The eXene Widgets Manual

(Version 0.4)

February 11, 1993

Emden R. Gansner
John H. Reppy

AT&T Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974

Contents

Chapter 1

Introduction

This document describes the **eXene** widget library; it is a companion to *The eXene Library Manual* [RG93], which is also included in the distribution.

The **eXene** widget library provides a collection of interface "widgets," plus the glue for connecting them into an application program. On the surface, the **eXene** widgets serve the analogous role to the base **eXene** library that the **Motif**, **OpenLook** and **Athena** toolkits play for **Xlib**. In **eXene**, however, the widget's internal structure and external connections are much more closely tied to the base library. Both the base library and the widget library were designed to reflect a common design philosophy for building user interfaces. Although the base library could be used to build libraries and applications in almost any style, it contains various pieces that strongly support a particular use of subwindows. The widget library uses these pieces to implement its components and to exemplify the underlying philosophy of constructing graphical interfaces.

The design promoted by **eXene** for widgets and their interactions with each other and with the application is based on a collection of related techniques.

- Most fundamental is concurrency. Highly-interactive graphical user interfaces are inherently concurrent. This concurrency should be made explicit and used. Allowing each widget its own thread separates it from other widgets and from the application code. This allows simpler structure inside the widget, with each widget synchronously reading its own input streams, and cleaner interfaces between widgets.

- Dealing with widget input can be further simplified by dividing "messages" to a widget into three functionally distinct streams, one each for the keyboard, mouse and control. The control stream provides such messages as "your window has been resized" and "redraw yourself". With this division, code for handling the keyboard or mouse can be written in a natural synchronous fashion, with no need to maintain state explicitly.

- Input is distributed hierarchically. Events are passed from the root widget down the hierarchy to the appropriate target widget. This allows the programmer to interpose widgets at any level to modify widget characteristics or alter the distribution of events.

- Few things are as full of state as graphical widgets. This state can best be controlled, especially in the context of a language such as **ML** that encourages immutable values, by encapsulating it within threads and channels. This avoids the need for the explicit classes, objects and inheritance that are usually

used in building user interfaces. Additional object-oriented techniques can be replaced with wrapper widgets and delegation.

The widget library is the most tentative part of the **eXene** system, and the most likely to undergo radical alterations in the near future. The current set of widgets is incomplete, and the individual widgets lack a finished look. The semantics of some widgets are not general enough, or are not what the user might expect. Some of the necessary protocols, and the underlying support, have not been completely designed or implemented. Most of the data structures, most notably the `Widget` structure itself, represent first passes at what will be necessary. Experience will require that some types be changed, that some fields be added, that some functions or types be judged wrong. There is, at present, no integration of **X** resources with the widget hierarchy.

Despite these shortcomings, the widget library provides a workable fabric for embroidering a collection of widgets into a user interface. It serves as an initial proof-of-concept for the **eXene** design philosophy, and points the direction for the construction of mature libraries based on the **eXene** design.

Chapter 2

Using Widgets

The widget library provides a higher-level platform for the construction of graphical interfaces than the basic **eXene** library. In particular, it emphasizes the use of widget components as the basic building block, blocks that can be used modularly in a wide variety of applications. Necessary alterations can be done externally, through resources, parameter settings, or by wrapping the widget in other widgets.

In order for widgets to work together at this level, a certain uniformity must be assumed. This uniformity is achieved by requiring widgets to provide a certain external interface and to respect certain internal protocols, and requiring the programmer to obey a few additional constraints on the construction and use of the widget hierarchy. The basic types and values for working with **eXene** widgets are given in the `Widget` structure, which matches the `WIDGET` signature. In this chapter, we describe the pieces of this structure in detail as they relate to tailoring new widgets from old and combining them into applications. A discussion of the implementation of a new widget from scratch is left for Chapter 5.

2.1 Widget geometry

In order to specify widget layouts, it is necessary to describe various geometric constraints on widgets, such as possible sizes and alignments. **EXene** provides a simple but fairly general way of modeling these constraints. The assortment of types and values employed for specifying these constraints are given below.

```
signature WIDGET =
  sig

    structure CML : CONCUR_ML
    structure G : GEOMETRY
    structure EXB : EXENE_BASE
    structure Interact : INTERACT
    structure EXW : EXENE_WIN

    datatype valign = VCenter | VTop | VBottom
    datatype halign = HCenter | HRight | HLeft

    exception BadIncrement

    datatype dim = DIM of {
        base    : int,
        incr    : int,
        min     : int,
        nat     : int,
        max     : int option
      }

    (* type bounds = { x_dim : dim, y_dim : dim } *)
    type bounds
    val mkBounds : { x_dim : dim, y_dim : dim } -> bounds

    val fixDim : int -> dim
    val natDim : dim -> int
    val minDim : dim -> int
    val maxDim : dim -> int option
    val fixBounds : (int * int) -> bounds
    val compatibleDim : (dim * int) -> bool
    val compatibleSize : (bounds * G.size) -> bool

    ...
  end
```

The `valign` and `halign` datatypes provide values for specifying vertical and horizontal alignment, respectively. For example, to specify that a collection of items should be left-justified, the programmer chooses the value `HLeft`.

The general size requirements of a widget are given by a value of type `bounds`. This record has fields `x_dim` and `y_dim` for specifying requirements in the horizontal and vertical directions, respectively. The `dim` datatype represents the size constraints of the widget in a given dimension. These sizes are expressed as a base size plus some multiple of an increment. Specifically, the minimum size, in pixels, is given by `base + min*incr`; the natural size is `base + nat*incr`; if `max = SOME mx`, the maximum size is `base + mx*incr`, otherwise, the dimension is unbounded. For example, a virtual terminal might use an `x_dim` of

```
DIM{base = pad, incr = fontw, min = 1, nat = 24, max = NONE}
```

where `pad` is some base padding and `fontw` is the width of a character. This gives a minimum width of one character, a natural width of 24 characters, with an unbounded maximum width.

The integer values given in a `dim` value should satisfy the following constraints:

- `base >= 0`
- `incr >= 1`
- `base + min*incr >= 1`
- `0 <= min <= nat <= max`.

Note that these constraints are not in general enforced by **eXene**, but are the responsibility of the programmer. In certain cases, the exception `BadIncrement` will be raised if the second constraint is violated. The third constraint is only necessary because **X** does not allow windows of zero width or height. It may be possible relax this constraint in **eXene**.

The `Widget` structure provides some auxiliary functions for dealing with `dim` and `bounds` values. The `mkBounds` function is the identity function. It provides a constructor function for `bounds` values needed in functors due to the absence of type abbreviations in signatures. The `minDim`, `natDim` and `maxDim` functions return the respective sizes in pixels of the given dimension. `fixDim` and `fixBounds` return rigid (i.e., `min = nat = max`) dimensions and bounds, respectively, corresponding to given parameters. The `compatibleDim` function returns true if the integer argument is at least the minimum size and no more than the maximum size specified by the `dim` argument. The `compatibleSize` function returns true if the size is compatible with the `bounds` argument in both dimensions.

2.2 Roots

The `root` type serves as an abstraction for a single screen. It can be viewed as representing the root window of the screen. In the future, the root structure may also carry various application specific information.

```
signature WIDGET =
  sig

    ...
    type root

    val mkRoot : string -> root
    val delRoot : root -> unit
    val sameRoot : (root * root) -> bool
    val displayOf : root -> display
    val screenOf : root -> screen

    ...
  end
```

The `mkRoot` function accepts a string argument specifying the server connection (see Chapter 3 of *The eXene Library Manual* for more information). The `mkRoot` function opens a connection to the specified **X** server, and returns a `root` value corresponding to the default screen of the server.

A `root` value is needed to create most widgets, and therefore `mkRoot` is usually invoked early in an application. Widgets use the root to get access to the display, and thus to various resources such as fonts, colors and pixmaps, all which require a display argument.

The `delRoot` function closes the display connection. This has the side effect of deleting all the windows associated with that root, plus freeing all server resources (e.g., fonts, colors, etc.) associated with the root. The `screenOf`, `displayOf` and `sameRoot` functions do the obvious.

2.3 Widgets

A widget is essentially a triple consisting of a `root` value, a `boundsOf` function, and a `realize` function. It is implemented as an abstract type through the `mkWidget` function.

```
signature WIDGET =
  sig

    ...
    type widget
    datatype wstate = Active of bool | Inactive of bool

    exception AlreadyRealized

    (* type realize_fn = {
     *      env : in_env,
     *      win : window,
     *      sz : size
     *   } -> unit
     *)
    type realize_fn

    val mkWidget : {
        root : root,
        boundsOf : unit -> bounds,
        realize : realize_fn
      } -> widget
    val rootOf : widget -> root
    val boundsOf : widget -> bounds
    val realizeFn : widget -> realize_fn
    val sameWidget : (widget * widget) -> bool
    val natSize : widget -> size
    val boundsFn : widget -> unit -> bounds
    val okaySize : (widget * size) -> bool

    ...
  end
```

The `root` value simply specifies the root to which the widget belongs. A widget's window and its external characteristics (e.g., size, origin, visibility) are controlled by the widget's parent. The parent creates the window and passes it to the child through the child's `realize` function. A widget can only be realized once; calling its `realize` function additional times will raise the `AlreadyRealized` exception. The child also receives the size of the window and its environment. The environment contains three event values used to transmit keyboard, mouse and control events to the child. This is input environment discussed in Chapter 7 of *The eXene Library Manual*. It is the child's responsibility to always be available to synchronize on these events. The `realize` function will be more fully discussed in terms of widget internals (see Chapter 5).

The `boundsOf` function is usually used by a widget's parent to determine the size constraints of the widget

when positioning the child within the parent's window. It should be noted that what bounds a widget returns through its `boundsOf` function is only a suggestion to the parent. Parent widgets should try to accommodate a child's size requests, but sometimes this is not possible. Widgets, in turn, should be designed to safely accommodate any size. Also note that the `boundsOf` function should not block; each widget should construct its `boundsOf` function to return as quickly as possible.

The `Widget` structure contains additional functions associated with widget properties. The `rootOf`, `boundsFn` and `realizeFn` functions act as selector functions for these defining attributes of a widget. The `boundsOf` function returns the current bounds desired by the widget. The function `natSize` returns the current natural size associated with the widget. The function `okaySize` returns true if the given size is compatible (see `compatibleSize` above) with the widgets bounds.

Because of the state implicitly associated with widgets, each widget receives a unique stamp in `mkWidget`. The `sameWidget` function reports on the equality of these stamps.

The `wstate` type is used various places in the widget library to specify widget state. Widgets associated with a `wstate` value can be considered either active or inactive. A widget in either state can be either set or unset. Thus, an active widget that is set would correspond to `Active true`. Usually, these states are reflected in the display of the widget. In addition, the user is usually prohibited from changing the set/unset state of an inactive widget.

Widgets live fairly circumscribed lives. They are created; they are inserted into a parent; they are realized, being given an input environment and a window; they might be removed from a parent widget; their environment and window are destroyed. As noted above, the program cannot realize a widget twice. Nor are there any provisions for removing a widget from the widget hierarchy, and then reinserting it later, even back into its original parent.

2.4 Programming with Widgets

There are a few rules concerning the use of widgets in building a program. At some point in the program, a `root` value must be created. This can be used to create widgets in varying order. Many composite widgets require a list of child widgets at the time of creation; this necessitates that a child widget be created before its parent. Certain composite widgets allow child widgets to be added dynamically. Even in these cases, it is a good practice and more efficient to provide the child widgets at the time the parent is created. At some point, a widget tree must be attached to a `shell` widget. To actually instantiate the widget tree (i.e., to realize the widgets and create and map the underlying windows), the `init` function must be called on the shell widget. In most **X** toolkits, the call to instantiate the widgets causes the application program to give up control to the event-driven control loop within the toolkit library. Alternatively, the application must be willing to provide its own event distribution mechanism. Neither of these deficiencies is true in **eXene**. After instantiating the widgets, the main application thread is still active and can continue to go about its business, whether performing computations, reading from standard input or interacting with the widgets.

Figure 2.1 contains the code for a simple program which uses widgets. This example draws a framed button labeled "`Goodbye, Cruel World!`". If the user clicks on the button with any mouse button, then the function `quit` is called. The program will also quit if the user enters "`quit\n`" on standard input. This is similar to the `xgoodbye` example in Chapter 2 of [NO90], but differs in one major way. In the `xgoodbye`,

```
open Widget Box FramedButton
fun goodbye server = let
      val root = mkRoot server
      fun quit () = (delRoot root; RunCML.shutdown())
      val layout = Box.widgetOf (mkLayout root (VtCenter [
            Glue {nat=30, min=0, max=NONE},
            WBox (widgetOf (mkFrTextCmd root {
              action = quit,
              foregrnd = SOME(EXB.blackOfScr(screenOf root)),
              backgrnd = NONE,
              label="Goodbye, Cruel World!",
              border_width = 1
            })),
            Glue {nat=30, min=0, max=NONE}
          ]))
      val shell = Shell.mkShell (layout, NONE, win_name = NONE, icon_name = NONE)
      fun loop () =
            if (CIO.input_line CIO.std_in) = "quit\n"
              then quit ()
              else loop ()
      in
        Shell.init shell;
        loop ()
      end
```

Figure 2.1: Goodbye

control is passed off to the **Xt** event loop; in our version, the application retains control (reading the standard input in this case).

There are many ways new widgets can be tailored from existing ones. In addition to the obvious technique of altering the parameters and values of a widget, the **eXene** widget library provides the ability to compose widgets graphically and functionally.

Graphical composition occurs when the window of a widget is included within the window of a parent widget. A trivial example of this technique occurred in the example above, in which a button was composed within a vertical box. Below, in Figure 2.2, we give a more substantial example, in which a label widget and a slider widget are combined to provide textual feedback for the slider's value.

The second technique for composing widgets occurs when a widget wraps another in order to interpose its own behavior. In the following example, the wrapping widget creates a rigid widget by interposing its own `boundsOf` function.

```
fun mkRigid w = let
      fun myBounds () = let
            val {x_dim,y_dim} = boundsOf w
            in
              {x_dim = fix x_dim, y_dim = fix y_dim}
            end
      in
        mkWidget{root=rootOf w, boundsOf=myBounds, realize=realizeFn w}
      end
```

```
fun mkLabelSlide root (wid, color) = let
      val label = Label.mkLabel root {
        label="  0",
        foregrnd=color,
        backgrnd=NONE,
        font=NONE,
        align=HRight
      }
      val slider = Slider.mkHSlider root {
        foregrnd=color,
        wid=wid,
        init=0,
        scale=100
      }
      val set = Label.setLabel label
      val evt = Slider.evtOf slider
      fun loop () = loop (set (makestring (sync evt)))
      in
        spawn loop;
        mkLayout root (HzCenter [
            WBox (Label.widgetOf label),
            Glue nat=20, min=20, max=SOME 20,
            WBox (Slider.widgetOf slider)
          ])
      end
```

Figure 2.2: Composing label and slider widgets

As another example of wrapping a widget to alter its behavior, Figure 2.3 shows how we can add a pop-up menu to the example of Figure 2.1. In this example, we have taken the code from the previous example, defined a menu, and used the `attachMenu` function to wrap a handler for this menu around the button widget. This produces a new widget and a CML event. The new widget works identically to the old widget, except when the user presses the third button in the window. When this happens, a pop-up version of the menu appears. If the user makes a selection, this selection is reported through the event value. The main `loop` function now services both menu selection as well as keyboard input.

The `Widget` structure provides several high-level functions that support modifying widgets through wrappers. These are discussed in Section 5.4.

```
val menu = MENU [
        MenuItem("item-1", 1),
        MenuItem("item-2", 2),
        MenuItem("item-3", 3),
        Submenu("submenu1", MENU [
            MenuItem("item-4", 4),
            MenuItem("item-5", 5),
            MenuItem("item-6", 6)
          ]),
        MenuItem("item-7", 7)
      ]
fun goodbye server = let
      val root = mkRoot server
      fun quit () = (delRoot root; RunCML.shutdown())
      val layout = Box.widgetOf (mkLayout root (VtCenter [
            Glue {nat=30, min=0, max=NONE},
            WBox (widgetOf (mkFrTextCmd root {
              action = quit,
              foregrnd = SOME(EXB.blackOfScr(screenOf root)),
              backgrnd = NONE,
              label="Goodbye, Cruel World!",
              border_width = 1
            })),
            Glue {nat=30, min=0, max=NONE}
          ]))
      val (widget, evt) = attachMenu (layout, [Interact.MButton 3], menu)
      val shell = mkShell (widget, NONE, win_name = NONE, icon_name = NONE)
      fun loop () =
            select[
              wrap(evt,
                fn n => (CIO.print("choice = "^ makestring n ^ "\n"); loop())),
              wrap(CIO.inputLineEvt CIO.std_in,
                fn line => if line = "quit\n" then quit () else loop ())
            ]
      in
        init shell;
        loop ()
      end
```

Figure 2.3: Goodbye with a pop-up menu

Chapter 3

Widgets

We note that widgets divide naturally into two classes: composite and simple. The former are those that take other widgets as parameters. They graphically contain one or more child widgets, and are responsible for the layout of the child widgets. Simple widgets usually occur as the leaf nodes in the widget hierarchy and provide the basic pseudo-devices (scrollbars, meters, dials, buttons, etc.) with which the user does input. These definitions are not meant to be precise. In particular, we note that some simple widgets are in fact implemented as composite widgets.

All widgets act like subtypes of the `widget` type. We use the convention that each widget type supplies an explicit type casting function `widgetOf` that exposes an underlying `widget` value.

In the following sections, we describe the currently available widgets. For each widget, we comment on what it does, how it reacts to some of the important base protocols, how it is created, and how it is parameterized. Note that there is little consistency as to which parameters, such as color, font or sizes, are available to the programmer dynamically and which are built into the widget. We assume that this problem will go away in a future release when a uniform resource mechanism is supplied.

3.1 Shell

The `shell` type does not define a widget per se. It provides an abstraction for top-level windows, which obviously cannot be inserted in other widgets, and handles the mismatch between the **X** model and the **eXene** model. It allows the user to provide top-level window information to the window manager. It also hides various other implementation differences between top-level windows and subwindows.

```
signature SHELL =
  sig
    structure W : WIDGET

    type shell

    type wm_args (* = { win_name : string option, icon_name : string option } *)
    val mkWMArgs : win_name : string option, icon_name : string option -> wm_args

    type hints (* = { size_hints : size_hints list, wm_hints : wm_hints list } *)
    val mkHints : {
            size_hints : W.EXW.ICCC.size_hints list,
            wm_hints : W.EXW.ICCC.wm_hints list
          } -> hints

    val mkShell : (widget * W.EXB.color option * wm_args) -> shell
    val mkTransientShell : W.EXB.window ->
          (widget * W.EXB.color option * wm_args) -> shell

    val init : shell -> unit
    val destroy : shell -> unit
    val setWMHints : shell -> hints -> unit
  end
```

The `mkShell` function creates a shell using a child widget, an optional color, and optional labels for the top-level window and its icon. The color specifies the top-level background color; by default, it is white. The labels are passed to the window manager. They correspond to the `win_name` and `icon_name` parameters used with `setWMProperties`, as described in Chapter 4 of *The eXene Library Manual*. The shell will be totally overlaid by its child widget; the bounds of the shell are the same as its child.

The widget tree attached to the shell is not instantiated until the program calls the `init` function on the shell. Only at this point will the underlying windows be created and mapped, and the widget hierarchy will become visible. The natural size of the top-level window is the natural size of the child widget. The shell uses the bounds of the top-level window to specify a base, minimum and maximum window size plus window size increments to the window manager. A call to `destroy` on a shell causes the destruction of all the windows in the widget hierarchy below the shell.

The `mkTransientShell` function allows you to create a "transient" top-level window. **X** provides three flavors of top-level windows. Most are meant to be "long-lived" and managed by the window manager. Typically, the window manager will decorate them with a frame, a title bar and various icons. These are the windows created by the `mkShell` function. At the other extreme are pop-up windows. These are meant to last very short periods of time, as in a pop-up menu, and are not registered with the window manager. At present, the widget library does not provide direct access to such windows, although they are used in the implementation of menus. The programmer can create them using the `createSimplePopupWin` function in the base library. The third type of window is transient, occupying a middle ground between the other two. A typical use of a transient window would be in a dialogue box. The window manager knows about the transient window, but usually does not give it a title bar or other decoration. A transient window is associated with another window (the first argument to the `mkTransientShell` function). Typically, the window manager will iconify/de-iconify a transient window in conjunction with its associated window.

The `setWMHints` function allows the application to specify hints about the top-level window and the

application to the display manager. In particular, the programmer can use `setWMHints` to override the default size hints that the shell passes to the window manager. If `setWMHints` is called before the `init` function, the shell queues the hints, which will be sent when `init` is called. A more complete description of the hints can be found in *The eXene Library Manual* or **X** manuals (e.g., [Nye90]).

A shell services its child's control channel. Requests from the child widget for resizing are respected; the shell will attempt to change the size of the top-level window. Receiving a `CO_KillReq` from the child is the same as calling the `destroy` function.

The `mkWMArgs` and `mkHints` functions are identity functions on the respective records.

3.2 Composite Widgets

Composite widgets are used to manage the layout of child widgets within a given window. A composite widget plays the same role for its window as the window manager plays for the screen. Because of their dual role as a child widget as well as a parent to various child widgets, composite widgets have more to do than simple widgets in maintaining the protocols on the control events. Graphically, they usually have little or nothing to do.

3.2.1 Background

The `background` widget is used to provide a default background color to a widget and its descendants. The **X** graphic model associates a background color with a window. It is this color that is used in the graphics operations `clearArea` and `clearDrawable` (cf. Chapter 5 in *The eXene Library Manual*). By default, windows in **eXene** inherit their parent's background color, which is set when the `shell` is created (cf. Section 3.1).

```
signature BACKGROUND =
  sig

    type background

    val mkBackground : { color : color option, widget : widget } -> background

    val widgetOf : background -> widget

  end
```

The `mkBackground` command creates a new widget with a new associated background color. The argument widget and all its descendents will inherit the new color, until a `background` is created lower in the hierarchy. If no color is specified, white is used.

3.2.2 Box

Box widgets provide for the non-overlapping placement of child widgets along a horizontal or vertical axis. A child widget is expanded or contracted to fill as much of its innermost containing box as possible, but the size bounds of a child are never violated.

```
signature BOX =
  sig

    exception BadIndex

    datatype box
      = HzTop of box list
      | HzCenter of box list
      | HzBottom of box list
      | VtLeft of box list
      | VtCenter of box list
      | VtRight of box list
      | Glue of { nat : int, min : int, max : int option }
      | WBox of widget

    type box_layout

    val mkLayout : root -> box -> box_layout
    val widgetOf : box_layout -> widget
    val insert : box_layout -> (int * box list) -> unit
    val append : box_layout -> (int * box list) -> unit
    val delete : box_layout -> int list -> unit
    val mapBox : box_layout -> int list -> unit
    val unmapBox : box_layout -> int list -> unit

  end
```

The `mkLayout` function requires a display root and a `box` value. The layout of the widgets is recursive. The boxes in a list are laid out from left to right in horizontal boxes, and from top to bottom in vertical boxes. `Glue` boxes act like transparent widgets and are used to provide spacing between other boxes.

The layout algorithm is simple. We describe the case for a horizontal box. Vertical boxes work the same way, switching the roles of horizontal and vertical. Each child is given its natural width. If the sum of these widths does not fill the width of the box, the slack is allocated uniformly to the child widgets, but only in multiples of a child's increment value and a child's maximum width is never exceeded. If there is still slack after all children have been increased to the maximum widths, it is placed to the right of all the children. If the sum of the widths is too large for the box, the excess is removed uniformly from the child widgets, but a child's minimum width and increment value are respected. If there is still excess after all children have been decreased to the minimum widths, some of the rightmost children will not appear in the window.

Each child is guaranteed its minimum height. If this does not equal the height of the box, the child's height is increased as much as possible, in multiples of the child's increment, up to its maximum height. If this is still not equal to the height of the box, the child is aligned vertically according to the box's alignment parameter. Thus, a `HzCenter` box will center its components vertically, while a `HzTop` box will top justify its components. If a child's height is too large for the box, the child is still aligned vertically according to the box's alignment parameter, but part of the child will not be visible.

We now describe the bounds of a box. It seems reasonable that the minimum, natural and maximum widths of a horizontal box should be the sums of the respective widths of its children. To this end, we set the base width `base` to be the sum of the minimum widths of its children, in pixels. The horizontal increment `incr` is the minimum of the horizontal increments of all children with a non-fixed horizontal size. The `min` value is set to zero. The `nat` value is the least integer such that `base + nat*incr` is greater than or equal

to the sum of the natural widths of its children. The `max` value is defined analogously. The natural height of the box is the maximum of the natural heights of its children. The minimum height of a box is the maximum of the minimum heights of its children. The maximum height of a box is the maximum of the natural height of the box along with the non-infinite maximum heights of its children, or infinite if all children have infinite height. We then set the base height `base` to the minimum height and `min` to 0. The vertical increment is the minimum of the vertical increments of all children with non-fixed vertical size and whose vertical increment is greater that 1. If there are no such children, the vertical increment is set to 1. As in the horizontal direction, `nat` and `max` are taken to be the smallest integers such that `base + nat*incr` and `base + max*incr` are greater than or equal to the natural height and the maximum height, respectively. In horizontal boxes, glue components act like widgets whose horizontal bounds are given by the glue's parameters, with an implicit `base` of 0 and `incr` of 1, and whose vertical bounds have a natural size of zero, with infinite shrinking and stretching.

The rules given above for determining the bounds of a box are obviously heuristics. They should work well in a given dimension when the sub-boxes have a fixed size in that dimension, have an increment of one, or have "compatible" sizes (e.g., the same natural size with increments that are multiples of some base increment). These conditions hold true in such common cases as attaching a scrollbar or using sufficient glue. When these conditions are not satisfied, the resultant bounds of a box can be unexpected.

If the box argument *b* to `mkLayout` is a `Glue` or `WBox` value, it is treated as `HzCenter [`*b*`]`. The `Widget.BadIncrement` exception is raised if a widget has a zero increment.

The box tree managed by a box layout widget can be dynamically altered using functions provided in the `Box` structure. At present, these changes can only be made in the top-level list in the box tree. (This is not a serious restriction, as a `box_layout` can be inserted within another `box_layout`.) A call of `insert` *layout* `(`*n*`,`*bl*`)` inserts the list of boxes *bl* before the *n*th box in the top-level box list. Indexing of boxes starts at 0; *n* can be any integer up to and including the length of the top-level box list, the maximum value corresponding to appending *bl* to the list. Any other value of *n* causes the exception `BadIndex` to be raised. When a collection of boxes in inserted into a box layout, the layout is recomputed using the geometry of the new boxes. The widgets in the new boxes are assumed to be unrealized; they will be realized at this time. `append` *layout* `(`*n*`,`*bl*`)` is equivalent to `insert` *layout* `(`*n*`+1,`*bl*`)`.

`delete` *layout il* removes the boxes whose indices are given in the index list *il*. This destroys any windows associated with widgets in the boxes, and effectively destroys the widgets. The `unmapBox` tells the layout widget to pretend that the boxes whose indices are given in the argument list have zero size and to reposition its children accordingly. Unmapping an already unmapped box has no effect. The `mapBox` function allows unmapped boxes to be made visible again. Mapping an already mapped box has no effect. All boxes are assumed to be in a mapped state when inserted. In all these three functions, an invalid index will cause the exception `BadIndex` to be raised.

3.2.3 Frame

The ability to specify a border of an **X** window is a programming convenience in simple applications, and improves performance by allowing the server to perform damage control on borders which would otherwise have to be handled by the application. On the negative side, the use of borders unnecessarily complicates the computation of window geometries. Knowing the upper left corner of a window and the size of its drawing

region is not enough to calculate the actual bounding box of the window; the window's border must be taken into account.

For these reasons, the **eXene** widget library assumes all underlying **X** windows will have borders of width zero. To provide borders, the library supplies frame widgets. A frame widget creates a border of a given size and color around its child widget.

```
signature FRAME =
  sig
    val mkFrame : {
            color : color option,
            width : int,
            widget : widget
          } -> widget

    val widgetOf : frame -> widget
    val setColor : frame -> color option -> unit

  end
```

The `mkFrame` function takes an optional color (the parent's background if `NONE`), a border width, and the child widget about which the border is wrapped. A negative border width causes the `LibBase.BadArg` exception to be raised. The bounds of a frame are the same as those of its child except that the base is increased to allow for the border. Except for the actual border, a frame widget is transparent, inheriting its parent's background. The `setColor` function can be used to change the frame's color dynamically.

3.2.4 Pile

Where a `box_layout` widget composes its children in non-overlapping rows and columns, a `pile` widget maintains a list of children, only one of which is visible at a time.

```
signature PILE =
  sig

    type pile

    exception NoWidgets
    exception BadIndex

    val mkPile : root -> widget list -> pile
    val widgetOf : pile -> widget

    val mkVisible : pile -> int -> unit
    val visible : pile -> int
    val size : pile -> int

    val insert : pile -> (int * widget list) -> unit
    val append : pile -> (int * widget list) -> unit
    val delete : pile -> int list -> unit

  end
```

The `mkPile` function takes a root and a list of widgets and creates a `pile` widget. In a `pile`, only one child widget is visible at a time. The bounds and resize behavior of a `pile` widget are the same as its currently visible child. The child widgets are indexed consecutively from zero. Initially child 0 is visible. `mkVisible` causes the child with the given index to become the new visible widget. This raises the exception `BadIndex` if the index is invalid. Note that, in this model, the currently visible child is not placed at the front of the list of children: it is only marked as the visible child. As long as no widgets are added or deleted to a pile, the indexing of the children remains fixed. The query functions `visible` and `size` return the index of the currently visible child and the number of children, respectively.

A `pile` allows widgets to be dynamically inserted and deleted with the `insert`, `append` and `delete` functions. Inserting or appending widgets does not alter the currently visible widget. It does change the indices associated with the widgets, so it is possible the currently visible widget will have a new index. Similary, deleting widgets does not necessarily alter the currently visible widget, though it does cause a reindexing. However, if the visible widget is one of those removed from the `pile`, widget 0 of the remaining widgets becomes the new visible widget. Additional semantics associated with these functions are analogous to the similar functions in `Box` (cf. Section 3.2.2).

An empty pile acts like a transparent widget whose bounds are fixed at one pixel by one pixel. Invoking `visible` on an empty pile raises the `NoWidgets` exception.

3.2.5 Scrollbar layout

The `ScrollLayout` structure implements a utility function for combining widgets into a typical "widget with scrollbar" layout. This layout consists of the main widget, with an auxiliary widget placed to the left or right of the main widget, and another auxiliary widget place below or above the main widget. The resulting widget is a straightforward application of `mkLayout`.

```
signature SCROLL_LAYOUT =
  sig

    structure Box : BOX

    val mkSBLayout : root ->
          widget : widget,
          hsb : {sb : widget, pad : int, top : bool} option,
          vsb : {sb : widget, pad : int, left : bool} option
         -> Box.box_layout

  end
```

The `mkSBLayout` function takes a root, plus the main and auxiliary widgets (plus some formatting information) and produces a `box_layout` widget. If `hsb` is not `NONE`, the associated widget will be centered above (below) the main widget if `top` is true (false). The `pad` parameter specifies how much space, in pixels, to leave between the main and auxiliary widget. The `vsb` field is handled analogously.

The bounds functions of the auxiliary widgets are not modified. If the programmer desires that they always occupy an entire side of the main widget, they must have bounds that are flexible in the appropriate dimension. The `sb` fields purposely take `widget` values rather than `scrollbar` values. This makes the function useful in more general cases. Even when an auxiliary widget is basically a scrollbar, it might still be

framed or composed with arrow buttons and therefore would not be usable as a `scrollbar` in the composite widget.

3.2.6 Scrollport

The `ScrollPort` structure is a first attempt at automatically attaching panning features to a widget. The programmer supplies a widget, and the scroll port attaches scrollbars. The user can use the scrollbars to pan over the underlying widget.

```
signature SCROLL_PORT =
  sig

    type scroll_port

    val mkScrollPort : {
            widget : widget,
            continuous : bool,
            color : color option,
            hsb : {top : bool} option,
            vsb : {left : bool} option
          } -> scroll_port

    val widgetOf : scroll_port -> widget

  end
```

The `widget` argument to `mkScrollPort` specifies the underlying widget whose view is to be controlled by the scroll port. The `hsb` and `vsb` arguments specify whether horizontal and vertical scrollbars are to be attached, and to which side they should be attached. The `color` argument is passed to the scrollbar creation functions (cf. Section 3.3.7). If `continuous` is true, the scroll port will attempt to keep the widget view continuously in step with the scrollbar position. Otherwise, it will wait until the user releases the mouse, fixing a final position, before it updates the widget view. The latter choice is recommended if it is expensive for a widget to update its display.

3.2.7 Shapes

The widgets in the `Shape` structure allow the programmer to control the bounds requirements of a given widget. The `mkShape` function takes a widget whose bounds are to be constrained and two constraint functions. It creates a new widget wrapping the given widget. The new widget is identical to the given widget except for two aspects. The new widget's `boundsOf` function returns the value given by the `bounds_fn` when called with the child's `boundsOf` function. In addition, whenever the child asks to be resized, the resize request is only passed on to the parent if the `resize_fn`, invoked with the child's `boundsOf` function, returns true.

```
signature SHAPE =
  sig

    val mkShape : {
      widget : widget,
      bounds_fn : ((unit -> bounds) -> bounds),
      resize_fn : ((unit -> bounds) -> bool)
    } -> widget

    val mkRigid : widget -> widget
    val mkFlex : widget -> widget
    val fixSize : (widget * size) -> widget
    val freeSize : (widget * size) -> widget

  end
```

The remaining functions are a special cases of `mkShape`. The `fixSize` function creates a widget whose bounds will always be fixed at the given size, and which will never request to be resized. The `freeSize` function creates a widget whose natural size will always be the given size, but is totally flexible in both dimensions. The widget will always pass on resize requests from its child. The `mkRigid` and `mkFlex` functions apply the `fixSize` and `freeSize` functions, respectively, using the natural size of the argument widget.

3.2.8 Viewport

The `Viewport` structure is an initial primitive attempt at providing the basis for general-purpose panning over widget. A viewport object provides a classical window on the virtual graphical space of an underlying window. In effect, the underlying widget can be arbitrarily large, but only the part of it that is projected through the viewport's window is visible. The amount of the underlying window that can be seen depends on the size of the viewport window. In addition, the viewport's position relative to the underlying widget can be changed, providing panning. A viewport is usually tied to other widgets such as scrollbars to give the user control over the panning (cf. Section 3.2.6).

```
signature VIEWPORT =
  sig

    type viewport

    val mkViewport : widget -> viewport

    val widgetOf : viewport -> widget
    val getGeometry : viewport -> {rect : rect, childSz : size}
    val setOrig : viewport -> point -> unit
    val setHorz : viewport -> int -> unit
    val setVert : viewport -> int -> unit
    val evtOf : viewport -> {rect : rect, childSz : size} event

  end
```

A `viewport` is obtained by applying `mkViewport` to a widget. The bounds of the viewport will have the same natural size as the underlying widget, but can be arbitrarily shrunk or grown. If the underlying widget

is potentially large, it is therefore usually a good idea to wrap a viewport in another widget to constrain its size. At realization time, the window of the underlying widget is made a subwindow of the viewport window. Regardless of the size of the viewport's window, the subwindow is precisely the size requested by the child widget.

In the current model, the viewport cannot extend beyond the boundaries of the underlying widget. In particular, it can be no larger than the widget's window. The viewport determines a rectangle in the underlying widgets coordinate system.

There are various functions for monitoring and controlling the viewport's position. The function `getGeometry` returns the underlying widget's current size, and the viewport's rectangle in the child widget's coordinates. The `setOrig` function can be used to move the viewport's origin in the widget's coordinate system. This raises the exception `LibBase.BadArg` if the new rectangle is illegal. The x and y coordinates can be set independently using `setHorz` and `setVert`. These functions involve the same exception as `setOrig`. Any changes to the viewport configuration are reported through the event supplied by `evtOf`. Monitoring this event allows these changes to be reflected in any associated widgets such as scrollbars.

3.2.9 Widget set

Strictly speaking, a `widget_set` is not a widget. It provides a mechanism for managing a collection of widgets, some of which can be selected, either by the user or under program control. A typical use would involve a collection of toggle buttons, each setting a piece of shared program state to some value. Using a `widget_set`, the user could change the state by clicking on one of the buttons; the button indicating the previous state setting would automatically be unset.

```
signature WIDGET_SET =
  sig

    exception BadIndex
    exception MultipleChoices

    type widget_set

      (* type set_item = {
       *   widget : widget,
       *   state : wstate,
       *   pick_fn : bool -> unit,
       *   active_fn : bool -> unit
       * }
       *)
    type set_item
    val mkSetItem : {
           widget : widget,
           state : wstate,
           pick_fn : bool -> unit,
           active_fn : bool -> unit
         } -> set_item

    val mkMultiSet : root -> set_item list -> (widget_set * widget list)
    val mkSingleSet : root -> set_item list -> (widget_set * widget list)

    val setChosen : widget_set -> (int * bool) list -> unit
    val setActive : widget_set -> (int * bool) list -> unit
    val getChosen : widget_set -> int list
    val getState : widget_set -> Widget.wstate list

    val insert : widget_set -> (int * set_item list) -> widget list
    val append : widget_set -> (int * set_item list) -> widget list

  end
```

The `mkMultiSet` and `mkSingleSet` functions are used to create a widget set. The principal difference is that the former allows multiple items to be set, whereas the latter allows at most one item to be set. These functions take a `root` and a list of items to be included in the set. They return a list of wrapped widgets, and a `widget_set` value for controlling the set. The `widget_set` does no placement and is not a composite widget. The return widgets must still be inserted into the widget hierarchy through some composite widget.

Each `set_item` specifies an underlying widget, the initial state of the widget, a function to be called when the widget changes between set and unset, and a function to be called when a transition is made between active and inactive. These functions are only called due to subsequent transitions in the `widget_set`; they are not applied to the initial widget state.

It is assumed that widgets are chosen that graphically reflect the current state. If the user clicks mouse button 1 on an active widget, the widget is put in the set state. If it is already in the set state, nothing happens. Otherwise, if it is in a singleton set, the `pick_fn` of any currently set item is called with false. Then the `pick_fn` of the new item is called with true. If the user clicks mouse button 2 on an active widget, the widget is put in the unset state. If it is already in the unset state, nothing happens. Otherwise, its `pick_fn` is called with false. User clicks on an inactive item have no effect. All input environment events, including mouse clicks, are passed to the underlying widget after processing by the widget set.

The states of the widgets are under program control. The `setChosen` function can be used to set the items whose indices are given to the specified state. Note that `setChosen` affects inactive as well as active items. The `getChosen` function returns a list of the indices of currently selected items. The `setActive` function can be used to alter which items are currently active. The `getState` function returns a list of the current widget states in order. Invalid indices with `setChosen` and `setActive` cause the `BadIndex` exception to be raised.

The composition of a `widget_set` can be changed dynamically, using the `insert` and `append` functions. Semantics associated with indexing are the same as with the analogous functions in `Box` (cf. Section 3.2.2). The returned list widgets must be inserted into some composite widget to be realized.

If multiple widgets in a set state are passed to `mkSingleSet`, or if `insert` or `append` cause multiple items in a singleton set to be in a set state, the exception `MultipleChoices` is raised. However, the `setChosen` processes its list sequentially, as though the user were making the choices. The setting of any item will cause a currently set item to be unset.

3.3 Simple Widgets

Simple widgets have no subwidgets, at least explicitly. They form the basic set from which more complex widgets can be composed. Simple widgets can be modified in various ways. Parameters can be set at creation or dynamically. The widget can be wrapped functionally within another widget, which can interpose control over the inner widget's `boundsOf` function or its input and control events. As an example, a widget might set bounds that allow it to be stretched or shrunk. A programmer wishing to nail the widget down to a specific size could wrap the widget in another widget that could reset the size changes to zero (cf. Section 3.2.7).

The current widget set is not very rich or well-developed, nor particularly pleasing to the eye. There are a variety of notable omissions. Some of the implementations are not graphically efficient. However, the current set is enough to demonstrate how widgets should be written and interconnected within the **eXene** framework.

3.3.1 Buttons

The `Button` structure provides various simple buttons for user input. There are two different control semantics. The command button semantics simply calls a given function whenever the button is pressed. (A button is pressed by moving the mouse over the button and pressing any mouse button.) The button will usually indicate graphically that it has been pressed. In the more general semantics, the button provides an event registering button transitions and the button-down state. When the button is pressed, the button generates a `BtnDown` event indicating the mouse button pressed. It continues to generate `BtnDown` events until the button is released, at which point it generates a `BtnUp` event, or until the mouse leaves the window, at which point it generates a `BtnExit` event.

```
signature BUTTON =
  sig

    datatype arrow_dir
      = AD_Up
      | AD_Down
      | AD_Left
      | AD_Right

    datatype button_act
      = BtnDown of mbutton
      | BtnUp of mbutton
      | BtnExit

    type button

    val evtOf : button -> button_act event
    val widgetOf : button -> widget
    val setActive : (button * bool) -> unit
    val getActive : button -> bool

    val mkArrowBtn : root -> {
            backgrnd : color option,
            dir : arrow_dir,
            foregrnd : color option,
            sz : int
          } -> button

    val mkArrowCmd : root -> {
            action : unit -> unit,
            backgrnd : color option,
            dir : arrow_dir,
            foregrnd : color option,
            sz : int
          } -> button

    val mkTextBtn : root -> {
            rounded : bool,
            backgrnd : color option,
            foregrnd : color option,
            label : string
          } -> button

    val mkTextCmd : root -> {
            rounded : bool,
            action : unit -> unit,
            backgrnd : color option,
            foregrnd : color option,
            label : string
          } -> button

  end
```

To create a button, the user supplies a display root, and various display characteristics. The `foregrnd` and `backgrnd` colors specify optional choices for foreground and background colors. The default colors are black and white, respectively.

Arrow buttons require a direction parameter indicating which way the arrow should point, and a size, a square of that size being used as the natural size of the button. If `sz < 4`, the `LibBase.BadArg` exception is raised.

Text buttons require a text label for the buttons. The button sets its bounds to be just large enough to contain its text. If `rounded` is true, the text button appears within a rectangle with rounded corners. Otherwise, no frame is attached to a text button. Text buttons use the `8x13` font.

The action attached to a command button is invoked on `BtnUp`. Thus, a user can cancel an action after pressing a button by first moving the mouse off the button and then releasing the mouse button. The program should not synchronize on the event of a command button; a thread is spawned to monitor this event and invoke the action routine.

Buttons and toggle buttons (cf. Section 3.3.10) are actually constructed as the cross-product of items in view and control modules. The structure `ButtonView` describes a simple state-view protocol, and provides various visual button representations (arrows, text, icons, etc.) accepting this protocol. The `ButtonCtrl` and `ToggleCtrl` structures implement input control semantics using the protocol. The buttons provided in the `Button` and `Toggle` structures are just common combinations of a view and a control. A programmer can easily create new buttons using different combinations, or combining a new view with an old controller or vice versa.

N.B. It is the programmer's responsibility to provide a thread to synchronize on the button event supplied by the `evtOf` function. Otherwise, the button thread will block. The protocol for the delivery of button events is that an initial `BtnDown` and a terminating `BtnUp` or `BtnExit` event are guaranteed to be generated and must be synchronized upon. In the interim, the button widget will supply a `BtnDown` event every time the client synchronizes on the button event value.

3.3.2 Canvas

The canvas widget provides a drawing surface that can also be used as basis for building new widgets.

```
signature CANVAS =
  sig

    structure D : DRAWING

    type canvas

    val mkCanvas : root -> bounds -> (canvas * size * in_env)

    val widgetOf : canvas -> widget
    val sizeOf : canvas -> size
    val drawableOfCanvas : canvas -> D.drawable

  end
```

To create a canvas, the programmer provides a display root and a preferred bounds. The `mkCanvas` function returns a canvas on which to draw, the current size of the canvas and an input environment. The function `drawableOfCanvas` returns the drawable for the canvas. This can be used with the drawing operations of the `Drawing` module in the **eXene** library. The drawable is not available until the canvas widget is realized. A

call to `drawableOfCanvas` beforehand will cause the calling thread to block. The programmer is responsible for handling the input environment in the same manner as the programmer of a raw widget (cf. Chapter 5). There is also a `sizeOf` function, which returns the current size of the canvas. The actual event of the canvas being resized is reported, as usual, through the input environment.

3.3.3 Color rectangle

The `ColorRect` structure implements a widget that fills its window with a fixed color.

```
signature COLOR_RECT =
  sig

    val mkColorRect : root -> (color option * (unit -> bounds)) -> widget

  end
```

The `mkColorRect` takes a `root` value, plus an optional color and a bounds function. The resulting widget uses the given bounds function to specify its size requirements. If the color is `NONE`, black is used.

3.3.4 Divider

The `Divider` structure provides for flexible lines to be used as dividers between other widgets.

```
signature DIVIDER =
  sig

    val mkHorzDivider : root -> {color : color option, width : int} -> widget
    val mkVertDivider : root -> {color : color option, width : int} -> widget

  end
```

The color of the divider is specified by the color option; black is used by default. In a horizontal divider, the bounds are fixed to `width` in the vertical direction, and are arbitrarily flexible in the horizontal direction. A vertical divider is similar with the horizontal and vertical roles reversed. A negative `width` raises the `LibBase.BadArg` exception.

3.3.5 Framed widgets

Various basic widgets with text labels do not possess frames or boundaries. This allows them to be used in situations where frames are undesired. Following the **eXene** approach, a programmer can add borders by wrapping a widget in a `frame` widget (cf. Section 3.2.3). However, since certain framed widgets are such common idioms, the library provides three structures `FramedButton`, `FramedLabel` and `FramedToggle` that provide this wrapping. The resulting types `fr_button`, `fr_label` and `fr_toggle` have the same semantics as their unframed brethren. The constructor functions takes the same arguments, with the assumption that the shape will be rectangular and with the addition of a border width argument. If this last argument is negative, the exception `Frame.BadWidth` is raised. The foreground color is used as the frame color.

3.3.6 Label

A label widget allows the programmer to put unadorned text in the interface. The text and colors are mutable, under control of the programmer.

```
signature LABEL =
  sig

    type label

    val mkLabel : root -> {
            label : string,
            font : string option,
            foregrnd : color option,
            backgrnd : color option,
            align : halign
          } -> label

    val widgetOf : label -> widget
    val setLabel : label -> string -> unit
    val setBackground : label -> color option -> unit
    val setForeground : label -> color option -> unit

  end
```

A label is created by supplying a display root, an initial label string, an optional font, optional foreground and background colors and an alignment. On the screen, a label consists of the string written in the given foreground color (black by default) on the given background color (by default, the parent's background). The font argument specifies the name of the font to use (the `8x13` font is used by default). A label is naturally high enough to contain any string written in the font, and wide enough to contain the current string, plus a bit of padding around the string. A label specifies no shrinking or stretching. If the window provided for the label is larger than necessary, the text will be aligned within the window according to the `align` parameter.

The functions `setLabel`, `setForeground` and `setBackground` allow the program to modify the appearance of the label. If a new label is too large for the current window, the widget requests more space. Otherwise, the new text is written with the specified alignment.

3.3.7 Scrollbar

The scrollbar widget is used to indicate a position and a size, and to allow the user to change the position. Scrollbars are used most commonly to specify the view a window gives on a logically much larger display area, and to allow the user to pan the window over the display area.

```
signature SCROLLBAR =
  sig

    datatype scroll_evt
      = ScrUp of real
      | ScrDown of real
      | ScrStart of real
      | ScrMove of real
      | ScrEnd of real

    type scrollbar

    val mkHScrollbar : root -> {color : color option,sz : int} -> scrollbar
    val mkVScrollbar : root -> {color : color option,sz : int} -> scrollbar

    val evtOf : scrollbar -> scroll_evt event
    val widgetOf : scrollbar -> widget
    val setVals : scrollbar -> {sz:real option,top:real option} -> unit

  end
```

The functions `mkHScrollbar` and `mkVScrollbar` make horizontal and vertical scrollbars, respectively. The user supplies a display root, an optional color and a size. The size indicates the desired size of the minor axis, and is used in the bounds with no stretch or shrink. If `dim` is non-positive, the exception `LibBase.BadArg` is raised. Along the major axis, the scrollbar allows arbitrary stretching and shrinking. The scrollbar is drawn in the given color on its parent's background. By default, the color is black.

Scroll values fall in the range [0.0,1.0]. The `setVals` function allows an application or another widget to specify the current scrollbar values. Thus, the call

```
setVals sb {sz = SOME 0.2, top = SOME 0.5}
```

causes the start of the scrollbar curosr or "thumb" to be half way along the window and be one-fifth the window size. If either `sz` or `top` is not given, the present value is retained. The scrollbar values are additionally constrained by `top + sz <= 1.0`.

When the user interacts with the scrollbar, the scrollbar generates an event of type `scroll_evt`. The following protocols are associated with these events. `ScrStart` indicates that the user has begun to move the scrollbar cursor with the middle mouse button, with the top at the given relative position. The scrollbar will continue to generate `ScrMove` events as the user continues to move the thumb. When the user has stopped, the `ScrEnd` reports this with the final top position. During this interaction, the scrollbar automatically updates its values and display.

The `ScrUp` and `ScrDown` events are discrete events associated with the user clicking on the scrollbar with buttons 1 and 3 respectively. The application can interpret these events as appropriate. By convention, the user has indicated a desire to move the application window up or down, with the value being the indicated relative position. For example, one application might use `ScrUp` *v* to move its window up to the value corresponding to *v*. A text window might instead interpret this event by moving the top line of the window to the position indicated by the user. As these events are application-dependent, it is up to the application to reposition its view and then use `setVals` to register this new view with the scrollbar.

N.B. It is the programmer's responsibility to provide a thread to synchronize on the scrollbar event supplied

by the `evtOf` function. Otherwise, the scrollbar thread will block.

3.3.8 Slider

The slider widget provides an analogue means for a user to supply a numerical value. It is comparable to a scrollbar, having a slide piece that can be moved along a track using any mouse button. The value varies linearly with the position of the slide piece. Unlike a scrollbar, a slider supports a state of only a single value rather than a range of values.

```
signature SLIDER =
  sig
    type slider

    val mkHSlider : root -> {
            foregrnd : color option,
            wid : int,
            init : int,
            scale : int
          } -> slider

    val mkVSlider : root -> {
            foregrnd : color option,
            wid : int,
            init : int,
            scale : int
          } -> slider

    val widgetOf : slider -> widget
    val evtOf : slider -> int event
    val setValue : slider -> int -> unit
    val getValue : slider -> int
    val getScale : slider -> int

  end
```

To create a slider, the programmer supplies the display root, an optional foreground color (with black the default), an integer value specifying the "width" (the size of the slider perpendicular to the slide track), a scale and an initial value. The `mkHSlider` function returns a slider widget with a horizontal slide track, and `mkVSlider` returns one with a vertical track.

The slider is drawn in the specified color on its parent's background. The size of the slider perpendicular to the slide track is fixed to the value of `wid`. It is arbitrarily flexible along the dimension parallel to the slide track.

The slider's value will fall in the range `[0,scale]`. The initial slider value is `init`. The slider constructors require that `0 <= init <= scale` and `scale > 0`. They raise the `LibBase.BadArg` exception otherwise. Clients wishing a different range format (e.g., `[~5,10]`), a different range type (e.g., reals), or a non-linear function can wrap the widget as necessary.

The functions `getValue` and `getScale` allow one to query the slider's state. The `setValue` can be used to change the slider's value, which is reflected in the position of the slide piece. An invalid value will raise the `LibBase.BadArg` exception.

The `evtOf` function returns an integer event associated with the changing of the slider's value. As the user moves the slide piece along the track or the slider value is changed by `setValue`, the slider uses the event to report the current value.

N.B. It is the programmer's responsibility to provide a thread to synchronize on the slider event supplied by the `evtOf` function. Otherwise, the slider thread will block.

3.3.9 TextList

A text list provides the user with a list of strings, and allows the user to select items from the list. The user selects an item by clicking on it with the left mouse button. The user can unselect an item by clicking on it with the middle button. The widget reports user selections as events, which can be tied to application-specific actions. There are two types of list behavior, viz., lists allowing multiple selections and lists allowing at most one selection. In the single selection case, selecting a new item automatically causes a previously selected item to be unselected (cf. Section 3.2.9).

```
signature TEXT_LIST =
  sig

    exception BadIndex
    exception MultipleChoices

    type 'a text_list
    datatype 'a list_evt = Set of 'a | Unset of 'a
    datatype list_mode = OneSet | MultiSet

    type 'a list_item (* = (string * 'a * wstate) *)
    val mkItem : (string * 'a * W.wstate) -> 'a list_item

    val mkHList : root -> {
            mode : list_mode,
            backgrnd : color option,
            foregrnd : color option,
            items : '2a list_item list
          } -> '2a text_list

    val mkVList : root -> {
            mode : list_mode,
            backgrnd : color option,
            foregrnd : color option,
            items : '2a list_item list
          } -> '2a text_list

    val evtOf : 'a text_list -> 'a list_evt event
    val widgetOf : 'a text_list -> widget

    val setChosen : 'a text_list -> (int * bool) list -> unit
    val setActive : 'a text_list -> (int * bool) list -> unit
    val getChosen : 'a text_list -> int list
    val getState : 'a text_list -> wstate list

  end
```

To create a text list, the programmer supplies the display root, optional foreground and background colors (with black and white the respective defaults), the selection mode of the widget, and a list of items. The `mkHList` function returns a horizontal list widget while `mkVList` returns one with a vertical orientation. The selection mode specifies which of the two list behaviors is desired, with `OneSet` and `MultiSet` corresponding respectively to allowing at most one selection and allowing multiple selections. An item is essentially a triple of a string, a value and an initial state. The string specifies the text displayed in the list. The value is returned when the user selects the item. The initial state specifies whether or not the item is active, and whether or not the item is set.

The set of selected and active items can be controlled by the program using the `setChosen` and `setActive` functions. The `getChosen` and `getState` allow the program to query the state of the list.

The `evtOf` function returns an `'a list_evt` event associated with items being selected or unselected. The `list_evt` indicates whether the user has selected (`Set`) or unselected (`Unset`) an item, along with the value associated with the item. In the `OneSet` mode, in which the widget automatically unsets the currently selected item when a new one is chosen, the widget supplies an `Unset` event before reporting the `Set` event.

The constraints and exceptions associated with creating and using a text list are described in Section 3.2.9.

N.B. It is the programmer's responsibility to provide a thread to synchronize on the list event supplied by the `evtOf` function. Otherwise, the list widget will block.

3.3.10 Toggles

Toggles are buttons that maintain an on-off state. The user changes the toggle's state by clicking on the button with any mouse button. As with ordinary buttons, toggle buttons can also be active or inactive. A button's display usually indicates both aspects of the button's state.

```
signature TOGGLE =
  sig
    type toggle

    val widgetOf : toggle -> widget
    val getState : toggle -> bool
    val setState : (toggle * bool) -> unit
    val setActive : (toggle * bool) -> unit
    val getActive : toggle -> bool

    val mkToggleCheck : root -> {
            state : wstate,
            action : bool -> unit,
            color : color option,
            sz : int
          } -> toggle

    val mkToggleText : root -> {
            state : wstate,
            rounded : bool,
            action : bool -> unit,
            backgrnd : color option,
            foregrnd : color option,
            label : string
          } -> toggle

    val mkToggleSwitch : root -> {
            state : wstate,
            action : bool -> unit,
            backgrnd : color option,
            foregrnd : color option
          } -> toggle

    val mkToggleCircle : root -> {
            state : wstate,
            action : bool -> unit,
            backgrnd : color option,
            foregrnd : color option,
            radius : int
          } -> toggle

    val mkToggleIcon : root -> {
            state : wstate,
            action : bool -> unit,
            backgrnd : color option,
            foregrnd : color option,
            icon : tile
          } -> toggle

  end
```

All constructors take an initial toggle state and an action function that is called whenever the toggle changes from on to off or from off to on. The new value, with on mapped to true, off mapped to false, is passed to the action function. The action function is not invoked on the initial toggle state. The various toggle constructors produce widgets differing only in the appearance of the widget. These differences are reflected in the remaining constructor arguments.

`mkToggleCheck` produces a check mark toggle within a square widget. The `sz` parameter indicates the desired size of a side. The widget bounds are set to a fixed square of this size. If `sz < 14`, the `LibBase.BadArg` exception is raised. The toggle is drawn in the specified color (black by default) on the parent's background.

`mkToggleText` produces a text button toggle. The display parameters have the same use as in `mkTextBtn` (cf. Section 3.3.1).

The `mkToggleSwitch` toggle is represented by a rocker switch of fixed bounds. The switch is drawn in the foreground color (black by default) on a field of the background color (white by default).

The `mkCircleToggle` button appears as a circular button, with the `radius` parameter specifying the desired radius. The widget's bounds will be a fixed square slightly larger than what is necessary to hold a circle of the given radius. If `radius < 4`, the `LibBase.BadArg` exception is raised. The widget is drawn in the foreground color (black by default) on a field of the background color (white by default).

The toggle produced by `mkIconToggle` displays the argument icon, using the foreground and background colors supplied. As usual, these colors are black and white, respectively, by default. The widget's bounds are fixed at the size of the icon.

As toggles are implemented using the same button views as the buttons described above in Section 3.3.1, most of the remarks in that section apply here. The only difference is that toggles supply the functions `getState` to query the toggle state and `setState` to set the toggle state. The program can change a toggle's state even if the toggle is inactive.

3.4 Text Widgets

Widgets providing textual input form a special class of widgets. Going beyond a simple mouse interface, these widgets are combinations of virtual terminals and editors. The underlying model is more complex, but the user is familiar with the model and expects a richer, more complicated set of interactions. In this section, we describe an initial set of widgets whose interface is principally based on the keyboard and characters. The text widgets are probably the least fully developed of the **eXene** widget set. For example, there is currently no support for text selection.

3.4.1 StrEdit

The `StrEdit` structure provides a simple string editing widget. As the user types, the corresponding characters are entered at the cursor position. The backspace character (`"\^H"`) can be used to erase the character preceding the cursor. The entire string can be deleted by typing `"\^X"`. The user can reposition the cursor by clicking the mouse on the desired character. If the insertion or deletion of a character would cause the cursor to move off the window, the widget shifts the window's view appropriately.

```
signature STREDIT =
  sig

    type str_edit

    val mkStrEdit : root -> {
            foregrnd : color option,
            backgrnd : color option,
            initval : string,
            minlen : int
          } -> str_edit

    val setString : str_edit -> string -> unit
    val getString : str_edit -> string
    val shiftWin  : str_edit -> int -> unit
    val widgetOf : str_edit -> widget

  end
```

The `mkStrEdit` function takes a display root, optional foreground and background colors (black and white, by default), an initial string and a minimum length. Initially, the cursor is placed at the end of the string.

The widget uses the font `9x15`. The natural height of the widget is the font height, with no vertical shrinking or stretching. Horizontally, the natural size specifies enough space to display all the characters in the string or `minlen` characters, which ever is larger. The widget will expand indefinitely, but will only shrink to the `minlen` minimum. Whenever the widget's string goes from fitting to not fitting within the widget's window, or vice versa, it requests its parent to resize it.

Note that the widget does not provide the user the ability to move the widget's window over the text. It does provide the `shiftWin` function, which an application can use to provide this interface (cf. the `FieldEdit` widget below). Using `shiftWin v` moves the widget view `|v|` characters to the left if `v` is negative and to the right if `v` is positive.

The `setString` and `getString` functions allow the application to query and set the widget's string.

3.4.2 FieldEdit

The `FieldEdit` structure provides a widget derived from the string edit widget above. The interface and interaction are almost identical. The only difference is that the field edit widget provides scroll buttons to allow the user to move the view of the underlying text when it does not all fit within the window. The buttons are only made available when this situation arises.

```
signature FIELD_EDIT =
  sig

    type field_edit

    val mkFieldEdit : root -> {
            foregrnd : color option,
            backgrnd : color option,
            initval : string,
            minlen : int
          } -> field_edit

    val setString : field_edit -> string -> unit
    val getString : field_edit -> string
    val widgetOf : field_edit -> widget

  end
```

3.4.3 Virtual terminal

The virtual terminal widget provides a simple way to support traditional IO stream-based applications[1]. It is implemented on top of the text widget (see Section 3.4.4), adding a device driver for the keyboard and the `instream`/`outstream` interface. The structure `Vtty` has the signature `VTTY`:

```
signature VTTY =
  sig
    structure CIO : CONCUR_IO

    type vtty

    val mkVtty : root -> {rows : int, cols : int} -> vtty
    val openVtty : vtty -> (CIO.instream * CIO.outstream)
    val widgetOf : vtty -> widget

  end (* VTTY *)
```

See the **CML** manual for the details of the operations provided by the `CIO` structure ([Rep90]). Writing on the `outstream` displays text in the window. User input is line-buffered: it is only available on the `instream` after the user has typed a carriage return (`"M̂"`) or newline (`"\n"`). An input line can be edited, with backspace (`"\ĥ"`) and delete (`"\127"`) erasing the previously input character. Tab characters are not handled correctly, nor does the `vtty` provide a visible cursor.

3.4.4 Text

The text widget is a low-level widget for managing a window of text. It provides limited highlighting, and a single font; a prototype text widget supporting multipl fontsand color has been implemented, and will be included in the next release.

[1] Thanks to Thomas Yan for finishing our implementation of this widget.

```
signature TEXT_WIDGET =
  sig

    datatype char_coord = ChrCrd of {col : int, row : int}

    type text_widget

    val mkTextWidget : root -> {rows : int, cols : int} -> text_widget

    val widgetOf   : text_widget -> widget
    val charSizeOf : text_widget -> {rows : int, cols : int}
    val sizeOf     : text_widget -> size

    val ptToCoord   : text_widget -> point -> char_coord
    val coordToRect : text_widget -> char_coord -> rect

    val scrollUp   : text_widget -> {from : int, nlines : int} -> unit
    val scrollDown : text_widget -> {from : int, nlines : int} -> unit

    val writeText : text_widget -> {at: char_coord, text : string} -> unit
    val highlightText : text_widget -> {at: char_coord, text : string} -> unit

    val insertLn : text_widget -> {lnum : int, text : string} -> unit
    val deleteLn : text_widget -> int -> unit
    val deleteLns : text_widget -> {lnum : int, nlines : int} -> unit

    val clearToEOL : text_widget -> char_coord -> unit
    val clearToEOS : text_widget -> char_coord -> unit

    val moveCursor : text_widget -> char_coord -> unit
    val cursorPos  : text_widget -> char_coord
    val cursorOn   : text_widget -> unit
    val cursorOff  : text_widget -> unit

  end (* TEXT_WIDGET *)
```

Chapter 4

Menus

The **eXene** widgets currently provide a simple form of pop-up menu support in the structure `SimpleMenu`. Figure 4.1 has the signature of this structure. A `menu` value specifies the structure of a menu and the value

```
signature SIMPLE_MENU =
  sig
    datatype 'a menu = MENU of 'a menu_item list
    and 'a menu_item
      = MenuItem of (string * 'a)
      | Submenu of (string * 'a menu)

    val attachMenu : (widget * mbutton list * '1a menu)
          -> (widget * '1a event)

    val attachLabeledMenu : (widget * mbutton list * string * '1a menu)
          -> (widget * '1a event)

    datatype menu_pos = Absolute of point | Item of int

    datatype where_info =
      WI of
          but : mbutton,
          pt : point,
          scr_pt : point,
          time : time


    val buttonMenu : (widget * mbutton list * '1a menu * (where_info -> menu_pos))
          -> (widget * '1a event)

    val popupMenu : (root * '1a menu * string option)
          -> (mbutton * menu_pos * point * mouse_msg addr_msg event)
          -> '1a option event

  end
```

Figure 4.1: Simple menus

associated with each entry. The `Submenu` constructor is used for defining hierarchical menus. The display

form of these menus is essentially that used by **twm** [**?**]. The menu remains displayed and active as long as some mouse button is depressed. The user's choice corresponds to the item under the mouse cursor when the last mouse button is released.

Once a menu has been defined, there are various ways it can be used. The simplest way is to attach it to a widget using one of the two attach functions. For menus without labels, use `attachMenu`; if a label is desired, then use `attachLabeledMenu`. The result of attaching a menu to a widget is a new widget and an event value that provides the user's menu choices as a stream of **CML** events. When attaching a menu to a widget, the programmer specifies the mouse buttons that are to be used to pop up the menu. If the user presses a specified mouse button while the cursor is over the widget to which the menu is attached, then the menu will be popped up. Any other mouse button press events, and all other mouse events, are passed on to the underlying widget. If the user selects a menu item, then the associated value is reported through the event. If the user does not select an item, no event is generated.

Figure 2.3 and the accompanying text provide an example of the this use of menus.

The `buttonMenu` function is the same as `attachMenu`, except it provides control over menu placement. In particular, as its name suggests, `buttonMenu` can be used to implement menu buttons (see Section 4.0.5). When a mouse button press triggers the display of the menu, the user-supplied function is invoked, being passed a value of type `where_info`. This is essentially the information supplied by the `MOUSE_FirstDown` event corresponding to the button press. The function returns a hint as to where the menu should appear. If the function returns `Absolute` *pt*, the upper left corner of the menu will be placed at *pt*, given in screen coordinates. If the function returns `Item` *n*, the menu will be positioned so that the mouse cursor is centered over item *n* of the menu, indexed from 0. The first form can be used to implement menu buttons; the latter form can be used to redisplay a menu with the last selected item chosen by default. The code will honor the position hint unless it has to shift the menu to make sure it fits on the screen.

At times, these high-level menu interfaces to not provide enough control. For these situations, the structure provides the `popupMenu` function. This low-level routine takes a `root`, a `menu` and an optional string. It returns a function that can be used to activate a pop-up menu on the given display. To use the activation function, a thread waits for some mouse button press and then calls the activation function with a tuple `(`*btn*`,` *pos*`,` *pt*`,` *mouse*`)`. The *btn* argument denotes the pressed mouse button. The *pos* parameter provides control of menu placement, as described in the previous paragraph. The *pt* parameter is the mouse position, in screen coordinates, reported with the button press event. Finally, the *mouse* parameter is the widget's mouse event stream. The function returns a **CML** event. The menu will be displayed and managed by a separate thread, getting its input from the mouse event stream. The protocol requires that the calling thread does not read events from that stream until it is notified, through synchronization on the menu event, that the menu thread is done. If the user did not make a choice, the event will evaluate to `NONE`. Otherwise, the value associated with the chosen menu item will be supplied.

4.0.5 Menu button

A menu button is a text button with a menu attached. When the user presses any mouse button on the menu button, the menu is displayed as a pull-down menu. They can be combined to form menu bars.

```
signature MENU_BUTTON =
  sig
    structure Menu : SIMPLE_MENU

    val mkMenuButton : root -> (string * 'a Menu.menu)
          -> (widget * 'a event)

  end
```

The `mkMenuButton` function takes a `root`, and a label and a menu, and returns a widget plus an event. The widget appears as a plain rectangular text button, using the supplied label. If the user makes a selection using the menu, the selection is reported through the event value.

Chapter 5

Widget Internals

We now turn to the internal widget structure and the low-level details of building a widget. The creation of a `widget` value is usually a simple affair, involving some initial parameter computation, the allocation of certain resources such as fonts, colors and pixmaps, and perhaps the spawning of a thread to encapsulate mutable widget data. No **eXene** windows are created at this stage. The widget value returned by `mkWidget` encapsulates the associated `root` value, a `boundsOf` function and a `realize` function (see Section 2.3). The semantics of the `boundsOf` function were described above. Here we discuss what should happen when a widget's `realize` function is called.

First, we note that in the **eXene** widget library, the parent widget controls the child's window resources. The parent allocates the child's window; it positions the window; it changes its size; it deletes it. If the child needs any of these actions performed, then it asks the parent to perform the action. A child should never directly alter the external configuration of its window; it should only deal with what is inside its window.

The parent widget invokes the child's `realize` function, passing in three arguments. One argument is the input environment. This is the value through which the child receives all of its input and by which it communicates with its parent. This will be discussed in detail below. The remaining arguments are the child's window and its size. The window's origin and size are presumably set by the parent using the child's `boundsOf` function. This window is the child's canvas. It can draw on the window using any of the functions supplied in the `Draw` structure in the base library. The size is supplied for convenience.

A widget's `realize` function must do the following: configure itself corresponding to the given size and spawn the necessary threads to handle graphics operations and events on its input environment. If the widget is a composite widget, then the `realize` function must also layout its children, allocate their windows, arrange to handle input to and from the children, and recursively call their `realize` functions. A composite widget is also responsible for mapping its children's windows (using the **eXene** function `mapWin`). A simple widget should not map or unmap itself.

All window system events come to a window through its input environment. This environment is defined in the `WindowEnv` structure.

```
datatype in_env = InEnv of {
    m : mouse_msg addr_msg event,
    k : kbd_msg addr_msg event,
    ci : cmd_in addr_msg event,
    co : cmd_out -> unit event
  }
```

It is assumed that a widget will spawn one or more threads to be able to synchronize on the `m`, `k`, and `ci` events immediately. The `addr_msg` implicitly contains a window path that specifies the target window of the accompanying value. This path is only of concern to composite widgets, and will be discussed below. For now, we assume that a widget has received an event targeted for it and has stripped off the window path part (using the **eXene** function `msgBodyOf`) and is left with the base value.

5.1 Keyboard and Mouse Events

Keyboard events are reported as either key presses or key releases. The related values specify an **X** keysym value and the state of the modifier keys. Keysyms can be translated into strings using functions related to the `translation` type of the base library. Mouse events correspond to mouse motion, button presses and releases, and window enter and leave events. Note that when a window receives a mouse button press, it is guaranteed to receive all mouse events on its mouse stream until all mouse buttons have been released. This corresponds to an *active grab* in **X** terminology (cf. *The eXene Library Manual* and [Nye90]). Also note that the first mouse button event a widget receives is a `MOUSE_FirstDown` event, and the last will be a `MOUSE_LastUp` event. Keyboard and mouse events also provide a synchronization value that can be used by composite widgets to coordinate event routing with window changes.

5.2 Control Events

The events received via the `ci` event correspond to window system events not directly related to user input. A widget should act as follows upon one of these events.

- A `CI_Redraw` message corresponds to an **X** `Expose` event, informing the widget that part of its window has been corrupted and it should redraw those parts. The event provides the list of rectangles, in the window's coordinate system, that should be redrawn. It is always safe, though not necessarily efficient, to redraw the entire window.
- A `CI_Resize` message corresponds to a notification that the widget's window has changed size (or position). The rectangle parameter specifies its new size and its origin in its parent's coordinate system. The actual resizing of the window was performed by the parent. On receiving this event, a widget should reconfigure itself, recomputing whatever window size dependent parameters it uses. In particular, a composite widget should recompute the layout of its children and, if necessary, call the `moveWin`, `resizeWin` or `moveAndResizeWin` to reposition them. The corresponding `CI_Resize` messages will be automatically generated by the system and delivered to the child through its input environment. Upon receiving a `CI_Resize` event, a widget should not automatically redraw itself. If this is necessary, it will receive a `CI_Redraw` event. Note that a widget is informed of its initial window size when its `realize` function is called.

- A `CI_OwnDeath` tells a widget that its window has been destroyed. At this point, it should cease from performing any further output to its window. **N.B.** It must still service its mouse and keyboard events, as these events may still be queued somewhere in the hierarchy. One approach is to attach null loops to these events.
- The `CI_ChildBirth` and `CI_ChildDeath` events are used by composite widgets to synchronize the routing of input.

A widget may require certain services of its parent. That is the purpose of the `co` field of the input environment. A `CO_ResizeReq` message passed to its parent informs the parent that the child's size needs have changed. The parent should call the widget's `boundsOf` function to determine these needs and try to meet them by repositioning the child's window. The parent is not obligated to meet these needs, and the child cannot assume anything has changed until it receives a `CI_Resize` event. The `CO_KillReq` message informs the parent that the child wishes its window destroyed. The parent should attempt to perform this service, using the `destroyWin` command. It will be informed of the loss of its window via a `CI_OwnDeath` message (see the item on handling the `CI_OwnDeath` message above).

5.3 Composite Widgets

A composite widget, containing one or more child widgets, must handle its own input environment, as described above, as well as manage its children. This involves several tasks. One task is the layout of child widgets within its window. Another task is creating a child's window and input environment, By convention, widget windows in **eXene** do not explicitly use the border or background property of windows in **X**, but rely on wrapper widgets (e.g., `frame` and `background`) to supply these effects. We adopted this convention as being cleaner and more consistent, and in order to better divorce the widget level from the specific **X** graphics model. The utility function `wrapCreate` conforms to this convention, and the writer of composite widgets is encouraged to use this function to create child windows. The `createWinEnv` function from the base library can be used to create the necessary input environment. Additional tasks require calling a child's `realize` function with its window and environment, and mapping the child's window.

A further task concerns servicing requests from a child. For each child and its input environment, the parent widget maintains the corresponding output environment. Upon receiving a `CO_ResizeReq` or a `CO_KillReq`, the composite widget should attempt to handle this as described above. Note that a parent widget can attempt to resize the child within the parent's current bounds, or has the opportunity of percolating a resize request further up the hierarchy.

We also emphasize the possibility of deadlock implicit in the two-way protocols between a composite widget and its children. For example, a composite widget might be in the process of positioning its children, during which it calls a child's `boundsOf` function. Meanwhile, because of user input, the child might be sending a resize request to its parent, with deadlock possibly ensuing. Because of the synchronous communication and hierarchical distribution of events used in **eXene**, this local deadlock could quickly disseminate throughout an application. It is the responsibility of the programmer implementing a widget to code against such possibilities. In the specific case of child-to-parent communication using control messages, we assume the additional specification that it is the parent's responsibility to always be receptive to control messages from its children. The `Widget` structure provides the function `wrapQueue` that wraps a queue

around a **CML** event. This can be used to safely queue child requests on the `co` event if other processing is going on.

The final task of a composite widget is handling the routing of all events to widgets in its subtree. Each event coming through its input environment may be targeted for the composite widget or one of its children or their children. Each input event is implicitly tagged with a routing path to the correct window. The `Interact` structure provides various functions with which a widget can determine if a message is for itself or one of its children. If the message is targeted for a child, the message must be distributed to that child's input environment.

The **eXene** widget library's `Router` structure provides functions that can handle typical routing scenarios.

```
signature ROUTER =
  sig

    structure EXB : EXENE_BASE
    structure Interact : INTERACT

    exception NotFound

    type router

    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

    val routePair : Interact.in_env * Interact.out_env * Interact.out_env -> unit
  end
```

At realization time, a composite widget can create a router using the `mkRouter` function, passing it the input environment that it received via the invocation of its `realize` function. In addition, the composite widget should create an new input/output environment pair, and pass the output component as the second argument to `mkRouter`. The widget will then receive its input from the corresponding input component of the environment pair. In addition, for each child widget, the composite widget should create an input/output environment pair. The input environment should be handed to the child through its `realize` function. The child's window and output environment are registered with the router. If desired, a list of these pairs can be passed as the third argument in creating the router. Alternatively, these pairs can be incrementally inserted and deleted from the router using `addChild` and `delChild`, respectively. The `getChildEnv` returns the output environment that was registered with the given window. The exception `NotFound` is raised if no such environment is registered.

For composite widget's with a single child, the `Router` module provides the `routePair` function. This creates a router that takes the composite widget's initial input environment, the new output environment for the composite widget and the output environment for the child widget.

With either of these routers, all events for children will be automatically routed to them. Only events targeted specifically to the composite widget will be received through its new input environment. We remark that this routing scheme is tentative, and may be replaced by more appropriate mechanisms in later releases.

The following schema illustrates the routing technique described above, plus a mechanism for allocating

child windows and instantiating child widgets.

```
fun realize {env, win, sz} = let
      val (myInEnv, myOutEnv) = createWinEnv ()
      val router = mkRouter (env, myInEnv, [])
      fun doChild (childWidget, childPt, childSz) =
            val crect = mkRect(childPt, childSz)
            val cwin = wrapCreate (win, crect)
            val (cinenv, coutenv as OutEnv{co=childco,...}) = createWinEnv ()
            in
              addChild router (cwin, coutenv);
              realizeFn childWidget {env=cinenv, win=cwin, sz=childSz};
              mapWin cwin
            end

      fun loop () =
      ...
          (*
           * obtain input events from myInEnv; handle child requests
           * received on childco.
           *)
      ...
     in
       app doChild children;
       spawn loop
     end
```

The `wrapCreate` function is a utility function provided by the `Widget` structure for creating child windows. It returns a subwindow of the given parent with the given placement. The `LibBase.BadArg` exception is raised if the height or width of the rectangle is not positive.

A composite widget may wish to allow widgets to be inserted and removed from it dynamically. If a widget is inserted, it should be assumed to be unrealized. The composite widget should reposition its children in light of the new widget, and allocate resources for and realize the new widget. If a widget is to be removed from a composite widget, the parent widget should destroy the child's window and remove the child from the router.

5.4 Miscellany

Rather than have a programmer write a widget from scratch, **eXene** promotes creating, when possible, a new widget by wrapping an old one. This is the compositional technique discussed in Section 2.4, and supported by the hierarchical routing of events. The wrapping widget interposes itself between the parent and the child, and alters the event streams between parent and child. This idiom is common enough that the `Widget` structure provides several functions to support it.

```
signature WIDGET =
  sig

    ...
    val filterMouse : widget ->
      (widget *
        ((mouse_msg addr_msg event * mouse_msg addr_msg chan) event))

    val filterKey : widget ->
      (widget *
        ((kbd_msg addr_msg event * kbd_msg addr_msg chan) event))

    val filterCmd : widget ->
      (widget *
        ((cmd_in addr_msg event * cmd_in addr_msg chan) event))

    val ignoreMouse : widget -> widget
    val ignoreKey : widget -> widget
    ...
  end
```

The `filterMouse` function takes a widget and returns a new widget wrapped around the old widget. The new widget lets the application have access to the plumbing of mouse events to the original widget. Before the original widget is realized, the wrapper widget uses the returned event value to supply the receiving end of the widget's stream of mouse events, plus a channel by which the application can transmit mouse events to the original widget. As usual with these event values, the application must be capable of synchronizing on the mouse event with little delay. The `filterKey` and `filterCmd` functions are the obvious analogues. The `ignoreMouse` and `ignoreKey` wrappers filter out all mouse and keyboard events, respectively, from the original widget.

The `Widget` structure has a utility function `val colorOf : root -> color_spec -> color` which provides a widget-level translation of color specifications into colors. The call `colorOf` *root spec* is equivalent to `colorOfScr (screenOf` *root*`)` *spec*.

Bibliography

[NO90] Nye, A. and T. O'Reilly. *X Toolkit Intrinsics Programming Manual*, vol. 4. O'Reilly & Associates, Inc., 1990.

[Nye90] Nye, A. *Xlib Programming Manual*, vol. 1. O'Reilly & Associates, Inc., 1990.

[Rep90] Reppy, J. H. *Concurrent programming with events – The Concurrent ML manual*. Department of Computer Science, Cornell University, Ithaca, N.Y., November 1990. (Last revised February 1993).

[RG93] Reppy, J. H. and E. R. Gansner. *The eXene Library Manual*. AT&T Bell Laboratories, Murray Hill, N.J. 07974, February 1993. Included in the eXene distribution.