ENHANCEMENTS TO EXENE
an X windowing toolkit for Standard ML

by

DUSTIN B. DEBOER
B.S., Kansas State University, 2003

A THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Science
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2005

Approved by:

Major Professor
Alley Stoughton

ABSTRACT

This thesis gives background on eXene, a multi-threaded X window system toolkit for Standard ML. First the strengths of the current version of eXene are reviewed, but then deficiencies that limit applications built on this toolkit are introduced. To overcome many of these deficiencies, enhancements are introduced, many of which were jointly designed by the author and his major professor. The author then describes his work in implementing these design changes. EXene widget programming conventions to maximize concurrency are established, and example widgets using these conventions are created. Extensions for setting widget input focus, customizing applications from X resource specifications, and revised interfaces for acquiring and requesting X selections are designed and implemented. EXene X authorization code is revised to behave similarly to the expected Xlib behavior. Finally, implementations are evaluated to verify that the design changes operate as specified and sufficiently address the deficiencies identified.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Both higher-order languages and graphical user interfaces are of great importance to application developers. Functional languages such as Standard ML offer many features such as strong typing, an emphasis on freedom from side-effects, an emphasis on recursive programming, and rule-based programming [18]. Graphical user interfaces allow applications to more naturally interact with users. In addition, modern applications seek to maximize concurrency, especially in graphical user interfaces, in order to offer greater application responsiveness and more efficiently utilize processing resources.

EXene is an X window system toolkit that allows application programmers to construct graphical applications in the functional language Concurrent ML. Concurrent ML (CML) is a set of libraries built in Standard ML that allows the creation of lightweight threads and offers inter-process communication via synchronous events [1].

CML is implemented as a library of SML that, using SMLs built-in first-class continuations, supports the creation of threads in SML programs. In addition, as concurrent programs frequently must collaborate in order to accomplish useful tasks, CML supports synchronous communication over typed channels. CML also provides powerful mechanisms for selective communication over these channels, since a thread may have several channels over which valid communication at any given point in time may flow and must therefore

choose between such communication to proceed with. [14]

```
structure CML =
    ...
    type thread_id
    type 'a event
    type 'a chan

    val spawn : ( unit -> unit ) -> thread_id
    val channel : unit -> a chan
    val sendEvt : (a chan * a) -> unit event
    val recvEvt : a chan -> a event
    val sync : a event -> a
    val wrap : (a event * a -> b) -> b event
    val choose : a event list -> a event
    val send : (a chan * 'a) -> unit
    val recv : a chan -> a
    val select : a event list -> a
    ...
end struct
```

Figure 1.1: Selected CML functions

Figure 1.1 shows several selected CML functions. The `spawn` function accepts a function, and returns a `thread_id` after spawning a thread that will execute that supplied function. For inter-thread communication, the `channel` constructor function allows for the creation of typed channels. After a CML channel is created, values may be sent and received synchronously on the channel by the `send` and `recv` functions – that is, `send` blocks until another thread receives the value from the same channel.

CML also introduces `event` types – a type τ `event` is "the type of a synchronous operation that returns a value of type τ when it is synchronized upon." [14] A thread may synchronize on an event with the `sync` function, or may cause an enabled event to be nondeterministically chosen upon

synchronization from a list of valid events with the `choose` function. We may now notice that `send` is equivalent to `sync o sendEvt`, `recv` to `sync o recvEvt`, and `select` to `sync o choose`. Also, the `wrap` function associates a post-synchronization action to take with an event.

An example Concurrent ML program is shown in Figure 1.2. When the `main` function is executed, it spawns two threads – one producer thread, and one consumer thread. The producer thread maintains an integer counter state, and attempts to send the value of that state over a channel, or attempts to receive an exit signal over another thread. The consumer thread receives integers over the first channel until the count reaches 100, when it signals the first thread to terminate over the exit channel.

EXene was written by John Reppy and Emden Gansner [3, 5] in CML to be naturally multi-threaded – that is, the state of each window and graphical widget would be encapsulated in the state of a thread. The threads maintaining the state of graphical components may then selectively listen for input events and, upon receiving such events, may independently update their state. This contrasts with the traditional approach followed by most windowing toolkits such as Xlib, where graphical components must register callback functions to be called by a central event processing loop [10, 11].

The X window system consists of a server that is connected to a physical display. It is the role of the X server to display output from connected X client applications, in the form of drawable text and graphics, to hardware display systems. In addition, the X server forwards input such as keyboard presses or mouse actions to the appropriate X client. Clients create lightweight rectangular resources on the server known as windows to accomplish this. Once a window is created, a client may draw text or graphics in the window, and typically the client receives mouse and keyboard input when the mouse pointer is positioned inside the window's area. These server X windows are created hierarchically: each X display screen is assigned a root window inside which client windows are created, and clients windows may contain arbitrary

numbers of subwindows.

EXene is a client toolkit: it allows programmers to create client applications in CML. That is, eXene connects via the X protocol over some network interface to an X server. This level of eXene is referred to as the library level; it is analogous to the Xlib library for implementers of C programs. This level provides basic client functionality such as creating windows, drawing text, and drawing graphics. As well as communicating client commands to the X server, the library level also exposes CML events which are enabled upon receipt of messages (such as mouse or keyboard input messages) from the X server, enabling applications to receive input by synchronizing on these events.

EXene also provides a higher-level widget abstraction layer on top of the library level. Just as widget libraries such as Motif are available for X window clients written in the C language, eXene also provides a widget level with reusable graphical components. A CML application may therefore be composed of a collection of graphical widgets. Each of these widgets may expose higher-level functionality than would be available at the library level. For example, a button widget may expose `BtnDown` and `BtnUp` events, and a text widget may allow `getText` and `setText` functions. Widgets are described in more detail in Chapter 2.

EXene is an extremely useful toolkit. It allows application programmers to assemble graphical applications in Standard ML that would be extremely complex without the support of the toolkit. However, there are a few aspects of the current version of eXene that are troublesome to developers. In this thesis, we shall examine the most major of these aspects, and solutions to address those issues will be described.

Please note that much of the design work for the eXene extensions described here was done jointly by the author and his major professor. In addition, in some parts of this thesis the author uses portions of the paper by his major professor and himself [1], primarily for those portions where he

was the primary author in that joint paper.

```
structure CMLDEMO : sig
    val main  : (string * string list) -> OS.Process.status
  end = struct

    val countChan : int CML.chan = CML.channel ()
    val exitChan : unit CML.chan = CML.channel ()

    (* send a count on the countChan until signaled to exit. *)
    fun produceCount n =
        CML.select([
            CML.wrap(CML.sendEvt(countChan,n),
                fn () => (produceCount (n+1))),
            CML.wrap(CML.recvEvt(exitChan),
                fn () => (RunCML.shutdown OS.Process.success))
        ])

    (* receive a count on countChan until count is at least 100,
     * then signal the producer to exit. *)
    fun consumeCount () =
        let
        val n = CML.recv(countChan)
        val _ = TextIO.print ("Consumer received "^
                    (Int.toString n)^"\n")
        in
            if (n<100)
            then consumeCount ()
            else CML.send (exitChan, ())
         end

    fun main _ = RunCML.doit ((fn () => (
        CML.spawn (fn () => (produceCount 0));
        CML.spawn (consumeCount); ()
    )),NONE)
end
```

Figure 1.2: Example CML program

Chapter 2

Widget Programming Conventions

When an eXene widget is realized, it is granted one X window upon which to render itself and an input environment on which messages will be received. This input environment consists of keyboard, mouse and command addressed message input streams [5, 6], as well as a command-out stream (whereby command messages such as resize requests may be sent), all represented as CML events. Each of these input streams has a corresponding output stream upon which the messages are sent by its parent.

Composite widgets—widgets containing one or more child widgets, such as layout widgets—maintain output streams corresponding to the child widgets' input streams. A composite widget must contain a router which determines the child widget (window) a message is destined for, and then sends it on to the child widget via the output stream corresponding to that child's input environment. In addition, many widgets offer CML events to the application—for example, a button widget may offer a button activity event on which `BtnDown` and `BtnUp` events may be received. The button widget maintains an output stream over which these events are sent to the user application.

EXene is intended to be a fully concurrent system [5]. To this end, each widget's state is normally encapsulated in its own thread. That is, a button widget's state is represented by a thread that listens for user input on its

input streams and sends selected messages to the application on its output stream.

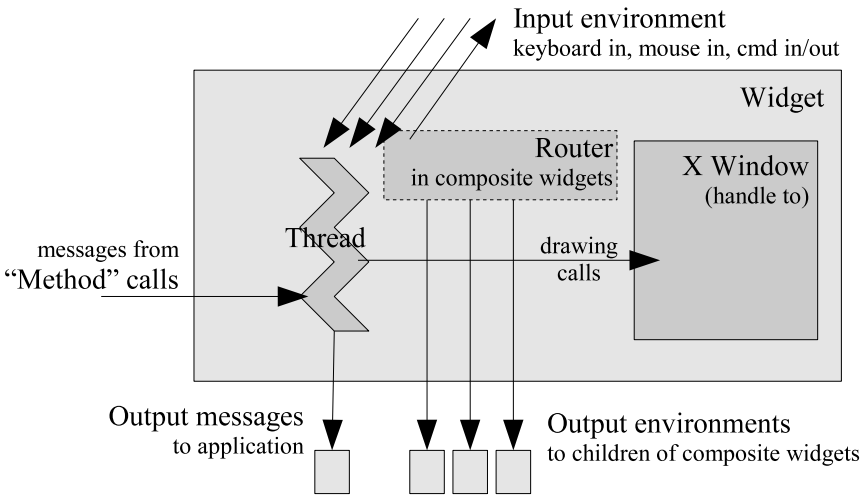


Figure 2.1: Widget components

The functionality of a widget is illustrated in Figure 2.1. User input messages are received by the widget thread, or by a router in a composite widget. Also, other input may be received as a result of the application or widgets calling functions that send messages to (and optionally return data from) the widget thread; since these functions operate on the state of the widget let us refer to them as widget "methods." Drawing messages are sent to the X server via a handle to an X window, and output messages are sent to the application, or to child widgets in composite widgets.

In principle, encapsulating the state of a widget in a thread would allow all widgets in an application to execute concurrently—e.g., if a particular widget was executing a long-running computation, other widgets could continue to execute, even responding to further user input. However, because communication in CML is synchronous, and since eXene's message router doesn't buffer messages, it turns out that some of this possibility for concur-

rency is lost.

Because CML communication and the widget input/output streams are synchronous, any failure of a widget or application to respond to input in a timely manner has the potential to block the execution of other widget threads. Consider a widget E that performs some extensive computation, and that is part of a composite widget C. While E is performing the computation, any messages that the router thread of C attempts to send to E will be blocked. Therefore, no further input messages will reach any other children of C, and furthermore, if C itself is part of another composite widget, that parent router thread will also become blocked. This is clearly a loss of concurrency, since eXene was designed so that C and all of its children should have the potential to execute concurrently. This cascading blocking may also arise if the application itself fails to respond (at all, or in a timely manner) to messages from a widget. The failure of any recipient of messages in an application to respond to input will eventually block execution of all widget threads in the same shell (the eXene abstraction for a top-level window).

In addition, some possibilities for deadlock arise by virtue of the bidirectional communication between widgets and applications, and between parent widgets and child widgets. A widget typically implements its methods by creating a request channel (stream) over which operation requests may be sent by applications. Optionally, return values may be sent via synchronous variables specified in the requests. This functionality is hidden from the application programmer, who sees only a method that blocks until complete; the application programmer cannot selectively communicate over the widget request channels. Now, suppose that an application A contains a widget W with a method m. Further suppose that the application applies m to W. The application is now blocked waiting for a reply (or receipt of the request message) from W. But simultaneously, W may be attempting to send a message to A without selective communication[1] (without allowing for the possibility

[1]This deadlock situation is avoidable if each widget uses selective communication for

to also receive the method request message). This causes both W and A to be blocked waiting on the other. Unfortunately, this blocking will soon cascade to the rest of the shell. Similarly, a child widget might be trying to communicate with its parent, while the parent was trying to communicate with the child.

2.1 Suggested Widget Programming Conventions

In [4, p. 42], Gansner and Reppy say that, in communication between a parent widget and one of its children, the parent has the responsibility to be responsive, and that queuing of a child's messages to its parent could be used to avoid deadlock. (A `wrapQueue` function is provided for this very purpose.) We propose using this idea of parental responsibility as the basis for widget programming conventions, and apply it not just in parent widget/child widget communication, but also in communications between a widget (thought of as the parent) and the application (thought of as the child). In general, we say that

Parents should be more responsible than children.

Widgets should be tolerant of errors in user applications, and composite widgets should be tolerant of errors in their child widgets.

A parent widget's first step in becoming more responsible than children is to queue messages sent to child widgets or applications. This prevents parent threads from being blocked by error-prone or slow children, at the cost of extra queue threads and some buffer space. It may sometimes be necessary to flush the queue—perhaps at the request of an application that knows that only future messages are of interest. Because all queued messages originated with the user, there is little risk of the message queues becoming especially long.

---

each output message sent—however, it is not trivial to design widgets in this way.

Queuing messages sent to child widgets does not entirely prevent parent threads from being blocked by unresponsive child threads, however. All widgets have a `boundsOf` method that returns the requested geometrical bounds of the widget. Parent widgets generally calculate their own bounds based on the requested bounds of their children. Even if parents queue messages to their children, a parent will still be blocked while it calls the `boundsOf` method of one of its children. Most `boundsOf` methods are implemented similarly to other methods, with bound-of requests sent on a request channel. To avoid this source of blocking or unresponsiveness, let us establish another convention—the `boundsOf` function may only be called prior to widget realization. After realization, the parent should cache the requested bounds of the child, and only update the bounds when the child requests it be resized in a resize request (which will now include the requested bounds).

Prior to realization, we shall assume that the `boundsOf` function of child widgets will terminate, in a reasonably short time. We may justify this assumption in several ways. First, it is much more trivial to program a widget to ensure termination of methods prior to realization, before input or output message streams must be monitored. In addition, any source of deadlock arising in a child widget prior to realization is likely to be deterministic, as no source of user input is yet available to the child widget or indeed any other widget in its application. Therefore it is likely that any design error leading to such non-termination will be identified and removed when the designer tests the widget.

This convention suggests the following life cycle of a widget:

- **Construction**. The widget is created, and the thread encapsulating its state is started. Some of its methods may now be called by the application.

- **Bounds Determination**. The `boundsOf` method of the widget is called, determining the requested bounds of the widget. The `boundsOf`

method should never be called again in the lifetime of the widget; exception `BoundsFunctionAlreadyCalled` will be raised if `boundsOf` is called again.

- **Realization**. The `realize` function of the widget is called, supplying the widget with an input environment and a window. The `realize` function should never again be called in the lifetime of the widget; calling `realize` again will raise exception `AlreadyRealized`.

  Note that, if the widget's desired size changes after its bounds function has been called but before it is realized, its parent won't know what this desired size is. Some widgets in the current eXene release suffer from this defect. It can be avoided by having the widget remember that it should ask its parent to resize it after realization.

- **Post-Realization**. The widget is realized, and may be visible on the display. User input or method calls causing a change in the desired bounds of the widget should cause the widget to send its parent a resize request accompanied by the desired bounds. Such requests may not be honored, and should not be repeated.

- **Death**. The widget is notified of the loss of its window by a `CI_OwnDeath` message.

To summarize, let us insist on the following eXene widget programming conventions:

- Parent widgets must queue output sent to child widgets and applications, and may flush those queues in some cases.

- A widget's `boundsOf` function may only be called prior to realization, and the parent should cache a child's desired bounds. Subsequently, the child is responsible for letting its parent know when its sizing wishes have changed, supplying it new bounds as part of the requests.

- A widget's methods must be guaranteed to terminate (ideally, in a timely fashion).
- Attempts by a child widget to send messages to its parent should always succeed (ideally, in a timely manner).

2.2 Implementation and Testing of Conventions

The forementioned conventions have been implemented and tested in a few key widgets and structures in the eXene distribution. Queues have been added to message streams in the generic router provided with the widget library, queues have been added to the `Button` widgets, a new example `SimpleEdit` text widget has been written that conforms to the widget life cycle, and the `BoxLayout` widget has been modified to cache the bounds of its children.

```
val mkRouter : Interact.in_env * Interact.out_env *
  (EXB.window * Interact.out_env) list -> router
val addChild : router -> EXB.window * Interact.out_env -> unit
val delChild : router -> EXB.window -> unit
val getChildEnv : router -> EXB.window -> Interact.out_env
```

Figure 2.2: Widget router functions

The eXene widget library provides a `Router` structure whereby composite widgets may easily construct a router to handle passing input messages on to the appropriate children. A parent widget may construct a router via the `mkRouter` function (shown in Figure 2.2). It is the `mkRouter` function's task to spawn a thread to route input messages to the widget they are addressed to – either a child widget or the parent widget itself. This constructor function therefore accepts as arguments the parent widget's original input environment I, an output environment O' (corresponding to the parent widget's

new input environment I'), and a list of child windows and child output enviroments.[2] The router then receives messages on input environment I and dispatches them to the correct child window's output environment. If the message is addressed to the parent widget, the router sends the message over the output environment O' which is received by the parent on the new input environment I'.

The `Router` structure has been modified to queue all messages sent on the keyboard, mouse, and command-in streams of the output environments held by the router (the output environment to the parent and output environments to the children). This satisfies item 1 of the four widget programming conventions outlined in the previous section, for composite widgets using the widget `Router` structure.

The `BoxLayout` widget provides a method for arranging lists of graphical widgets in horizontal or vertical layouts. In order to compute its own desired bounds, as well as to perform this layout task, the `BoxLayout` widget must know the desired bounds of its children. In the current release of eXene, the layout widget calls the bounds function of all of its children whenever it needs to recompute its layout, or whenever it must calculate its own desired bounds. This code has been modified to cache the first bounds reply from the child widgets – which satisfies item 2 above, provided that the bounds function of the layout itself is called only before realization.

The eXene `ButtonCtrl` functor provides the foundation for all of the events dispatched by eXene buttons. `ButtonCtrl` interprets mouse messages as appropriate and converts these into `button_act` messages, such as `BtnUp` or `BtnDown` messages. If a client application were to use eXene button widgets, but failed to receive button events in a timely manner (either because

[2]Widget input and output environments are created in pairs; user input and control-in commands are sent to a widget over streams of an output environment and received over the streams of the corresponding input environment, while control-out commands are sent over a stream of the input environment and received over a stream of the corresponding output environment.

```
signature WIDGET_BASE =
  sig
    ...
    val wrapFlushableQueue : 'a CML.event ->
         ('a CML.event) * (unit CML.event)
    ...
  end
```

Figure 2.3: The wrapFlushableQueue method.

of lengthy computation or design error) button widget threads would become blocked. Therefore, the `ButtonCtrl` code has been modified to use the `wrapFlushableQueue` function that converts a blocking CML event into a non-blocking CML event, by a queue that is optionally flushable by synchronizing on the additional unit CML event returned.

In addition, a `flushEvts : button -> unit` method has been added to the eXene buttons, whereby button events may be flushed from the outbound event queue. This may be useful in situations where an application wishes to invalidate all user input occurring after some user input event is received. For example, suppose a button bar where upon the press of one button, all buttons became inactive until some processing is completed. In this case, it would be useful to flush the queues of all buttons in the button bar upon deactivation to ensure that any events received after reactivation occurred only after reactivation.

An example `SimpleEdit` text widget has also been created that demonstrates principles outlined in this widget conventions chapter (as well as some features to be introduced in later chapters). The `SimpleEdit` widget has no output message streams to a client application, so it is not obligated to buffer its output. However, it does raise the `BoundsFunctionAlreadyCalled` exception if its bounds function is called more than once. Methods of the widget are designed to send messages to the thread encapsulating the state

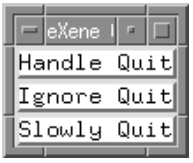
```
signature SIMPLEEDIT =
  sig
    structure W : WIDGET
    type simple_edit
    val simpleEdit : (W.root * Widget.view * Widget.arg list)
            -> string -> simple_edit
    val setString : simple_edit -> string -> unit
    val getString : simple_edit -> string
    val setSelection : simple_edit ->
            (int * int * W.EXB.XTime.time) -> unit
    val getSelection : simple_edit -> string
    val widgetOf : simple_edit -> W.widget
    val takeFocus : simple_edit * W.EXB.XTime.time -> unit
    val focusableOf : simple_edit -> Shell.focusable
  end
```

Figure 2.4: The SIMPLEEDIT signature.

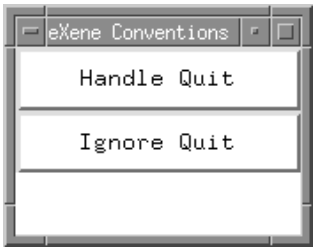
of the widget, which selectively receives these messages along with user input messages. This ensures responsiveness as there is no state where some input cannot be accepted.

Finally, a demonstration application has been built that demonstrates how the above conventions serve to increase responsiveness in applications. For use with this application, a copy of the eXene text button widget was created that introduces a delay in passing mouse button-down and button-up events on to the application. An excerpt of this source code is shown in Figure 2.6. This modification allows this button to simulate a slow-responding widget in an application.

A screenshot of this demonstration application is shown in Figure 2.5. The application uses three (almost identical) buttons labeled "Handle Quit", "Ignore Quit", and "Slowly Quit". Ordinarily we would supply to `CML.select` the list of all events that would occur in the operation of this application – in this particular application, `quitEvt1`, `slowEvt`, *and* `quitEvt2`. However,



(a) Upon startup.



(b) After resize.

Figure 2.5: Widget conventions demonstration screenshots

by failing to select on `quitEvt2` we may simulate the case where an event of `quitBttn2` caused long-running computation (or perhaps even deadlock). Thus, `quitBttn2` will fail to send any events to the demo application. However, we may verify that the widget threads are not blocked and continue to execute concurrently, since button events are buffered in the button's output queue. This may be verified with the following test case: launch the application, click on "Ignore Quit" (the event of which will not be received by the application), resize the application, and click on "Handle Quit". In this test, the application will respond to resize requests (the buttons should redraw after the resize) and should exit when the "Handle Quit" button is clicked. That is, even though no events are received from the second button, both button threads continue to execute and events may be received from the first button.

In addition, by the use of the slowly-responding button, we may simulate the case where a slow or compute-intensive widget causes input messages

to be buffered in the composite widget router. We may launch the application, then click on "Slowly Quit". The slow-running button will wait five seconds before sending a `BtnDown` message, and another five seconds after releasing the mouse button before sending a `BtnUp` message. If we resize the application while we are waiting to respond, we note that the application looks similar to screenshot (b) of Figure 2.5. That is, we note that the upper two buttons are responding and have redrawn, but the bottom slow button is busy and has not redrawn itself. This demonstrates that the composite widget router is behaving as designed, and that the eXene widgets in the application are executing concurrently.

The results obtained from the test cases performed on this test application suggest that the conventions introduced in this chapter are indeed helpful in increasing eXene widget concurrency.

```
    val quitBttn1 = Button.textBtn (root, view,
        [([], Attrs.attr_label, Attrs.AV_Str "Handle Quit")])
    val quitEvt1  = Button.evtOf quitBttn1
    val quitBttn2 = Button.textBtn (root, view,
        [([], Attrs.attr_label, Attrs.AV_Str "Ignore Quit")])
    val quitEvt2  = Button.evtOf quitBttn2
    val slowBttn  = TestButton.textBtn (root, view,
        [([], Attrs.attr_label, Attrs.AV_Str "Slowly Quit")])
    val slowEvt   = TestButton.evtOf slowBttn
    ...
    fun loop():unit =
        let
        fun handleQuit (Button.BtnUp _) = quit()
          | handleQuit (_) = loop()
        in CML.select
           [CML.wrap(quitEvt1, handleQuit),
            CML.wrap(slowEvt, handleQuit)]
        end
    ...
```

(a) Demo application code excerpt.

```
...
| handleM (MseUp (bttn,time),((s,isin,isdown),drawf)) = let
    val state' = (s,isin,false)
    in
      drawf state';
      (* timeout 5 seconds to test buffering *)
      CML.sync (CML.timeOutEvt (Time.fromSeconds 5));
      send(evtc, ... BT.BtnUp ...
    end
...
```

(b) Slow button code excerpt.

Figure 2.6: Widget conventions demonstration code

Chapter 3

Handling Input Focus

By default, the keyboard input focus of an X application is set to the root window, which means that keyboard input is sent to the window currently pointed to by the mouse [12, p. 612] [9]. This functionality can be annoying to deal with in eXene applications, particularly when trying to enter text in an application with multiple text input fields. The X protocol (the set of valid requests that a standard X server will accept) provides the `SetInputFocus` request for assigning keyboard focus to a particular window. This allows, e.g., an application to assign a text input widget input focus so that movement of the mouse pointer will not affect the user's ability to enter text in that widget.

Motif also provides the ability to navigate between widgets by moving the keyboard focus between "tab groups" of widgets; this is accomplished by pressing a particular key (usually, of course, Tab). As normally all widgets are assigned to be part of a tab group, this effectively allows a user to move keyboard focus to every widget accepting keyboard input in an application by the use of the Tab navigation key [12, p. 172]. This focus-handling functionality would be very useful in eXene, as it would help provide a more pleasant experience for users.

A top-level window may participate in the `WM_TAKE_FOCUS` window manager protocol, so that the window manager will send it a `CLIENT_TakeFocus`

client message when it assigns focus to the window; in addition, any window may receive `FocusIn` and `FocusOut` events indicating that it has received or lost input focus [17, pp. 648,592]. When a top-level window receives a `CLIENT_TakeFocus` client message, it might use the `SetInputFocus` X request to reassign focus to the sub-widget that had it before focus was lost.[1] And some widgets might highlight their borders when they have input focus.

3.1 Implementation of Input Focus Extensions

As a basis for widgets to set input focus in eXene, a `setInputFocus` method for setting the keyboard input focus to a window has been added. EXene's `createSimpleTopWin` function has also been modified to return a `client_msg` CML event whereby `CLIENT_TakeFocus` client messages may be read. Also, the ability for eXene windows to receive `CI_FocusIn` and `CI_FocusOut` messages over their input environments has been added.

Support has also been provided for the `WM_DELETE_WINDOW` window manager protocol. When a top-level window participates in this protocol, it will receive a `CLIENT_DeleteWindow` client message when the user, via the window manager, has requested that window delete itself. This message can also be received on the above-mentioned `client_msg` CML event. A `deletionEvent` method has been added to the widget shell whereby a unit event may be obtained that can be synchronized on when the shell's top-level window has received a `CLIENT_DeleteWindow` message (Figure 3.1).

```
val deletionEvent : shell -> unit CML.event
```

Figure 3.1: The `deletionEvent` method

[1] A `CLIENT_TakeFocus` client message carries the timestamp of the X event that caused the window manager to assign focus to the top-level window. This timestamp (which isn't part of a `FocusIn` event), must be supplied to a subsequent `SetInputFocus` request.

```
signature SHELL =
sig
  ...

    datatype focusable_msg = FocusIn
                           | FocusOut
                           | Assign of Interact.time
                           | Release of Interact.time
                           | Next of Interact.time
                           | Previous of Interact.time

    datatype focusable = Focusable of
                       {focusableEvt : focusable_msg CML.event,
                        takeFocus    : Interact.time -> unit}

    type fid

    val addFocusableFirst : shell -> focusable -> fid
    val addFocusableAfter : shell -> fid * focusable -> fid
    val deleteFocusable   : fid

  ...
end
```

Figure 3.2: The SHELL signature

A focus manager has been added to the widget top-level shell (the widget-level abstraction of a top-level window). The additions to the SHELL signature are shown in Figure 3.2.

This focus manager allows a user to move input focus through a list of eXene widgets/windows by means of some navigation keys, for example Tab. Widgets that can be turned into objects of type `focusable`, e.g., via a

```
val focusableOf : some_widget -> focusable
```

method, may be added to the manager. A focusable object will inform the

manager by means of a `focusable_msg` when input received indicates that the focus has been received or lost, when focus should be assigned to the object (perhaps upon a mouse click; carried out by invoking the object's `takeFocus` method), when focus should be moved to the next or previous focusable object (perhaps upon a Tab or Shift+Tab), or when focus should be released (perhaps upon an Esc).

Because the focus manager of the shell will know which, if any, of its focusable objects currently has the focus, when it receives a `CLIENT_TakeFocus` client message, it can take appropriate action when none of its focusable objects currently have the focus. If none of its objects ever had the focus, or the last one to have the focus explicitly gave it up, then the manager can assign focus to the first of its objects. Otherwise, it can set the focus back to the object that had the focus before focus was lost. The time that is included as part of some of the focusable messages and that is passed to the `takeFocus` method is always supposed to be the time at which the user pressed/released the key or mouse button that initiated the change.[2] This time must be passed to a subsequent call of `setInputFocus`.

The type `fid` stands for "focusable object identifier"; `fid`'s are used to refer to managed focusable objects. A shell's focus manager is told to manage focusable objects using the methods `addFocusableFirst` and `addFocusableAfter`; they return `fid`'s for referring to those objects. The `addFocusableFirst` method makes the supplied focusable object be the first element of the list of managed objects, whereas the `addFocusableAfter` method makes the supplied object be the next object after the object named by the supplied `fid`. Finally, the `deleteFocusable` method is used to stop a focus manager from managing a given focusable object.

A `FocusFrame` composite widget has been added to the eXene code base (Figure 3.3). A `FocusFrame` wraps around a widget and its focusable object,

[2]The current eXene release didn't annotate keyboard messages with timestamps; this has been modified so that a `KEY_Press` contains the timestamp so that, e.g., Tab may cause focus to be assigned to the next focusable object.

```
signature FOCUSFRAME =
  sig
    type frame
    val focusframe : (W.root * W.view * W.arg list) ->
          (W.widget * Shell.focusable) -> frame
    val widgetOf : frame -> W.widget
    val focusableOf : frame -> Shell.focusable
  end
```

Figure 3.3: The FOCUSFRAME signature.

and draws a border around the child widget when that widget has focus. This is done by monitoring the `focusableEvt` of the focusable object. Of course, the `FocusFrame` widget has a `focusableOf` method that may be used to turn it into a focusable object, enabling it to be added to the focus manager of the shell.

The `SimpleEdit` widget (Figure 2.4) has been implemented as an example of a widget that accepts focus. It sends an `Assign` focus message to its focus manager when it receives a `MOUSE_FirstDown` event, a `Next` focus message when it receives a Tab keypress, and a `Previous` focus message when it receives a Ctrl+Tab keypress. The source code for the `SimpleEdit` widget has been included as Appendix A.

3.2 Testing of Input Focus Extensions

The `SimpleEdit` widget has been incorporated into an input-focus demonstration application `demo-focus`, screenshots of which are shown in Figure 3.4. We may verify the correct operation of the `SimpleEdit` widget, the `FocusFrame` widget, and the `Shell`'s focus manager with a test case executed on this simple application.

We may first launch the application, then click the mouse in one of the

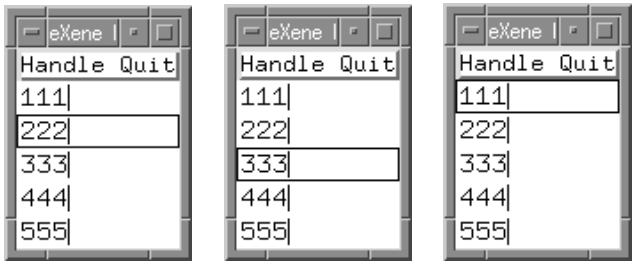


Figure 3.4: Input focus improvements demo application

`SimpleEdit` widgets. This causes the `SimpleEdit` widget to send an `Assign` focus message to its focus manager – in this case, the `FocusFrame` widget enclosing it. The `FocusFrame` widget then sends an `Assign` focus message to its enclosing `Shell` focus manager. The `Shell` focus manager calls the `FocusFrame`'s `takeFocus` function, which in turn calls the `SimpleEdit`'s `takeFocus` function. This causes the `setInputFocus` function to be called, supplying the `SimpleEdit`'s window and the timestamp of the initial mouse click. When the X server notifies the `SimpleEdit` that it has received input focus by way of the `CI_FocusIn`, it enables its `focusableEvt` allowing its `FocusFrame` to determine that it should draw a border around itself, signifying that the widget it contains currently holds the input focus. Suppose that we had clicked on the second `SimpleEdit` widget; the application should now resemble (A) in Figure 3.4. We may easily verify that the widget does indeed hold the input focus by moving the mouse pointer to some other widget, then typing some text that will appear in the widget that holds the focus.

Now, we may press the Tab key – this will cause the second `SimpleEdit` widget to send a `Next` focus message to the `Shell` via its enclosing `FocusFrame`. This in turn causes the `Shell` to call the third `FocusFrame`'s `setInputFocus` function, and it the third `SimpleEdit`'s `setInputFocus` function. If this third widget successfully obtains the input focus after requesting it, and is no-

tified of such an event, a border will be drawn around the third `FocusFrame`. The application will now resemble (B) of Figure 3.4.

We may instead press the Shift+Tab keys. This will cause the `SimpleEdit` widget that currently holds the focus to send a `Previous` focus message to the `Shell` focus manager. In the scenario above, the focus will then move to the first `SimpleEdit` widget, causing the application to resemble (C) of Figure 3.4.

We may perform a few additional tests with this demo application. If we cause a widget in the demo to obtain the input focus, next move the input focus to another top-level window, then cause the demo top-level window to regain focus, the input focus will revert to the widget previously holding the focus before the demo top-level window lost focus.

Additional work on existing eXene widgets would be helpful. For example, it would be helpful to modify the `Button` widgets to accept keyboard input, then modify allow them to return a `Shell.focusable` structure. It would then be possible to create a quite accessible and useful application with buttons and text input that could be navigated entirely from the keyboard. However, in the work relating to this thesis the `Button` widgets have not been modified to return such a `Shell.focusable` structure: this is left to others to implement based on the `SimpleEdit` example.

Chapter 4

Customization using X Resources

Xlib provides for user customization of applications by means of "resource specifications". For example, an application may allow background and foreground color, window geometry, and font settings to be configured by the user [7, p. 339]. Some of these resource settings might be passed as arguments to the application on the command line, such as "`-background white`". On the other hand, some resource specifications may be general to several applications or to all instances of a given application, and these may be stored in a configuration file. Xlib provides support for both of these methods, with a `XrmParseCommand` function for loading resource settings from a list of arguments into a resource "database", and a `XrmGetFileDatabase` function for loading a resource configuration file into a resource database [17, p. 460].

In addition, as X users may often wish to apply a set of resource specifications to all applications on a given display, regardless of whether those applications all have access to a common filesystem, X distributions provide an `xrdb` ("rdb" stands for resource database) utility that loads the contents of a resource specification file into a `RESOURCE_MANAGER` property of the X display [17, p. 450]. The contents of this property may then be used as the contents of a file would.

Finally, as resource specifications may originate from several sources (say, from command line options or the `RESOURCE_MANAGER` property), application

developers must have a way of combining resource databases in such a way that the specifications of one database takes preference over another. Xlib provides the `XrmMergeDatabases` function for this purpose.

EXene currently provides support for user customization of widgets [2]. Widgets are passed the following resource-related information:

- A "view", consisting of a a "style" and a "style-view", where a style is the eXene version of an Xlib resource database, and a style-view is a search key into that style, such as the name of the application.
- An "args" list, consisting of a list of attribute/value pairs.

Internally, the widget maintains an "attrs" list of triples, where each triple consists of an attribute, its type (an element of a datatype of attribute types) and its default value. EXene provides support for searching for the value of an attribute that is in the attrs list, first looking in the args list, then looking in the style as filtered by the style-view, and falling back on the default in the attrs list if necessary. When this search succeeds, it's guaranteed to have the type listed in the widget's attrs list.

4.1 Command Line Parsing Extensions

Xlib provides the routine `XrmParseCommand` that, given a specification of the command line options that the application wishes to recognize, parses command line arguments into a resource database. While it is useful to emulate the option specification used by Xlib, so that eXene applications might recognize the same types of command line options, we must also recognize that eXene does not have any other command line parsing functions available. Therefore, a command line argument parsing function may be useful not only for obtaining user preferences, but for obtaining data for processing by the application. For example, an application may wish to accept requests to set the background color by " `-background blue`", but may also wish to

obtain the value of a filename by " `-filename foo`". There is really no need for the filename to be recognized as a user preference for the application and all of its widgets to view. Let us therefore distinguish between two types of options - "resource" options whose purpose is to be loaded into an eXene style, and "named" options, whose purpose is to be used for application processing.

It is the application's responsibility to set up a command line option specification table. This table, `Styles.optSpec`, shall be a:

```
(optName * argName * optKind * Attrs.attr_type) list
```

The option name, `Styles.optName`, shall be of either `Styles.OPT_NAMED of string` or `Styles.OPT_RESSPEC of string`, where the former is a named option and the latter is a resource option. The string given for a named option shall be any string of the application's choosing, simply used to identify the option when retrieving a value later, such as `OPT_NAMED ("filename")`. The string given for a resource option is a resource name, such as `OPT_RESSPEC ("*background")` or `OPT_RESSPEC ("appname.background")`.

The argument name, `Styles.argName`, shall be a string that is valid to be used on the command line to specify the setting of this option. For example, `"-background"` or `"--bg"` or `"/bg="` might be used as argument name values.

The option "kind" (term taken from Xlib), `Styles.optKind`, shall be of one of the values listed in Table 4.1.

Finally, the option type, of type `Attrs.attr_type`, is the type of the value to be returned.

An example option specification is given here, for an application wishing to find values for named options of "help", "flag", "x", and "y"; and resource options of "*background", "*foreground", and "*borderWidth". It also allows the user to skip an argument or all following arguments on the command line with "-skip" or "-ignore". In addition, note that the option "help" may be toggled on with "-help" or off with "-nohelp".

`OPT_NOARG of string`	Similar to Xlib's `XrmOptionNoArg`. Option will assume the value of the string given if set.
`OPT_ISARG`	Similar to Xlib's `XrmOptionIsArg`. Option will assume the value of the argument name itself if set.
`OPT_STICKYARG`	Similar to Xlib's `XrmOptionStickyArg`. Option will assume the value of the substring following the argument name if set. For example, if `"-bg="` is the argument name, and "` -bg=blue`" is given on the command line, the value of the option will be "`blue`".
`OPT_SEPARG`	Similar to Xlib's `XrmOptionSepArg`. Option will assume the value of the command line argument immediately following if set.
`OPT_RESARG`	Similar to Xlib's `XrmOptionResArg`. A resource specification argument should follow on the command line. For example, if `"-res"` is the argument name, and "` -res "*background:blue"`" is given on the command line, there will be two option values created in the option db returned: one with a name of `"-res"`, of either named or resource specification name, and a value of "`*background:blue`"; and the other of name `OPT_RESSPEC("*background")` and value of "`*background:blue`".
`OPT_SKIPARG`	Similar to Xlib's `XrmOptionSkipArg`. Skip the next argument given on the command line; do not attempt to match it to any option nor to assign it as a value to any option.
`OPT_SKIPLINE`	Similar to Xlib's `XrmOptionSkipLine`. Ignore all following command line arguments given.

Table 4.1: The option "kind" (term taken from Xlib), `Styles.optKind`, shall be of one of the above.

```
structure S = Styles
structure A = Attrs
val optSpec =
[(S.OPT_NAMED("help"),"-help",  S.OPT_NOARG("on"), A.AT_Bool),
 (S.OPT_NAMED("help"),"-nohelp",S.OPT_NOARG("off"),A.AT_Bool),
 (S.OPT_NAMED("sum"), "-sum",   S.OPT_ISARG,       A.AT_Str),
 (S.OPT_NAMED("x"),   "-x=",    S.OPT_STICKYARG,   A.AT_Real),
 (S.OPT_NAMED("y"),   "-y",     S.OPT_SEPARG,      A.AT_Real),
 (S.OPT_NAMED("res"), "-res",   S.OPT_RESARG,      A.AT_Str),
 (S.OPT_NAMED("skip"),"-skip",  S.OPT_SKIPARG,     A.AT_Str),
 (S.OPT_NAMED("ign"), "-ignore",S.OPT_SKIPLINE,    A.AT_Str),
 (S.OPT_RESSPEC("*background"), "-bg",S.OPT_SEPARG,A.AT_Str),
 (S.OPT_RESSPEC("*foreground"), "-fg",S.OPT_SEPARG,A.AT_Str),
 (S.OPT_RESSPEC("*borderColor"),"-bc",S.OPT_SEPARG,A.AT_Str)]
```

Figure 4.1: An example option specification is given here, for an application wishing to find values for named options of “help”, “flag”, “x”, and “y”; and resource options of “*background”, “*foreground”, and “*borderColor”. It also allows the user to skip an argument or all following arguments on the command line with “-skip” or “-ignore”. In addition, note that the option “help” may be toggled on with “-help” or off with “-nohelp”.

```
signature STYLES = sig (* actually defined in StylesFunc *)
   ...
   fun parseCommand: ctxt * optSpec ->
       string list -> optDb * string list
   fun findNamedOpt: optDb -> optName -> Attrs.attr_value list
   fun findNamedOptStrings: optDb -> optName -> string list
   fun styleFromOptDb: ctxt * Styles.optDb -> Styles.style

end
```

Figure 4.2: The `STYLES` command-line functions

Command line arguments may be parsed into an "option database", `Styles.optDb`, with the function `parseCommand`. ParseCommand takes a styles context (basically, a connection to the X server, used in converting strings to certain attribute values such as colors), an option specification, and a list of command line arguments as strings, and returns an option database and the list of strings that were not recognized as option arguments. Note that any unique prefix of an argument name will be recognized as a valid command line option. Also note that the function will not throw any exceptions upon encountering an unknown command line argument, it will simply add that string to the list of unrecognized arguments. The position of the unrecognized arguments is not noted in the list returned; this may be a future enhancement, as the position of these arguments may be important to some applications.

Named command line option values may be retrieved from an option database using the `findNamedOpt` function. The list of attribute values returned are the list of all values specified for the named option on the command line, in reverse order. For example, if "`-x=3.14 -x=3.15 -x=3.16`" were given on the command line, and an `optDb` was returned from parsing these arguments based on the option specification in Figure 4.1, `findNamedOpt`

```
signature DISPLAY = sig
    ...
    fun rootWinOfScr: screen -> XProtTypes.win_id
    ...
end
```

Figure 4.3: The `rootWinOfScr` function

`optDb (OPT_NAMED "x")` would return a list `[A.AV_Real 3.16, A.AV_Real 3.15, A.AV_Real 3.14]`. Note that this is the list of all arguments associated with the "`-x`" flag on the command line, converted to real numbers, in reverse order. In this way, the application could choose to let the last option given have priority over the others, in which case it would choose the head of the list. Or, the application could choose to use all of the option values given and process the whole list.

Note that only "named" options may be returned by `findNamedOpt`; if a resource option is searched on, an empty list will be returned.

If it is sufficient to obtain the value of named arguments as strings, and not as `attr_value`s, one may use the `findNamedOptStrings` function – in which case it is not necessary to supply the context necessary for attribute conversions.

Finally, resource options and their values in an option database may be converted to an eXene style. The function `styleFromOptDb` takes a context and an option database and returns a style.

4.2 Style Management Extensions

When `xrdb` is run, it loads preferences (as strings) into the resource manager (`XA_RESOURCE_MANAGER`) property of the root window of the X-Server. The function `rootWinOfScr`, shown in Figure 4.3, returns the root window id of a screen.

```
signature ICCC = sig
    ...
    fun xrdbOfScr : EXB.screen -> string list
    ...
end
```

Figure 4.4: The `xrdbOfScr` function

```
signature STYLES = sig
    ...
    fun mergeStyles : style * style -> style
    ...
end
```

Figure 4.5: The `mergeStyles` function

The function `ICCC.xrdbOfScr`, Figure 4.4, uses `rootWinOfScr` to retrieve the resource properties stored by xrdb in the `XA_RESOURCE_MANAGER` property and convert them into strings.

When an application obtains eXene styles from several sources (such as command line arguments and X-server preferences) it is necessary to combine or "merge" these styles together. It is also necessary to perform this merge in such a way that certain preferences have priority over others. For example, an application may wish to allow run-time preferences in a style obtained from the command line to have priority over preferences from a style obtained from X-server preferences, which may have priority over preferences from an application default style.

The function `mergeStyles`, Figure 4.5, takes two eXene style arguments, the first the "source" style and the second the "target" style, and merges them into one "merged" style. The function `mergeStyles`, when applied to $(style_1, style_2)$, returns a style where all specifications of $style_1$ have been

inserted into $style_2$, effectively giving priority to the specifications of $style_1$ (this function was trivial to write, given the previously existing support for updating styles).

For ease of use by developers using widgets, functions (shown in Figure 4.6) have been added to the widget `ROOT` signature that facilitate parsing of command line arguments into named values and styles, and the merging of styles.

4.3 Testing X Resource Extensions

The functionality of most of the resource handling code discussed here may be demonstrated with a simple eXene application, the source code of which is included as Appendix D. The `demo-res` application first uses the option specification table shown in Figure 4.1 to parse command line arguments supplied to the application. These include some numeric x and y values, a "`-sum`" flag, and some "`-fg`" and "`-bg`" resource arguments. The x and y arguments are used to pass input to the application. Upon startup, the application obtains the list of all x argument values, sums them, and sets the topmost `SimpleEdit` widget to that value. Likewise, it also sums all y values provided, and sets the second text widget to that sum. If the "`-sum`" flag is given, the application also sums the x values with the y values, and sets the third text widget to that value.

We can see a screenshot of this application in Figure 4.7, run with command line arguments "`-fg White -bg DarkGrey -x=1 -x=2 -x=3 -y 3 -y 5 -y 7 -y 11 -sum`". (Note that in the option specification, x arguments are specified as "sticky": that is, they immediately follow the "`-x=`" string, whereas the y arguments are said to be "separate", separated from the "`-y`" string by a some number of spaces.) The display of the sums of the argument inputs in this test case implies that the command-line parsing code used to obtain program inputs works correctly.

```
signature ROOT = sig
    ...
    (* existing: *)
    type style
    val styleOf : root -> style
    val styleFromStrings : root * string list -> style
    ...
    (* added: *)
    val mergeStyles : style * style -> style
    val styleFromXRDB : root -> style
    type optName
    type argName
    type optKind
    type optSpec
    type optDb
    type attr_value
    val parseCommand : optSpec -> string list ->
            optDb * string list
    val findNamedOpt : optDb -> optName -> root ->
            attr_value list
    val findNamedOptStrings : optDb -> optName ->
            string list
    val styleFromOptDb : root * optDb -> style
    ...
end
```

Figure 4.6: Functions added to widget `ROOT`

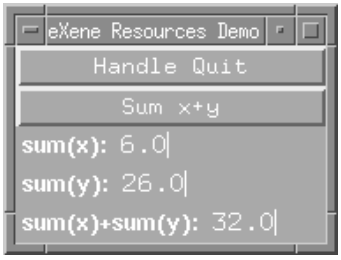


Figure 4.7: A screenshot of the resource customization demo application, with command line arguments "`-fg White -bg DarkGrey -x=1 -x=2 -x=3 -y 3 -y 5 -y 7 -y 11 -sum`" supplied.

We also wish to test the resource specification code. We may obtain resource specifications from three sources: the default resources specified in the application code, the resource specifications stored by `xrdb` in the X server, and arguments specified by the user on the command line. We wish for server resource specifications to take priority over application defaults, and for command line resource specifications to take priority over those specified by the server. Therefore, the application code first merges the server resources with those specified by the application, then merges the command line resource specifications with that resulting style.

The test case must therefore test not only the ability of the application's appearance to be customized from all three of these sources, but must test the order in which these resources are merged. Let us first test the resources specified by the application (note that this test does not involve any code newer than the latest eXene release). We may run the `demo-res` application and note that the background color is white and the foreground is black, as specified in the application (or we may alter these specifications, and see the changes reflected in the application).

Next, we wish to verify that the resources loaded into the X server by `xrdb` are applied. We may create a file, perhaps called `xresources`, and enter

some resource specification such as "`*background: yellow`" into it. Then, we run the command "`xrdb xresources`" to load these specifications into the X server, and we relaunch the `demo-res` application. The background of the application will now be yellow, overriding the background specified in the application.

Finally, we wish to verify that any resources specified on the command line are applied after those specifications provided by the X server. We may launch the application again, this time giving arguments such as "`-fg White -bg DarkGrey`". The application will appear as in the screenshot of Figure 4.7.

Chapter 5

X Selections

It is often desirable to exchange information between X client applications. For example, a user may wish to cut or copy information from one application and paste it into another. The Inter-Client Communications Conventions [16] (ICCC) documents two methods for the exchange of information between clients: the cut buffer and the selection mechanism.

The cut buffer mechanism is the simpler of the two. The cut buffer consists of eight properties with string text encoding on the first root window of the display. When an application wishes to store a string in the cut buffer, it rotates the values of the first seven properties to be the values of the last seven of the properties, and then stores the string in the first property.[1] Retrieving a value from the cut buffer is as simple as obtaining the value of the first cut buffer property from the X server.

EXene provides functions for getting and setting properties of a root window on a display, and a function for rotating properties. In this way, it is quite possible for an eXene application to make full use of cut buffers, provided the developer has some understanding of how the cut buffer mechanism

[1] In addition to the functions `XStoreBytes (display,string)` and `XFetchBytes (display)` which store and retrieve a string to the first property of the cut buffer, respectively, Xlib also provides the function `XRotateBuffers(display,integer)` by which the properties of the cut buffer may be rotated by an arbitrary offset, and functions for storing to and retrieving from arbitrary properties within the buffer[17].

is implemented. However, it would be helpful to have a few simple functions available in eXene, similar to Xlib's `XStoreBytes(display,string)` and `XFetchBytes(display)`, to work with the cut buffer[17, p. 476].

Cut buffers provide a simple mechanism for exchanging text between clients. However, the more general selection mechanism allows for greater flexibility in exchanging information. The ICCC Manual states, "Selections are a much more powerful and useful mechanism for interchanging data between clients and generally should be used instead of cut buffers." [17, p. 476]

Like the cut buffers, the selection mechanism also transfers information between clients via X window properties. In contrast to the cut buffer mechanism, though, selection values do not persist in those properties. A selection is owned by one client window, and other clients may request the value of that selection by providing a target type to convert the value to along with an empty property to store the value in. Several predefined selections are specified by the ICCC Manual, namely `PRIMARY`, `SECONDARY`, and `CLIPBOARD`, but any number of selections may exist on a given X server.[2]

To acquire a selection, a client window W (usually via some toolkit like Xlib) sends a `SetSelectionOwner` protocol request to the X server. This request must be accompanied by the timestamp of the event that triggered the request for selection ownership (perhaps the timestamp accompanying the keystroke or mouse event that signifies selecting a region). If W specifies a time later than the last time the selection was set or changed, the server releases the current selection and W is said to "own" the selection specified [17, p. 524]. Thereafter until W loses the selection to another window, any requests for the current selection are routed to W. When another client window W' requests the value of the current selection, it will provide a property to store the value in and a target type to convert the selection to; for exam-

[2]A selection is named by an "atom", a unique name represented by the X server as a 32-bit integer. [17, p. 608] Atoms are used for identifying many entities in X windows including selection names and selection target types.

ple, `STRING` indicating that the requestor wants the value as type string or `TIMESTAMP` indicating that the requestor wants the value of the timestamp that caused W to acquire the selection. If W is unable or unwilling to convert the selection value to a requested type, it may refuse to return any value [17, p. 617]. Otherwise, W will encode the selection value as the specified type, or some (hopefully similar) target type of its own choice, and store this converted value along with the corresponding actual type in the requestor W''s property. The requestor W' will then be notified via an event that the value has been stored (or that no value was converted or stored).

The ICCC Manual specifies that clients are required to support several targets, namely the `TARGETS`, `MULTIPLE`, and `TIMESTAMP` targets [17, p. 623]. When a requestor asks for a selection to be converted to type `TARGETS`, it is asking for a list of target types (X atoms) that the selection owner is willing to convert the selection to. The `MULTIPLE` target specifies that a list of target type, property pairs will accompany the conversion request, and requires the selection owner to proceed through the target list, in order, converting the selection to each target specified (and storing the converted value in the accompanying property).

In addition, selection owners must be capable of transferring large selection values incrementally in order to respect X server property size limits, and selection requestors must be capable of receiving values sent incrementally. The incremental transfer of values proceeds as follows: the selection owner responds to a conversion request by setting the property value to type `INCR`. When the selection requestor encounters this type, it deletes the contents of the property and waits until notification of a new property value. It then reads the value, deletes the value, and continues in this fashion until the new property value is of zero length (empty). The owner, meanwhile, initially waits for notification that the property has been cleared, stores a value in the property, then waits for that value to be cleared before continuing to store sub-parts of the value until the transfer is complete[17, p. 619].

5.1 Implementation of Selection Extensions

The current distribution of eXene contains basic support for aquiring and requesting selections. EXene contains a

```
acquireSelection : (window * atom * time)
    -> selection_handle option
```

function for acquiring the ownership of a selection, and a

```
selectionReqEvt : selection_handle -> {
        target : atom,
        time : time option,
        reply : XProtTypes.prop_val option -> unit,
        property : atom
      } CML.event
```

function for obtaining an event that will become enabled when a client requests the value of the selection. However, this leaves the responsibility for handling the target types required by the ICCC on the client as well as all the technical details of converting target types to raw property values (`prop_val`s). Similarly, clients may use `requestSelection` to request the conversion of a selection value to a target type, but the requestor must have intimate knowledge of what format the value will be stored as within the `prop_val` in order to convert to some workable data type. These non-trivial format conversions, along with the requirement that owners and requestors support incremental transfers, make it a bit difficult to author eXene clients or widgets that fully support X selection transfers.

In order to make it easier for eXene clients to support X selections, an eXene selection interface has been implemented that is similar to the existing one, but that allows clients to operate at a somewhat higher level. As figure 5.1 shows, datatypes have been created that specify common selection

```
structure ICCCSelection : sig
    datatype selection = Sel_PRIMARY
                       | Sel_SECONDARY
                       | Sel_CLIPBOARD
                       | Sel_OTHER of XProtTypes.atom
    datatype target = Tgt_TARGETS
                    | Tgt_MULTIPLE of (target list)
                    | Tgt_TIMESTAMP
                    | Tgt_STRING
                    | Tgt_LENGTH ...
    datatype value  = Val_TARGETS           of target list
                    | Val_MULTIPLE          of value list
                    | Val_TIMESTAMP         of XTime.time
                    | Val_STRING            of string
                    | Val_LENGTH            of int ...
    type convertfn : target -> value option
    val convertString : string -> target -> value option
    type selection_handle
    val releaseEvt : selection_handle -> unit CML.event
    val releaseSelection : selection_handle -> unit
    val acquireSelection : (Window.window * selection *
            XTime.time * convertfn) -> selection_handle option
    val requestSelection : (Window.window * selection *
            target * XTime.time) -> value option CML.event
    val requestSelectionString : (Window.window * selection *
            XTime.time) -> string option CML.event
    val storeCutBuffer : (Window.window * string) -> unit
    val fetchCutBuffer : (Window.window) -> string option
    val rotateCutBuffer : (Window.window * int) -> unit
  end
```

Figure 5.1: New selection functions

names, target types, and return types. Just as the existing `SELECTION` signature contains an `acquireSelection` function, so does this new interface. However, this function also takes a `convertfn` function as an argument. It is this conversion function's task, when a request for a selection value occurs, to convert the requested target type to `SOME` v where v is a valid `value`, or to `NONE` if the conversion is not possible. The conversion function may communicate via CML channels with a selection owner thread to obtain an up-to-date value of a dynamic selection value, or the conversion function may be created to statically hold the state of the selection value. Indeed, for the common case where a string value is selected, a `convertString` curried function is provided, so that a selection owner may simply furnish the string value of the selection and the `convertString` function applied to that string may then handle the conversions of that string to the value types that can reasonably be converted to (including string, compound text, hostname as a string, and length of the string).

In all cases, the server thread spawned by calling `acquireSelection` automatically handles the `Tgt_TIMESTAMP` target type even if the `convertfn` provided fails to handle that target. In addition, the list of target types supported by the provided `convertfn` should be revealed by applying the `convertfn` to `Tgt_TARGETS`, but the server thread will append the built-in `Tgt_TIMESTAMP` and `Tgt_MULTIPLE` targets to the list returned by the `convertfn` before returning the list to the requestor.

Like the existing interface, a `requestSelection` function is provided whereby clients can request the value of a particular selection. In contrast to the existing code, however, this function hides the allocation and deallocation of a property to use in the transfer of the selection value. In addition, it handles the details of incremental transfers, and converts the raw data of a `prop_val` to a value of type `value option`. In addition, for the common case where a client wishes to request the value of a selection as a string, a `requestSelectionString` function is provided.

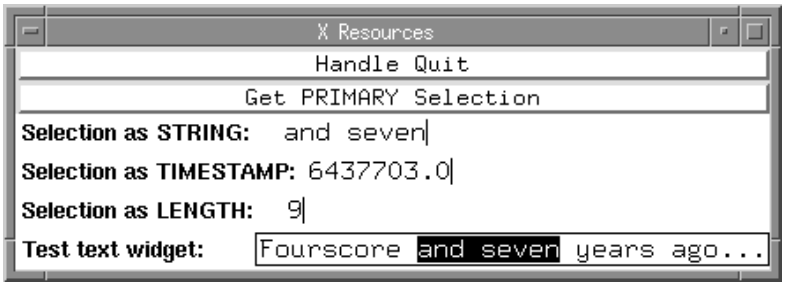


Figure 5.2: Selection demo application requesting multiple values

Finally, for those clients who wish to transfer data via the cut buffer, several "syntactic sugar" functions are provided whereby clients can set the string value of the first cut buffer, obtain the value of the first cut buffer, and rotate the cut buffers.

5.2 Example and Testing of Selection Extensions

The example `SimpleEdit` text widget (Figure 2.4) has been created to work with the improved selections interface. Text in a `SimpleEdit` widget may be selected by clicking and dragging the mouse over the text; this selected text will then become the `PRIMARY` selection.

We may demonstrate this functionality by the use of a simple application entitled `demo-sel`. This demo application shown in Figure 5.2 contains four `SimpleEdit` widgets – three that we may use to display results of selection requests, and one that we may use to create test selections. The application is written so that when the "Get PRIMARY Selection" button is clicked, a request for the primary selection is sent with target type `Tgt_MULTIPLE [Tgt_STRING, Tgt_TIMESTAMP, Tgt_LENGTH]` to the eXene selection server. If a reply is returned, the first text widget is set to the value of the selection as a string, the second widget is set to the timestamp at which the selection was acquired, and the third widget is set to the selection length. However, if no

valid reply is returned in response to the `MULTIPLE` request, the application will try a second request with only the target `Tgt_STRING` of which any valid returned value shall be displayed in the first text widget.

Let us perform the first test case entirely against the eXene selection code. When text from the fourth text widget is selected, it becomes the primary selection. This primary selection may be obtained as a string, as well as its length and the timestamp at which it was acquired, by pressing the "Get PRIMARY Selection" button. The requested attributes of the selection will then appear in the first three text widgets.

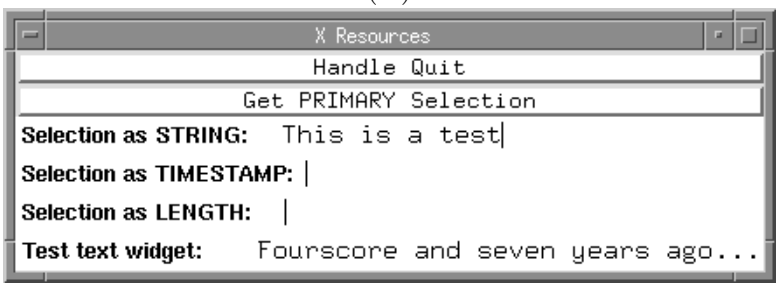
We may then test the application against other X applications to ensure interoperability. However, other X applications have not been found that support the `MULTIPLE` target type (all seem to return value `NONE` to a `requestSelection` request with `MULTIPLE` target type specified). We may still test requests with other target types, though. Figure 5.3 shows such a test case. We may select text in another X application, such as `xemacs`. We may then display the `Val_STRING` value of this `PRIMARY` selection by clicking the "Get PRIMARY Selection" button.

Unfortunately, even though support for incremental selection transfers has been added – both for selection acquisition and selection requests – testing revealed that this code does not function correctly. In order to perform an incremental transfer, we must be capable of monitoring an X property for new values and deletions. EXene provides a

```
datatype prop_change = NewValue | Deleted
val watchProperty : property -> (prop_change * time) CML.event
```

function for obtaining this event. Therefore, the incremental transfer functions rely heavily on this `watchProperty` function. However, it appears that watched property events never become enabled. If we execute code such as that shown in Figure 5.4, the diagnostic message is never printed. Although it is unfortunate that incremental transfers do not currently work, the central issue has been identified and it will hopefully soon be resolved.

(A)



(B)

Figure 5.3: Selection demo application requesting primary selection

```
let
val w = ... (* some window *)
val p = Property.unusedProperty w
fun f () = ((CML.sync (Property.watchProperty p));
            TextIO.print "Test property changed.\n")
val _ = CML.spawn f
in Property.deleteProperty p end
```

Figure 5.4: Code to test `watchProperty` function

The new selection extensions provide an easy-to-use interface for application or widget developers to add selection support to their code. The extensions hide many of the low-level details of selection transfers and provide ICCC compliance for the required target types. In addition, testing of these extensions proves most of them to be reliable and workable.

Chapter 6

X Authorization

When an X client application starts, it is provided with a "display" variable, either from a command line argument or from an environment variable, that specifies the X server to connect to. The syntax of the display variable is:

```
[protocol/][host]:server[.screen]
```

where `protocol` is the transport protocol by which the client should connect to the X server, `host` is the hostname on which the server is running and `server` refers to a specific X server instance on that host. Since a given X server instance may be capable of displaying graphics on multiple screens, the `screen` specifies which screen to initially display any windows on[13, p. 63]. The `server` and `screen` arguments are specified as integers; for example, `display` "localhost:0.0" specifies the first screen of the first server instance on host "localhost". Note that the `host` and `screen` arguments are optional - if `host` is omitted, the connecting code should default to the local host; if `screen` is omitted, the connecting code should default to the first screen ("0") [13, p. 64].

Using this information on what display to connect to, client code may then connect over some desired transport mechanism to the X server. Often, especially over network transport, it is necessary to provide some authorization information to the X server upon connecting so that unauthorized clients may not use the display. This authorization information – the name of the

2 bytes	Family value; common values include 0=`FamilyInternet`, 256=`FamilyLocal`, and 65535=`FamilyWild`
2 bytes	Address length A
A bytes	Host address; four bytes for an IP address or a string for a local host name
2 bytes	Display number length S
S bytes	Display number string
2 bytes	Authorization name length N
N bytes	Authorization name string
2 bytes	Authorization data length D
D bytes	Authorization data string

Table 6.1: The fields and field lengths of a record of an `XAUTHORITY` file [15] [19, xc/lib/xtrans/Xtransutil.c].

authorization protocol being used, and some data proving authorized status – is provided within the first request on connection setup [17, p. 500]. Therefore, the X client must obtain this information prior to connection setup.

Typically, X authorization information is obtained from a file named in a user's `XAUTHORITY` environment variable, or if no file is named there, a file named `.Xauthority` found in the user's home directory. (Let us refer to this file simply as the `XAUTHORITY` file in the remainder of this chapter.) This file is to contain a series of records, each containing a connection family field (Local/Unix Socket, Internet, etc.), host address field, display number field, authorization name field, and authorization data field, as shown in Table 6.1. It is then the task of the client, or the toolkit that the client is built upon, to search this file for the correct authorization data to supply.

While testing eXene applications, especially applications to be displayed over a tunneled SSH connection, we encountered a few issues with eXene. It seems that eXene would often fail to locate the correct authorization information to supply when initializing the connection, and the connection would be denied.

In an attempt to resolve the issues with eXene, I examined the Xlib source code [19]. When opening a connection to an X server, Xlib first determines the protocol, if supplied, and the hostname from the display variable. If no protocol is supplied, Xlib chooses the most efficient connection protocol with respect to the hostname supplied (for example, if "`localhost`" is supplied, a local connection is preferable to a TCP/IP connection). EXene also chooses between possible connection protocols; no issues have been encountered with this code. In Xlib, after a connection protocol is chosen a connection is created. Prior to sending the initialization request, however, Xlib determines the authorization data it must send. It does this by determining the peer address of the connection (or the local system name if a local connection), and this address string is then passed to a `GetAuthByAddr` function that searches for the correct authorization data in the `XAUTHORITY` file. EXene has a counterpart `getAuthByAddr` that serves the same function. Once this authorization data is obtained, it is supplied to the X server over the newly created connection.

In the current version of eXene, however, the hostname passed to eXene's `getAuthByAddr` function for comparison with the entries of the `XAUTHORITY` file is obtained from the display variable. This means, for example, that if the display variable were "`:0.0`" for connecting to the first server on the local host, the empty string would be passed to `getAuthByAddr`; or if the display variable were "`linux.cis.ksu.edu:23.0`" for an ssh-tunnelled connection, the string "`linux.cis.ksu.edu`" would be passed to `getAuthByAddr`. In either case, no entry would be found in the `XAUTHORITY` file. In the first case, we should search for a `FamilyLocal` family record in which the host address field is set to the full system name of the host, which would never match the empty string. Or, in the second case, we should search records with `FamilyInternet` family with a host address equal to "`linux.cis.ksu.edu`" – but `FamilyInternet` records store the four bytes of the IP address in the host address field rather than a full host name.

These issues have been resolved by ensuring that the address string sent to `getAuthByAddr` matches the connection method to be used. That is, if a local connection is to be created, the full system name of the local host is sent to `getAuthByAddr`, and if an Internet connection is to be created, a string containing the IP address (in standard dotted notation) of the resolved display hostname is sent to `getAuthByAddr`. It was also necessary to modify `getAuthByAddr` to unpack the four host IP address bytes of `FamilyInternet` records into a standard dotted notation string for comparison.

Chapter 7

Conclusion

EXene is an important toolkit for CML programmers: it allows development with all the advantages of the strongly-typed, functional CML when creating easy-to-use graphical user interfaces. It has been my goal in this thesis to address the most troublesome issues with the current release of eXene, so that the eXene programmer may find the eXene toolkit as easy to use as possible.

In view of the test cases performed on the extensions added, it may be concluded that the extensions are reliable and helpful. It is the goal of the widget programming conventions to provide a framework for the development of robust, concurrently operating widgets, and the test cases performed demonstrate that they are capable of providing that robustness.

The input focus extensions were designed to make the user experience more enjoyable. By providing a simple interface for the application developer to use widgets that obtain keyboard input focus, users will find eXene applications easier to use and more accessible.

Command line parsing extensions provided will make it easier for application developers to accept user input via command line arguments, in addition to providing avenues for customizing the look-and-feel of applications at runtime. In addition, extensions added to obtain resource specifications from the X server, and methods to merge these resulting styles, makes it easier

for a developer to create applications that will respond similarly to other X applications.

Just as it is important to provide easy interfaces for developers to assign input focus to widgets and to easily customize application resources, it is also important to provide extensions for easily acquiring and requesting the value of X selections. These extensions must also comply with the requirements of the Inter-Client Communications Conventions. With the new selections functions provided, developers may work with selections with assurance that ICCC compliance is provided, as well as concerning themselves as little as possible with the low-level details of X selections.

Finally, miscellaneous improvements such as fixes to the eXene authorization code were necessary in order that users could use eXene applications in a wide variety of environments.

These improvements will greatly improve the usability of the eXene toolkit. There is certainly much room for future work, of course: many existing eXene widgets should be rewritten to follow the widget programming conventions, and miscellaneous bugs will surely appear over time in the eXene toolkit that must be fixed. However, it is the author's opinion that eXene is well on its way to becoming a widely-used basis for the development of SML applications.

An archive file containing the eXene source code updated in the course of this thesis work will be available at:

`http://www.cis.ksu.edu/~stough/eXene/index.html`

References

[1] D. deBoer and A. Stoughton. On the Future of eXene. `http://www.cis.ksu.edu/~stough/eXene/future.pdf`, 2005.

[2] E. M. Gansner. Notes on the new eXene widgets. Included as part of version 1.0 of the eXene distribution, 1995.

[3] E. M. Gansner and J. H. Reppy. eXene. In *1991 CMU Workshop on SML*, 1991.

[4] E. M. Gansner and J. H. Reppy. *The eXene widgets manual*. AT&T Bell Laboratories, February 1993.

[5] E. R. Gansner and J. H. Reppy. A multi-threaded higher-order user interface toolkit. In Bass and Dewan, editors, *User Interface Software*, volume 1 of *Software Trends*. Wiley, 1993.

[6] D. Haahr. Montage: Breaking windows into small pieces. In *USENIX Summer Conference*, pages 289-297, USENIX Association, June 1990.

[7] A. Nye. *Xlib programming manual*, volume 1 of *The definitive guides to the X window system*. O'Reilly & Associates, Inc., third edition, 1992.

[8] A. Nye. *Xlib reference manual*, volume 2 of *The definitive guides to the X window system*. O'Reilly & Associates, Inc., third edition, 1992.

[9] A. Nye. *X protocol reference manual*, volume 0 of *The definitive guides to the X window system*. O'Reilly & Associates, Inc., fourth edition, 1995.

[10] A. Nye and T. O'Reilly. *X toolkit intrinsics programming manual*, volume 4 of *The definitive guides to the X window system*. O'Reilly & Associates, Inc., second edition, 1992.

[11] A. Nye and T. O'Reilly. *X toolkit intrinsics reference manual*, volume 5 of *The definitive guides to the X window system*. O'Reilly & Associates, Inc., second edition, 1992.

[12] A. Nye and T. O'Reilly. *X toolkit intrinsics programming manual*, volume 4 of *The definitive guides to the X window system*. O'Reilly & Associates, Inc., Motif edition, 1993.

[13] V. Quercia and T. O'Reilly. *X Window System User's Guide*, volume 3 of *The Definitive Guides to the X Window System*. O'Reilly & Associates, Inc., Motif edition, 1993.

[14] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[15] J. H. Reppy and J. Buntrock. xauth.sml. Included as part of version 1.0 of the eXene distribution, 1995.

[16] D. Rosenthal. *The inter-client communication conventions manual, Version 2.0*. Sun Microsystems, Inc., December 1993.

[17] R. W. Scheifler and J. Gettys. *X window system: the complete reference to Xlib, X protocol, ICCCM, and XLFD.* Digital Press, third edition, 1992.

[18] J. D. Ullman. *Elements of ML Programming.* Prentice Hall, 1998.

[19] The XFree86 Project, Inc. XFree86 4.4.0 source release. The XFree86 Project, Inc., February 2004.

Appendix A

SimpleEdit source code

```
(* simple-edit.sml
 * Based on str-edit.sml,
 * COPYRIGHT (c) 1991 by AT&T Bell Laboratories  See COPYRIGHT file for details.
 *
 * Simple string edit widget, spring 2005, Dusty deBoer, Kansas State University.
 *)

signature SIMPLEEDIT =
  sig

    structure W : WIDGET

    type simple_edit

    val simpleEdit : (W.root * Widget.view * Widget.arg list) ->
            string -> simple_edit

    val setString : simple_edit -> string -> unit
    val getString : simple_edit -> string
    val setSelection : simple_edit -> (int * int * W.EXB.XTime.time) -> unit
    val getSelection : simple_edit -> string
    val widgetOf : simple_edit -> W.widget

    val takeFocus : simple_edit * W.EXB.XTime.time -> unit
    val focusableOf : simple_edit -> Shell.focusable

  end (* SIMPLEEDIT *)

structure SimpleEdit : SIMPLEEDIT =
  struct
```

```
  structure EXB = EXeneBase
  structure W = Widget
  structure S = Shell

  open CML Geometry EXeneWin Interact Drawing EXeneBase

  val attrs = [
      ([], Attrs.attr_font,         Attrs.AT_Font,    Attrs.AV_Str "9x15"),
      ([], Attrs.attr_background,   Attrs.AT_Color,   Attrs.AV_Str "white"),
      ([], Attrs.attr_foreground,   Attrs.AT_Color,   Attrs.AV_Str "black")
    ]

  val min = Int.min
  val max = Int.max

  datatype rqst
    = GetString
    | GetBounds
    | SetString of string
    | SetSelection of (int * int * W.EXB.XTime.time)
    | GetSelection
    | DoRealize of {
        env : Interact.in_env,
        win : EXB.window,
        sz : size
      }
    | TakeFocus of EXB.XTime.time

  datatype reply
    = Bnds of W.bounds
    | BndsExn
    | Str of string

datatype simple_edit = SimpleEdit of (W.widget * rqst chan * reply chan
      * S.focusable_msg chan)

fun simpleEdit (root,view as (name,style),args) initval =
  let
  val view as (name,style) = (Styles.extendView (name,"simpleedit"),style)
  val fattrs = W.findAttr (W.attrs(view,attrs,args))
  val font = Attrs.getFont (fattrs Attrs.attr_font)
  val backc = Attrs.getColor (fattrs Attrs.attr_background)
  val forec = Attrs.getColor (fattrs Attrs.attr_foreground)
  val penn = newPen [PV_Foreground forec, PV_Background backc]
  val penb = newPen [PV_Foreground backc, PV_Background backc]
  val pens = newPen [PV_Foreground backc, PV_Background forec]
```

```
val reqChan = channel () and repChan = channel ()
val focChan : S.focusable_msg chan  = channel ()
val {ascent=fonta,descent=fontd} = Font.fontHt font
val fonth = fonta + fontd
val Font.CharInfo{left_bearing=lb,char_wid=fontw,...} =
        Font.charInfoOf font (Char.ord #"A")
fun bound (x,y,z) = max(x,min(y,z))
fun pttopos (str,PT{x,y}) = bound(0, x div fontw, String.size str)
fun getbnds slen =
  let
  val wid = slen*fontw
  in {x_dim=W.DIM{base=0,incr=1,min=wid,nat=wid,max=NONE},
        y_dim=W.fixDim fonth} end
fun realizeSimpEdit {env=InEnv{m,k,ci,co},win,sz=SIZE{wid,ht}}
            {str,selpos,sellen,seltime} =
    let
    val rq = recvEvt reqChan
    val dst = drawableOfWin win
    fun setsel (str,ss,sl) xt =
        let
        val str = String.substring (str,ss,sl)
        in if sl>0 then (case
            (ICCC.acquireSelection (win,ICCC.Sel_PRIMARY,xt,
            (ICCC.convertString str))) of
             SOME sh => ((ICCC.releaseEvt sh),
                        (fn () => (ICCC.releaseSelection sh)) )
           | NONE    => (alwaysEvt (), (fn () => ()) ))
            else (never, (fn () => ()) ) end
    val (initsre,initsrf) =
        case seltime of
          SOME xt => (setsel (str,selpos,sellen) xt)
        | _ => (never,fn () => ())
    fun redraw {str,selpos=ss,sellen=sl,selre,selrf,wid,ht} =
        let
        val (x,y) = (0,fonta)
        val cx = x+(fontw*ss)
        val se = ss+sl
        val ll = (String.size str)-se
        val (p1,s1) = (PT{x=x,y=y},String.substring (str,0,ss))
        val (p2,s2) = (PT{x=(x+(fontw*ss)),y=y},String.substring (str,ss,sl))
        val (p3,s3) = (PT{x=(x+(fontw*se)),y=y},String.substring (str,se,ll))
        in  (* clearDrawable dst; *)
            fillRect dst pens (RECT{x=0,y=0,wid=wid,ht=ht});
            imageString dst penn font (p1,s1);
            if (sl > 0)
             then (imageString dst pens font (p2,s2)) else ();
```

```
        imageString dst penn font (p3,s3);
        if (sl = 0)
         then drawSeg dst penn (LINE(PT{x=cx,y=0},PT{x=cx,y=fonth})) else ()
    end
fun handleMse (MOUSE_FirstDown {pt,but,time,...},
        me as {str,selpos,sellen,selrf,selre,wid,ht}) =
    let
    val _ = selrf()
    in send (focChan, (S.Assign time));
       {str=str,selpos=(pttopos (str,pt)),sellen=0,selrf=(fn ()=>()),
        selre=never,wid=wid,ht=ht} end
  | handleMse (MOUSE_LastUp {pt,but,time,...}, {str,selpos=ss,wid,ht,...}) =
    let
    val se = pttopos (str,pt)
    val (ss,sl) = if (se<ss) then (se,ss-se) else (ss,se-ss)
    val (sre,srf) = (setsel (str,ss,sl) time)
    val me' = {str=str,selpos=ss,sellen=sl,selrf=srf,selre=sre,wid=wid,ht=ht}
    in (redraw me'; me') end
  | handleMse (_,me) = me
fun handleKey (KEY_Press key, me as {str,selpos=ss,sellen=sl,selrf,selre,wid,ht}) =
    (case key of
        (KEYSYM(65289),_,xt) => (* tab *)
            (send(focChan,S.Next xt); me)
      | (KEYSYM(65056),_,xt) => (* shift+tab *)
            (send(focChan,S.Previous xt); me)
      | _ => let
             val s1 = String.substring (str,0,ss)
             val s2 = lookupString defaultTranslation key
                handle KeysymNotFound => ""
             val s3 = String.substring (str,ss+sl,(String.size str)-ss-sl)
             val (s1,s2,ss) = if (s2 = "\^H") then (* backspace *)
                  (String.substring (s1,0,max((String.size s1)-1,0)),"",
                    max(ss-1,0))
                  else if (s2 = "\^X") then (* kill *)
                  (s1,"",ss) else (s1,s2,ss)
             val me = {str=(s1^s2^s3),selpos=ss+(String.size s2),sellen=0,
                       selre=never,selrf=(fn ()=>()),wid=wid,ht=ht}
             val _ = selrf() (* release the current selection if we have one *)
             in redraw me; me end
    )
  | handleKey (_,me) = me
fun handleCI (CI_Resize (RECT{wid,ht,...}),
        {str,selpos,sellen,selre,selrf,wid=ow,ht=oh}) =
            let
            val me' = {str=str,selpos=selpos,sellen=sellen,selre=selre,
                selrf=selrf,wid=wid,ht=ht}
```

```
           in (redraw me'; me') end
  | handleCI (CI_Redraw _, me) = (redraw me; me)
  | handleCI (CI_FocusIn, me)  = (CML.send(focChan,S.FocusIn); me)
  | handleCI (CI_FocusOut, me) = (CML.send(focChan,S.FocusOut); me)
  | handleCI (_, me) = me
fun handleReq (GetString, me as {str,...}) =
        (send(repChan, Str str); me)
  | handleReq (GetSelection, me as {str,selpos=ss,sellen=sl,...}) =
        (send(repChan, Str (String.substring (str,ss,sl))); me)
  | handleReq (GetBounds, me) =
        (send(repChan, BndsExn); me) (* bounds function should not be called now *)
  | handleReq (SetString s, {selrf,wid,ht,...}) =
        let
        val _ = selrf()
        val me' = {str=s,selpos=(String.size s),sellen=0,
                   selrf=(fn ()=>()),selre=never,wid=wid,ht=ht}
        in redraw me'; me' end
  | handleReq (SetSelection (a,b,xt), {str,selrf,wid,ht,...}) =
        let
        val _ = selrf()
        val mx = String.size str
        val a = bound(0,a,mx)
        val (ss,sl) = (a,bound(0,b-a,mx-a))
        val (sre,srf) = (setsel (str,ss,sl) xt)
        val me' = {str=str,selpos=ss,sellen=sl,selre=sre,
                   selrf=srf,wid=wid,ht=ht}
        in redraw me'; me' end
  | handleReq (DoRealize _, me) =
        (raise W.AlreadyRealized; me)
  | handleReq (TakeFocus(xt), me) =
        (EXeneWin.setInputFocus(win,xt); me)
fun handleSelRel {str,selpos=ss,sellen,selrf,selre,wid,ht} =
        let
        val me' = {str=str,selpos=ss,sellen=0,selre=never,selrf=(fn ()=>()),
            wid=wid,ht=ht}
        in redraw me'; me' end
fun loop (me as {selre,...}) =
  loop (select [
    wrap (m,  fn evt => handleMse (msgBodyOf evt,me)),
    wrap (k,  fn evt => handleKey (msgBodyOf evt,me)),
    wrap (ci, fn evt => handleCI  (msgBodyOf evt,me)),
    wrap (rq, fn evt => handleReq (evt,me)),
    wrap (selre, fn () => (handleSelRel me))
  ])
in
  loop {str=str,selpos=selpos,sellen=sellen,selre=initsre,selrf=initsrf,
```

```
          wid=wid,ht=ht}
      end
  fun initLoop (me as {str,selpos,sellen,seltime}) =
    case recv reqChan of
      GetString =>
       (send(repChan, Str str); initLoop me)
    | GetSelection =>
       (send(repChan, Str (String.substring (str,selpos,sellen))); initLoop me)
    | GetBounds =>
       (send(repChan, Bnds (getbnds (size str))); initLoop me)
    | SetString str' =>
       (initLoop {str=str',selpos=(String.size str'),sellen=0,seltime=seltime})
    | SetSelection (ss,se,xt) =>
      let
      val mx = String.size str
      val ss = bound(0,ss,mx)
      val sl = bound(0,(se-ss),(mx-ss))
      in (initLoop {str=str,selpos=ss,sellen=sl,seltime=SOME xt} ) end
    | DoRealize arg =>
       (realizeSimpEdit arg me)
    | TakeFocus(xt) =>
       (initLoop me) (* should perhaps set flag for taking focus upon realization. *)
  in
    spawn (fn () =>
            (initLoop {str=initval,selpos=(String.size initval),sellen=0,seltime=NONE};
            ()));
    SimpleEdit (
      W.mkWidget{
        root=root,
        args= fn () => {background = NONE},
        boundsOf = fn () => (
          send (reqChan, GetBounds);
          case recv repChan of
            Bnds b => b
          | BndsExn => raise W.BoundsFunctionAlreadyCalled
          | Str _ => raise LibBase.Impossible "StrEdit.mkStrEdit"
        ),
        realize = (fn arg => (send (reqChan, DoRealize arg)))
      },
      reqChan,
      repChan,
      focChan
    )
  end

fun widgetOf (SimpleEdit(widget,_,_,_)) = widget
```

```
  fun setString (SimpleEdit(_,reqc,_,_)) arg = (send (reqc, SetString arg))
  fun setSelection (SimpleEdit(_,reqc,_,_)) arg = (send (reqc, SetSelection arg))

  fun getString (SimpleEdit(_,reqc,repc,_)) = (
    send (reqc, GetString);
    case recv repc of
      Str s => s
    | _ => raise LibBase.Impossible "SimpleEdit.getString"
  )
  fun getSelection (SimpleEdit(_,reqc,repc,_)) = (
    send (reqc, GetSelection);
    case recv repc of
      Str s => s
    | _ => raise LibBase.Impossible "SimpleEdit.getSelection"
  )
  (* added ddeboer, spring 2005 *)
  fun takeFocus (SimpleEdit(_,reqc,_,_),xt) = (send (reqc,TakeFocus(xt)))
  fun focusableOf (SimpleEdit(widget,reqc,_,fc)) =
        S.Focusable {focusableEvt=(WidgetBase.wrapQueue (recvEvt fc)),
                     takefocus=(fn xt => (send (reqc,TakeFocus(xt))) )}
end (* StrEdit *)
```

Appendix B

Widget Conventions Demo source code

```
(*
 * Dusty deBoer, Kansas State University.
 *
 * Based on basicwin.sml, (C) 1990 J.H. Reppy; and goodbye.sml, (C) 1990 AT&T.
 *)

structure XDEMO : sig

    val doit  : string option * string list -> OS.Process.status
    val main  : (string * string list) -> OS.Process.status

  end = struct

    structure EXB = EXeneBase

fun init (dpyOpt,args) =
    let
    val root  = Widget.mkRoot(GetDpy.getDpy(dpyOpt))
        handle EXB.BadAddr s =>
            (TextIO.print s; RunCML.shutdown OS.Process.failure)

    val styleView = Styles.mkView{name = Styles.styleName["democonv"],aliases = nil}
    val view      = (styleView, (Widget.styleOf root))

    fun quit ()  = (Widget.delRoot root; RunCML.shutdown OS.Process.success)

    val quitBttn1 = Button.textBtn (root, view,
                        [([], Attrs.attr_label, Attrs.AV_Str "Handle Quit")])
    val quitEvt1  = Button.evtOf quitBttn1

    val quitBttn2 = Button.textBtn (root, view,
```

```
                    [([], Attrs.attr_label, Attrs.AV_Str "Ignore Quit")])
  val quitEvt2  = Button.evtOf quitBttn2

  val slowBttn = TestButton.textBtn (root, view,
                    [([], Attrs.attr_label, Attrs.AV_Str "Slowly Quit")])
  val slowEvt  = TestButton.evtOf slowBttn

  val layout =
      Box.layout (root, view, []) (Box.VtCenter[
          Box.WBox(Button.widgetOf quitBttn1),
          Box.WBox(Button.widgetOf quitBttn2),
          Box.WBox(Button.widgetOf slowBttn)
      ])
  val shellArgs =
      [([], Attrs.attr_title, Attrs.AV_Str "eXene Conventions Demo"),
       ([], Attrs.attr_iconName, Attrs.AV_Str "demo-conv")]
  val shell = Shell.shell (root, view, shellArgs) (Box.widgetOf layout)

  val hints = Shell.mkHints{size_hints=[],wm_hints=[ICCC.HINT_Input(true)]}
  val _     = Shell.setWMHints shell hints
  val cmEvt = Shell.deletionEvent shell

  fun loop():unit =
      let
      fun handleQuit (Button.BtnUp _) = (TextIO.print " [demo-conv quitting]\n";
              quit())
        | handleQuit (_) = (loop())
      in CML.select
         [CML.wrap(quitEvt1, handleQuit),
          (* Events from the second quit button shall be ignored:
           * CML.wrap(quitEvt2, handleQuit),*)
          CML.wrap(cmEvt, quit),
          (* events from the slow button will be handled when received. *)
          CML.wrap(slowEvt, handleQuit)
        ]
      end
  in
      Shell.init shell;
      loop()
  end

fun doit (dpyOpt,args) =
    (RunCML.doit (fn () => (init (dpyOpt,args)), NONE))

fun main (prog, "-display"::(server::args)) =
          ((TextIO.print ("display="^server)); doit(SOME server,args))
```

```
    | main (prog, args) = doit(NONE,args)

end
```

Appendix C

Input Focus Demo source code

```
(*
 * Dusty deBoer, Kansas State University.
 *
 * Based on basicwin.sml, (C) 1990 J.H. Reppy; and goodbye.sml, (C) 1990 AT&T.
 *)

structure XDEMO : sig

    val doit  : string option * string list -> OS.Process.status
    val main  : (string * string list) -> OS.Process.status

  end = struct

    structure EXB = EXeneBase

fun init (dpyOpt,args) =
    let
    val root  = Widget.mkRoot(GetDpy.getDpy(dpyOpt))
        handle EXB.BadAddr s =>
            (TextIO.print s; RunCML.shutdown OS.Process.failure)

    val styleView = Styles.mkView{name = Styles.styleName["demofocus"],aliases = nil}
    val view      = (styleView, (Widget.styleOf root))

    fun quit ()  = (Widget.delRoot root; RunCML.shutdown OS.Process.success)

    val quitBttn1 = Button.textBtn (root, view,
                    [([], Attrs.attr_label, Attrs.AV_Str "Handle Quit")])
    val quitEvt1  = Button.evtOf quitBttn1

    val smplEdit1 = SimpleEdit.simpleEdit (root, view, []) "111"
```

```
val smplEdit2 = SimpleEdit.simpleEdit (root, view, []) "222"
val smplEdit3 = SimpleEdit.simpleEdit (root, view, []) "333"
val smplEdit4 = SimpleEdit.simpleEdit (root, view, []) "444"
val smplEdit5 = SimpleEdit.simpleEdit (root, view, []) "555"

val ff1 = FocusFrame.focusframe (root,view,[])
    ((SimpleEdit.widgetOf smplEdit1),(SimpleEdit.focusableOf smplEdit1))
val ff2 = FocusFrame.focusframe (root,view,[])
    ((SimpleEdit.widgetOf smplEdit2),(SimpleEdit.focusableOf smplEdit2))
val ff3 = FocusFrame.focusframe (root,view,[])
    ((SimpleEdit.widgetOf smplEdit3),(SimpleEdit.focusableOf smplEdit3))
val ff4 = FocusFrame.focusframe (root,view,[])
    ((SimpleEdit.widgetOf smplEdit4),(SimpleEdit.focusableOf smplEdit4))
val ff5 = FocusFrame.focusframe (root,view,[])
    ((SimpleEdit.widgetOf smplEdit5),(SimpleEdit.focusableOf smplEdit5))

val layout =
    Box.layout (root, view, []) (Box.VtCenter[
        Box.WBox(Button.widgetOf quitBttn1),
        Box.WBox(FocusFrame.widgetOf ff1),
        Box.WBox(FocusFrame.widgetOf ff2),
        Box.WBox(FocusFrame.widgetOf ff3),
        Box.WBox(FocusFrame.widgetOf ff4),
        Box.WBox(FocusFrame.widgetOf ff5)
    ])
val shellArgs =
    [([], Attrs.attr_title, Attrs.AV_Str "eXene Focus Demo"),
     ([], Attrs.attr_iconName, Attrs.AV_Str "demo-focus")]
val shell = Shell.shell (root, view, shellArgs) (Box.widgetOf layout)

val id1 = Shell.addFocusableFirst shell (FocusFrame.focusableOf ff1)
val id2 = Shell.addFocusableAfter shell (id1,FocusFrame.focusableOf ff2)
val id3 = Shell.addFocusableAfter shell (id2,FocusFrame.focusableOf ff3)
val id4 = Shell.addFocusableAfter shell (id3,FocusFrame.focusableOf ff4)
val id5 = Shell.addFocusableAfter shell (id4,FocusFrame.focusableOf ff5)

val hints = Shell.mkHints{size_hints=[],wm_hints=[ICCC.HINT_Input(true)]}
val _     = Shell.setWMHints shell hints
val cmEvt = Shell.deletionEvent shell

fun loop():unit =
    let
    fun handleQuit (Button.BtnUp _)   = (TextIO.print " [demo-focus quitting]\n";
            quit())
      | handleQuit (_) = (loop())
    in CML.select
```

```
          [CML.wrap(quitEvt1, handleQuit),
           CML.wrap(cmEvt, quit)
          ]
        end
    in
        Shell.init shell;
        loop()
    end

  fun doit (dpyOpt,args) =
      (RunCML.doit (fn () => (init (dpyOpt,args)), NONE))

  fun main (prog, "-display"::(server::args)) =
           ((TextIO.print ("display="^server)); doit(SOME server,args))
    | main (prog, args) = doit(NONE,args)

end
```

Appendix D

Resource Demo source code

```
(*
 * Dusty deBoer, Kansas State University.
 *
 * Based on basicwin.sml, (C) 1990 J.H. Reppy; and goodbye.sml, (C) 1990 AT&T.
 *)

structure XDEMO : sig

    val doit  : string option * string list -> OS.Process.status
    val main  : (string * string list) -> OS.Process.status

  end = struct

    structure EXB = EXeneBase

(* set up the option spec table. *)
val optSpec =
   [(Styles.OPT_NAMED("help"), "-help",   Styles.OPT_NOARG("on"),  Attrs.AT_Bool),
    (Styles.OPT_NAMED("help"), "-nohelp", Styles.OPT_NOARG("off"), Attrs.AT_Bool),
    (Styles.OPT_NAMED("x"),    "-x=",     Styles.OPT_STICKYARG,    Attrs.AT_Real),
    (Styles.OPT_NAMED("y"),    "-y=",     Styles.OPT_STICKYARG,    Attrs.AT_Real),
    (Styles.OPT_NAMED("res"),  "-res",    Styles.OPT_RESARG,       Attrs.AT_Str),
    (Styles.OPT_NAMED("skip"), "-skip",   Styles.OPT_SKIPARG,      Attrs.AT_Str),
    (Styles.OPT_NAMED("ign"),  "-ignore", Styles.OPT_SKIPLINE,     Attrs.AT_Str),
    (Styles.OPT_RESSPEC("*background"), "-background", Styles.OPT_SEPARG, Attrs.AT_Str),
    (Styles.OPT_RESSPEC("*background"), "-bg",         Styles.OPT_SEPARG, Attrs.AT_Str),
    (Styles.OPT_RESSPEC("*foreground"), "-foreground", Styles.OPT_SEPARG, Attrs.AT_Str),
    (Styles.OPT_RESSPEC("*foreground"), "-fg",         Styles.OPT_SEPARG, Attrs.AT_Str),
    (Styles.OPT_RESSPEC("*borderWidth"),"-border",     Styles.OPT_SEPARG, Attrs.AT_Str)]

(* set up application resource defaults. *)
```

```
val appResources =
   ["*background: #ddd",
    "*foreground: black"]

fun init (dpyOpt,args) =
    let
    val root  = Widget.mkRoot(GetDpy.getDpy(dpyOpt))
        handle EXB.BadAddr s =>
            (TextIO.print s; RunCML.shutdown OS.Process.failure)

    (* parse the command line arguments using the option spec table. *)
    val (optDb,unargs) = Widget.parseCommand (optSpec) args

    (* obtain the value of a named argument.
     * note that in this case we let the last argument (the head of the returned list)
     * override any previous arguments. *)
    val help  = (case (Widget.findNamedOpt optDb (Styles.OPT_NAMED("help")) root) of
                   [] => false               (* application must supply default here. *)
                 | Attrs.AV_Bool(b)::_ => b) (* let the last argument override. *)

    (* obtain the value of a OPT_STICKYARG argument.
     * note that in this case, we use every argument value given. *)
    val sumx = (List.foldl (fn (Attrs.AV_Real(r),s) => (r+s)) 0.0
                (Widget.findNamedOpt optDb (Styles.OPT_NAMED("x")) root))
    val sumy = (List.foldl (fn (Attrs.AV_Real(r),s) => (r+s)) 0.0
                (Widget.findNamedOpt optDb (Styles.OPT_NAMED("y")) root))

    (* create a style from the application default resource table. *)
    val appStyle = Widget.styleFromStrings(root,appResources)
        handle Styles.PRS.BadSpec (n,s) =>
           (TextIO.print "bad resource specification: ";
            TextIO.print(Int.toString n); TextIO.print (":"^s^"\n");
            Widget.delRoot root; RunCML.shutdown OS.Process.failure)

    (* create a style from the properties stored by xrdb. *)
    val xrdStyle = Widget.styleFromXRDB(root)
        handle Styles.PRS.BadSpec (n,s) =>
           (TextIO.print "bad resource specification: ";
            TextIO.print(Int.toString n); TextIO.print (":"^s^"\n");
            Widget.delRoot root; RunCML.shutdown OS.Process.failure)

    (* create a style from the resource options in the option db. *)
    val argStyle = Widget.styleFromOptDb(root,optDb)
        handle Styles.PRS.BadSpec (n,s) =>
           (TextIO.print "bad resource specification: ";
            TextIO.print(Int.toString n); TextIO.print (":"^s^"\n");
```

```
        Widget.delRoot root; RunCML.shutdown OS.Process.failure)

(* Merge: xrdb strings with app style, overwriting any conflicting app styles.
 * Then merge arg style with the result, giving priority to runtime args. *)
val mainStyle = Widget.mergeStyles(argStyle,Widget.mergeStyles(xrdStyle,appStyle))

val styleView = Styles.mkView{name = Styles.styleName["demores"],aliases = nil}
val view      = (styleView, mainStyle)

(* widget setup. *)
fun quit ()  = (Widget.delRoot root; RunCML.shutdown OS.Process.success)

val quitBttn = Button.textBtn (root, view,
        [([], Attrs.attr_label, Attrs.AV_Str "Handle Quit")])
val quitEvt  = Button.evtOf quitBttn

val sumBttn = Button.textBtn (root, view,
        [([], Attrs.attr_label, Attrs.AV_Str "Sum x+y")])
val sumEvt  = Button.evtOf sumBttn

val editArgs = []
val smplEdit1 = SimpleEdit.simpleEdit (root, view, editArgs) (Real.toString sumx)
val smplEdit2 = SimpleEdit.simpleEdit (root, view, editArgs) (Real.toString sumy)
val smplEdit3 = SimpleEdit.simpleEdit (root, view, editArgs) ""

val ff1 = FocusFrame.focusframe (root,view,[])
    ((SimpleEdit.widgetOf smplEdit1),(SimpleEdit.focusableOf smplEdit1))
val ff2 = FocusFrame.focusframe (root,view,[])
    ((SimpleEdit.widgetOf smplEdit2),(SimpleEdit.focusableOf smplEdit2))
val ff3 = FocusFrame.focusframe (root,view,[])
    ((SimpleEdit.widgetOf smplEdit3),(SimpleEdit.focusableOf smplEdit3))

val lbl1 = Label.label (root, view, [])
val _    = Label.setLabel lbl1 (Label.Text("sum(x): "))
val lbl2 = Label.label (root, view, [])
val _    = Label.setLabel lbl2 (Label.Text("sum(y): "))
val lbl3 = Label.label (root, view, [])
val _    = Label.setLabel lbl3 (Label.Text("sum(x)+sum(y): "))

val layout =
    Box.layout (root, view, []) (Box.VtCenter[
        Box.WBox(Button.widgetOf quitBttn),
        Box.WBox(Button.widgetOf sumBttn),
        Box.WBox (Box.widgetOf (Box.layout (root,view,[])
         (Box.HzCenter[
            Box.WBox(Label.widgetOf lbl1),
```

```
            Box.WBox(FocusFrame.widgetOf ff1)
        ])),
        Box.WBox (Box.widgetOf (Box.layout (root,view,[])
         (Box.HzCenter[
            Box.WBox(Label.widgetOf lbl2),
            Box.WBox(FocusFrame.widgetOf ff2)
        ])),
        Box.WBox (Box.widgetOf (Box.layout (root,view,[])
         (Box.HzCenter[
            Box.WBox(Label.widgetOf lbl3),
            Box.WBox(FocusFrame.widgetOf ff3)
        ]))
    ])
val shellArgs =
    [([], Attrs.attr_title, Attrs.AV_Str "eXene Resources Demo"),
     ([], Attrs.attr_iconName, Attrs.AV_Str "demo-res")]
val shell = Shell.shell (root, view, shellArgs) (Box.widgetOf layout)

val id1 = Shell.addFocusableFirst shell (FocusFrame.focusableOf ff1)
val id2 = Shell.addFocusableAfter shell (id1,FocusFrame.focusableOf ff2)
val id3 = Shell.addFocusableAfter shell (id2,FocusFrame.focusableOf ff3)

val hints = Shell.mkHints{size_hints=[],wm_hints=[ICCC.HINT_Input(true)]}
val _     = Shell.setWMHints shell hints
val cmEvt = Shell.deletionEvent shell

fun loop():unit =
    let
    fun handleQuit (Button.BtnUp _)   = (TextIO.print " [demo-res quitting]\n";
            quit())
      | handleQuit (_) = (loop())
    fun handleSum  (Button.BtnUp (b,t)) =
            let
            val r1 = (case (Real.fromString (SimpleEdit.getString smplEdit1)) of
                        SOME r => r | _ => 0.0)
            val r2 = (case (Real.fromString (SimpleEdit.getString smplEdit2)) of
                        SOME r => r | _ => 0.0)
            val rs = Real.toString (r1+r2)
            val _ = SimpleEdit.setString smplEdit3 rs
            in loop() end
      | handleSum  (_) = (loop())
    in CML.select
       [CML.wrap(quitEvt, handleQuit),
        CML.wrap(sumEvt,  handleSum),
        CML.wrap(cmEvt, quit)
      ]
```

```
        end
    in
        Shell.init shell;
        loop()
    end

  fun doit (dpyOpt,args) =
      (RunCML.doit (fn () => (init (dpyOpt,args)), NONE))

  fun main (prog, "-display"::(server::args)) =
            ((TextIO.print ("display="^server)); doit(SOME server,args))
    | main (prog, args) = doit(NONE,args)

end
```

Appendix E

Selections Demo source code

```
(*
 * Dusty deBoer, Kansas State University.
 *
 * Based on basicwin.sml, (C) 1990 J.H. Reppy; and goodbye.sml, (C) 1990 AT&T.
 *)

structure XDEMO : sig

    val doit  : string option * string list -> OS.Process.status
    val main  : (string * string list) -> OS.Process.status

  end = struct

    structure EXB = EXeneBase

fun init (dpyOpt,args) =
    let
    val root  = Widget.mkRoot(GetDpy.getDpy(dpyOpt))
        handle EXB.BadAddr s =>
            (TextIO.print s; RunCML.shutdown OS.Process.failure)

    val styleView = Styles.mkView{name = Styles.styleName["xdemo"],aliases = nil}
    val view      = (styleView, (Widget.styleOf root))

    fun quit ()  = (Widget.delRoot root; RunCML.shutdown OS.Process.success)

    val quitBttn = Button.textBtn (root, view,
            [([], Attrs.attr_label, Attrs.AV_Str "Handle Quit")])
    val quitEvt  = Button.evtOf quitBttn

    val getBttn = Button.textBtn (root, view,
```

```
        [([], Attrs.attr_label, Attrs.AV_Str "Get PRIMARY Selection")])
val getEvt  = Button.evtOf getBttn

val smplEdit1 = SimpleEdit.simpleEdit (root, view, []) ""
val smplEdit2 = SimpleEdit.simpleEdit (root, view, []) ""
val smplEdit3 = SimpleEdit.simpleEdit (root, view, []) ""
val smplEdit4 = SimpleEdit.simpleEdit (root, view, [])
        "Fourscore and seven years ago..."

val ff1 = FocusFrame.focusframe (root,view,[])
    ((SimpleEdit.widgetOf smplEdit1),(SimpleEdit.focusableOf smplEdit1))
val ff2 = FocusFrame.focusframe (root,view,[])
    ((SimpleEdit.widgetOf smplEdit2),(SimpleEdit.focusableOf smplEdit2))
val ff3 = FocusFrame.focusframe (root,view,[])
    ((SimpleEdit.widgetOf smplEdit3),(SimpleEdit.focusableOf smplEdit3))
val ff4 = FocusFrame.focusframe (root,view,[])
    ((SimpleEdit.widgetOf smplEdit4),(SimpleEdit.focusableOf smplEdit4))

val lbl1 = Label.label (root, view, [])
val _    = Label.setLabel lbl1 (Label.Text("Selection as STRING:    "))
val lbl2 = Label.label (root, view, [])
val _    = Label.setLabel lbl2 (Label.Text("Selection as TIMESTAMP: "))
val lbl3 = Label.label (root, view, [])
val _    = Label.setLabel lbl3 (Label.Text("Selection as LENGTH:    "))
val lbl4 = Label.label (root, view, [])
val _    = Label.setLabel lbl4 (Label.Text("Test text widget:       "))

val layout =
    Box.layout (root, view, []) (Box.VtCenter[
        Box.WBox(Button.widgetOf quitBttn),
        Box.WBox(Button.widgetOf getBttn),
        Box.WBox (Box.widgetOf (Box.layout (root,view,[])
         (Box.HzCenter[
            Box.WBox(Label.widgetOf lbl1),
            Box.WBox(FocusFrame.widgetOf ff1)
        ]))),
        Box.WBox (Box.widgetOf (Box.layout (root,view,[])
         (Box.HzCenter[
            Box.WBox(Label.widgetOf lbl2),
            Box.WBox(FocusFrame.widgetOf ff2)
        ]))),
        Box.WBox (Box.widgetOf (Box.layout (root,view,[])
         (Box.HzCenter[
            Box.WBox(Label.widgetOf lbl3),
            Box.WBox(FocusFrame.widgetOf ff3)
        ]))),
```

```
        Box.WBox (Box.widgetOf (Box.layout (root,view,[])
         (Box.HzCenter[
            Box.WBox(Label.widgetOf lbl4),
            Box.WBox(FocusFrame.widgetOf ff4)
        ])))
    ])
val shellArgs =
    [([], Attrs.attr_title, Attrs.AV_Str "X Resources"),
     ([], Attrs.attr_iconName, Attrs.AV_Str "xres")]
val shell = Shell.shell (root, view, shellArgs) (Box.widgetOf layout)

val id1 = Shell.addFocusableFirst shell (FocusFrame.focusableOf ff1)
val id2 = Shell.addFocusableAfter shell (id1,FocusFrame.focusableOf ff2)
val id3 = Shell.addFocusableAfter shell (id2,FocusFrame.focusableOf ff3)
val id4 = Shell.addFocusableAfter shell (id3,FocusFrame.focusableOf ff4)

val hints = Shell.mkHints{size_hints=[],wm_hints=[ICCC.HINT_Input(true)]}
val _     = Shell.setWMHints shell hints
val cmEvt = Shell.deletionEvent shell

fun loop():unit =
    let
    fun handleQuit (Button.BtnUp _)   = (TextIO.print " [xdemo quitting]\n";
            quit())
      | handleQuit (_) = (loop())
    fun handleGet  (Button.BtnUp (b,t)) =
            let
            fun rqs tgt = Shell.requestSelection shell (ICCC.Sel_PRIMARY,tgt,t)
            fun recv ((ICCC.Val_STRING s)) =
                    SimpleEdit.setString smplEdit1 s
              | recv ((ICCC.Val_COMPOUND_TEXT s)) =
                    SimpleEdit.setString smplEdit1 s
              | recv ((ICCC.Val_TIMESTAMP t)) =
                    SimpleEdit.setString smplEdit2
                        (Real.toString (EXB.XTime.toReal t))
              | recv ((ICCC.Val_LENGTH l)) =
                    SimpleEdit.setString smplEdit3 (Int.toString l)
              | recv _ = ()
            val _ = SimpleEdit.setString smplEdit1 ""
            val _ = SimpleEdit.setString smplEdit2 ""
            val _ = SimpleEdit.setString smplEdit3 ""
            val _ = case (rqs (ICCC.Tgt_MULTIPLE [ICCC.Tgt_STRING,
                        ICCC.Tgt_TIMESTAMP,ICCC.Tgt_LENGTH])) of
                    (SOME (ICCC.Val_MULTIPLE (mv))) => List.app recv mv
                  | _ => (case (rqs ICCC.Tgt_STRING) of
                             SOME (ICCC.Val_STRING s) =>
```

```
                                (SimpleEdit.setString smplEdit1 s)
                            | SOME (ICCC.Val_COMPOUND_TEXT s) =>
                                (SimpleEdit.setString smplEdit1 s)
                            | _ => ())
                in loop() end
          | handleGet  (_) = (loop())
        in CML.select
           [CML.wrap(quitEvt, handleQuit),
            CML.wrap(getEvt, handleGet),
            CML.wrap(cmEvt, quit)
          ]
        end
    in
        Shell.init shell;
        loop()
    end

  fun doit (dpyOpt,args) =
      (RunCML.doit (fn () => (init (dpyOpt,args)), NONE))

  fun main (prog, "-display"::(server::args)) =
           ((TextIO.print ("display="^server)); doit(SOME server,args))
    | main (prog, args) = doit(NONE,args)

end
```