial current location within that application, and an ID token which can be used to programmatically refer to the application. When the action fires, speech output is generated to inform the user of the presence of the new application, and the user's context is changed to the new application.

```
proc NewApplication {name class loc id} {
    speak "Application $name has started."
    currentclient $id
    speak "Current location is now $loc."
}
```

This action procedure is first made visible to the C++ side of Mercator through a call to `addaction`. `Addaction` is a language extension we have added which "publishes" the name of a TCL procedure so that it may be called from compiled code. After this, bindevent is called to bind the action procedure `NewApplication` with the event type (also called `NewApplication`) which is generated whenever a new application is started:

```
addaction NewApplication
bindevent NewApplication NewApplication
```

Bindings can be changed at any time, and rules can themselves change the event to action association.

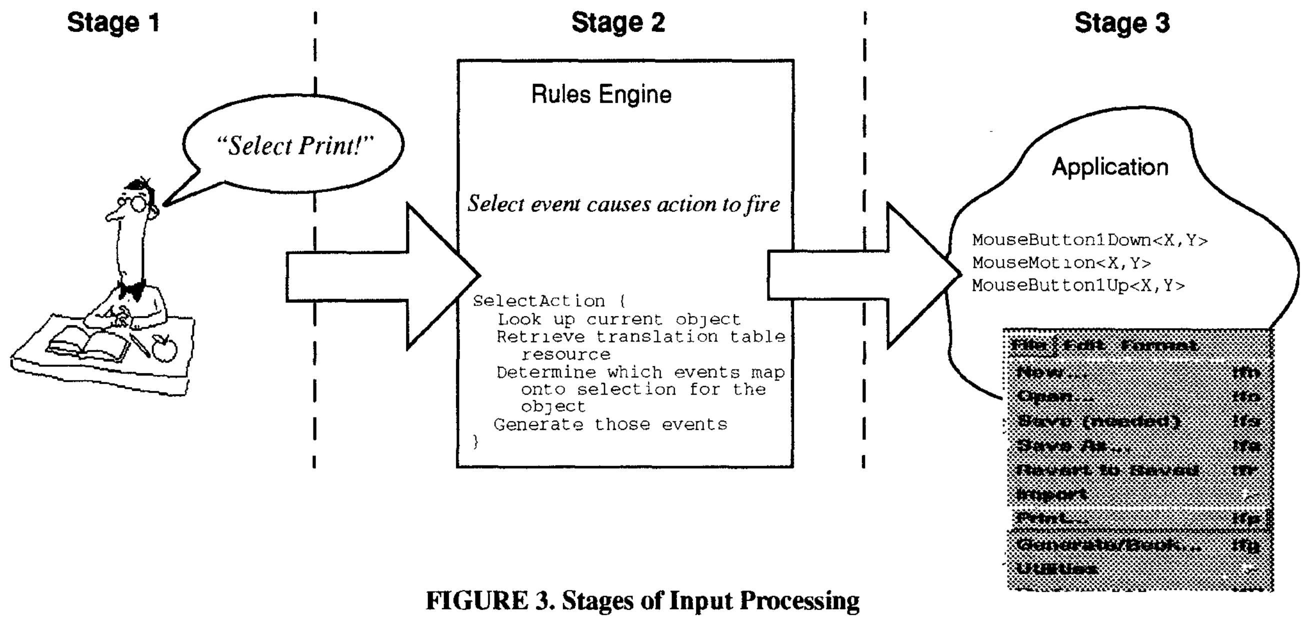
Output

All output to the user is generated through the interface rules. 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 various 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.

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



```
SelectAction {
  Look up current object
  Retrieve translation table
    resource
  Determine which events map
    onto selection for the
    object
  Generate those events
}
```

```
MouseButton1Down<X,Y>
MouseMotion<X,Y>
MouseButton1Up<X,Y>
```

FIGURE 3. Stages of Input Processing

Input

User input handling can be conceptually divided into three stages (see Figure 3). At the stage closest to the user, 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 keypad or a speech recognition engine).

At the middle 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). Some rules, however, will generate controlling input to the application. This input is passed through to the third stage.

At the third stage, the stage closest to the application, 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 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. The figure shows an example of Mercator generating the events required to select a menu item ("Print") when a spoken selection command is detected.

We have opted to synthesize input to the application by directly generating the low-level events which would be required if a user with a mouse and keyboard were operating the interface. One possible alternative, providing messages in the RAP protocol to operate interface components, was rejected as not practical because it is possible that some applications rely on actually receiving "real" events.

Consider the example of selecting a push button. The user's context is at a push button widget in the application interface. The actual input the user provides to select the push button may be a spoken command or a keypress. Mercator detects the user input and fires any action procedures bound to this input. If the input provided by the user is bound to an action which attempts to select the current object, this action will query the current widget to retrieve its translation table. The translation table will be parsed to determine the events which must be generated to cause an actual "select" action to take place in the application interface. The action will then generate these events to the widget.

We are currently using the XTEST X server extension to generate these events to the widget (see Figure 2).

STATUS

The client-side library hooks 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 standard distribution of X.

The various components of Mercator are written in C++; the current system is approximately 16,000 lines of code, not counting 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 as Remote Procedure Call services, with C++ wrappers around the RPC interfaces.

The speech server supports the DECtalk hardware and the Centigram software-based text-to-speech system and provides multiple user-defined voices. The non-speech audio server controls access to the built-in audio hardware and provides prioritized access, on-the-fly mixing, spatialization of multiple sound sources, room acoustics, and several filters. In our previous release, the non-speech audio server ran only on DSP-equipped workstations (either a NeXT machine or a Sun SPARCstation equipped with an Ariel DSP board). The current system will run on any Sun SPARCstation, although a SPARCstation 10 or better is required for spatialization [Bur92].

We are undertaking a commercialization effort to bring our work to the potential users of such a system.

FUTURE ISSUES

There are several new directions we wish to pursue with both the Mercator interface and the Mercator implementation.

Our current implementation of the Mercator core is single-threaded. While the various I/O servers are implemented as separate heavy-weight processes, the actual application manager itself consists of only one thread of control. This can create problems in the case of, for example, non-robust interpreted code. Any interpreted code which loops indefinitely will effectively "hang" the system. We believe that a multithreaded approach will provide more modularity, robustness, and performance to the system.

We have also begun to experiment with voice input to the system. We are using the IN3 Voice Control System, from Command Corp, which is a software-only speech recognition system for Sun SPARCstations. Recognized words from the voice input system are passed into the rules engine just like any other events: keypresses, mouse button presses, and so on. We are investigating high-level abstractions for input and output so that users can easily select which I/O media they wish to use on the fly. Essentially these new abstractions would add a level of indirection between the low-level hardware-generated events and the tokens which the rules engine uses to fire action procedures.

ACKNOWLEDGEMENTS

This work has been funded by Sun Microsystems and the NASA Marshall Space Flight Center. We are grateful to them for their generous support.

REFERENCES

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

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

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

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

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

[My94] Mynatt, E.D., "Mapping GUIs to Auditory Interfaces, In Kramer G. (ed), Auditory Display: The Proceedings of ICAD '92. SFI Studies in the Sciences of Complexity Proc. Vol. XVIII, Addison-Wesley, April 1994.

[ME92] Elizabeth Mynatt and W. Keith Edwards. Mapping GUIs to Auditory Interfaces. In *UIST '92: The Fifth Annual Symposium on User Interface Software and Technology Conference Proceedings*, November 1992.

[MW94] Elizabeth Mynatt and Gerhard Weber. Nonvisual Presentation of Graphical User Interfaces: Contrasting Two Approaches," in the *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 1994.

[Ous90] J.K. Ousterhout. "TCL: An Embeddable Command Language," in the *Proceedings of the 1990 Winter USENIX Conference*, pp. 133-146.

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

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

[Ste86] Stefik, M.J., Bobrow, D.G., and Kahn, K.M. "Integrating Access-Oriented Programming into a Multiparadigm Environment." *IEEE Software*, 3,1, IEEE Press, January, 1986, 10-18.