On the Future of eXene

Dustin deBoer and Alley Stoughton[1]

Kansas State University
Department of Computing and Information Sciences
234 Nichols Hall
Manhattan, KS 66506, USA
WWW: `cis.ksu.edu/~stough/eXene/`
E-mail (deBoer): `ddeboer@cis.ksu.edu`
E-mail (Stoughton): `stough@cis.ksu.edu`

July 18, 2005

Abstract. Up through the mid-1990s, Gansner and Reppy designed and implemented eXene—a multi-threaded, higher-order user-interface toolkit for the X window system. EXene is implemented in Concurrent ML (CML), which is a Standard ML of New Jersey (SML/NJ) library, and is provided as part of the SML/NJ distribution. EXene has many appealing attributes and is certainly usable in its current state. But it suffers from some deficiencies which have limited its use, and there are a number of ways in which it could usefully be improved. In this paper, we describe our efforts at restarting the development of eXene, saying what we have accomplished so far, and detailing our plans for the future.

1 Introduction

From the mid-1980s until the mid-1990s, Emden Gansner and John Reppy designed and developed a multi-threaded, higher-order user-interface toolkit for the X window system that they called eXene [4, 6]. EXene is implemented in Concurrent ML (CML) [14], which is actually a set of Standard ML of New Jersey (SML/NJ) [1] libraries. CML supports very light-weight threads and provides flexible selective communication

[1]Please direct correspondence to Stoughton.

via its use of first-class synchronous events. Unlike many X toolkits, e.g., [11, 12], eXene is multi-threaded, allowing processing that would naturally be concurrent to be expressed directly, instead of having to be scheduled as part of an event-loop. This makes it much easier to write applications that appear to run concurrently. EXene communicates directly with an X server, but provides a higher-level interface than does Xlib [8, 9], making a programmer's life easier, and handling some operations locally, without contacting the X server. A widget in eXene has encapsulated state that is managed by its own thread; thus it can be thought of as an object, and we often refer to operations on widgets as methods. When a hierarchy of widgets is realized, each widget is given a window to manage, along with an input environment on which it communicates with its parent. EXene turns X events into addressed messages, which are routed to a destination widget through a widget hierarchy. This organization makes it easy to provide wrappers for widgets that change their behavior.

In the early 1990s, Gansner and Reppy developed various applications using eXene [6], and several other researchers used eXene to develop graphical user-interfaces (GUIs) for their applications [2, 7, 16]. This use of eXene demonstrated its elegance, utility and acceptable performance. But in more recent years, both the development of eXene and its use by others has dwindled. Gansner and Reppy have put their time into other projects, and potential users have apparently been put off by the lack of up-to-date documentation and ongoing development, because some aspects of eXene, like support for X resources and authorization, are unfinished, and since some needed features, like the handle of input focus, are missing. When developing a pretty-printing library using eXene [18], the second author wasn't aware of significant ongoing development or use of eXene.

During the last two years, as an outgrowth of the second author's use of CML and eXene in a graduate course, our group at K-State has been attempting to restart the development of eXene. In the following sections, we will describe what we have achieved so far, as well as our plans for the future. In Section 2, we consider widget programming conventions, explaining our scheme for avoiding deadlock and maximizing concurrency. In Section 3, we describe our approach to handling input focus. Section 4 is concerned with customization using X resources. In Section 5, we consider support for X authorization. In Section 6, we describe our plan for supporting X selections. In Section 7, we consider widget bounds and the layout widget. In Section 8, we mention the past and present widget development projects undertaken at K-State. Finally, in Section 9, we mention our plans for improving eXene's documentation.

EXene is an open source project, and we encourage members of the SML commu-

nity to become involved in its design and development. For more information, contact the second author. Because X servers now run on all major platforms, we believe that a revitalized eXene has the potential to become popular with SML developers.

2 Widget Programming Conventions

When an eXene widget is realized, it is granted one X window upon which to render itself and an input environment on which messages will be received. This input environment consists of keyboard, mouse and control addressed message input streams, represented as CML events. Each of these input streams has a corresponding output stream upon which the messages are sent by its parent. Composite widgets—widgets containing one or more child widgets, such as layout widgets—maintain output streams corresponding to the child widgets' input streams. A composite widget must contain a router that determines the child widget (window) a message is destined for, and then sends it on to the child widget via the output stream corresponding to that child's input environment. In addition, many widgets offer CML events to the application—for example, a button widget may offer a button activity event on which `BtnDown` and `BtnUp` events may be received. The button widget maintains an output stream over which these events are sent to the user application.

EXene is intended to be a fully concurrent system [6]. To this end, each widget's state is normally encapsulated in its own thread. E.g., a button widget's state is represented by a thread that listens for user input on its input streams and sends selected messages to the application on its output stream. In principle, this would allow all widgets in an application to execute concurrently—e.g., if a particular widget was executing a long-running computation, other widgets could continue to execute, even responding to further user input. However, because communication in CML is synchronous, and since eXene's message router doesn't buffer messages, it turns out that some of this possibility for concurrency is lost.

Because CML communication and the widget input/output streams are synchronous, any failure of a widget or application to respond to input in a timely manner has the potential to block the execution of other widget threads. Consider a widget E that performs some extensive computation, and that is part of a composite widget C. While E is performing the computation, any messages that the router thread of C attempts to send to E will be blocked. Therefore, no further input messages will reach any other children of C, and furthermore, if C itself is part of another composite widget, that parent router thread will also become blocked. This cascading

blocking may also arise if the application itself fails to respond (at all, or in a timely manner) to messages from a widget. The failure of any recipient of messages in an application to respond to input will eventually block execution of all widget threads in the same shell (the eXene abstraction for a top-level window).

In addition, some possibilities for deadlock arise by virtue of the bidirectional communication between widgets and applications, and between parent widgets and child widgets. A widget typically implements its methods by creating a request channel (stream) over which operation requests may be sent by applications. Optionally, return values may be sent via synchronous variables specified in the requests. This functionality is hidden from the application programmer, who sees only a method that blocks until complete; the application programmer cannot selectively communicate over the widget request channels. Now, suppose that an application A contains a widget W with a method m. Further suppose that the application applies m to W. The application is now blocked waiting for a reply (or receipt of the request message) from W. But simultaneously, W may be attempting to send a message to A without selective communication[2] (without allowing for the possibility to also receive the method request message). This causes both W and A to be blocked waiting on the other. Unfortunately, this blocking will soon cascade to the rest of the shell. Similarly, a child widget might be trying to communicate with its parent, while the parent was trying to communicate with the child.

In [5, p. 42], Gansner and Reppy say that, in communication between a parent widget and one of its children, the parent has the responsibility to be responsive, and that queuing of a child's messages to its parent could be used to avoid deadlock. (A `wrapQueue` function is provided for this very purpose.) We propose using this idea of parental responsibility as the basis for our widget programming conventions, and we apply it not just in parent widget/child widget communication, but also in communications between a widget (thought of as the parent) and the application (thought of as the child). In general, we say that

Parents should be more responsible than children.

We feel that widgets should be tolerant of errors in user applications, and composite widgets should be tolerant of errors in their child widgets.

[2]This deadlock situation is avoidable if each widget uses selective communication for each output message sent—however, it is not trivial to design widgets in this way. In fact, this was how the second author modified the scrollbar widget to avoid deadlock; this modification is incorporated in the current eXene release.

Our approach to making parents more responsible than children does use queuing, but in a different way than was suggested by [5, p. 42]. We recommend that parent widgets queue messages sent to child widgets or applications.[3] This prevents parent threads from being blocked by error-prone or slow children, at the cost of extra queue threads and some buffer space. It may sometimes be necessary to flush the queue—perhaps at the request of an application that knows that only future messages are of interest. Because all queued messages originated with the user, there is little risk of the message queues becoming especially long.

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

This convention suggests the following life cycle of a widget:

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

- **Bounds Determination**. The `boundsOf` method of the widget is called, determining the requested bounds of the widget. The `boundsOf` method should never be called again in the lifetime of the widget; calling `boundsOf` again will raise the exception `BoundsFunctionAlreadyCalled`.

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

  Note that, if the widget's desired size changes after its bounds function has been called but before it is realized, its parent won't know what this desired size is.

[3] Top-level windows already buffer the X events they receive, before turning them into addressed messages destined for descendants.

Some widgets in the current eXene release suffer from this defect. It can be avoided by having the widget remember that it should ask its parent to resize it after realization.

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

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

We have implemented the queuing of messages sent by composite widget routers, as well as those sent by some widgets to applications. In addition, we have implemented a version of the button widget in which the button activity queue is flushed whenever a button is made inactive (unresponsive to user input). We have not yet enforced the above widget cycle on the existing eXene widgets.

To summarize, we recommend the following eXene widget programming conventions:

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

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

- A widget's methods must be guaranteed to terminate (ideally, in a timely fashion).

- Attempts by a child widget to send messages to its parent should always succeed (ideally, in a timely manner).

3 Handling Input Focus

By default, the keyboard input focus of an X application is set to the root window, which means that keyboard input is sent to the window currently pointed to by the mouse [13, p. 612] [10]. This functionality can be annoying to deal with in eXene

applications, particularly when trying to enter text in an application with multiple text input fields. The X protocol provides the `SetInputFocus` request for assigning keyboard focus to a particular window. This allows, e.g., an application to assign a text input widget input focus so that movement of the mouse pointer will not affect the user's ability to enter text in that widget.

Motif also provides the ability to navigate between widgets by moving the keyboard focus between "tab groups" of widgets; this is accomplished by pressing a particular key (usually, of course, Tab). As normally all widgets are assigned to be part of a tab group, this effectively allows a user to move keyboard focus to every widget accepting keyboard input in an application by the use of the Tab navigation key [13, p. 172]. We feel that this focus-handling functionality would be very useful in eXene, as it would help provide a more pleasant experience for users. Although it may not be appropriate to mimic all Xlib and X toolkit functionality, we feel that a few of these features would be useful in eXene.

A top-level window may participate in the `WM_TAKE_FOCUS` window manager protocol, so that the window manager will send it a `CLIENT_TakeFocus` client message when it assigns focus to the window; in addition, any window may receive `FocusIn` and `FocusOut` events indicating that it has received or lost input focus [17, pp. 648,592]. When a top-level window receives a `CLIENT_TakeFocus` client message, it might use the `SetInputFocus` X request to reassign focus to the sub-widget that had it before focus was lost.[4] And some widgets might highlight their borders when they have input focus.

We have added a `setInputFocus` method for setting the keyboard input focus to a window. We have also modified eXene's `createSimpleTopWin` function to return a `client_msg` CML event whereby `CLIENT_TakeFocus` client messages may be read. Also, we have added the ability for eXene windows to receive `CI_FocusIn` and `CI_FocusOut` messages over their input environments.

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

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

the shell's top-level window has received a `CLIENT_DeleteWindow` message.

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

We are also in the middle of adding a focus manager to the eXene widget top-level shell (the widget-level abstraction of a top-level window).

```
signature SHELL =
sig
  ...

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

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

    type fid

    val addFocusableFirst  : shell -> focusable -> fid
    val addFocusableLast   : shell -> focusable -> fid
    val addFocusableBefore : fid * focusable -> fid
    val addFocusableAfter  : fid * focusable -> fid
    val deleteFocusable    : fid -> unit

  ...
end
```

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

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

method, may be added to the manager. A focusable object will inform the manager by means of a `focusable_msg` when input received indicates that the focus has been

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

Because the focus manager part of the shell will know which, if any, of its focusable objects currently has the focus, when it receives a `CLIENT_TakeFocus` client message, it can take appropriate action when none of its focusable objects currently have the focus. If none of its objects ever had the focus, or the last one to have the focus explicitly gave it up, then the manager can assign focus to the first of its objects. Otherwise, it can set the focus back to the object that had the focus before focus was lost. The time that is included as part of some of the focusable messages and that is passed to the `takeFocus` method is always supposed to be the time at which the user pressed/released the key or mouse button that initiated the change.[5] This time must be passed to a subsequent call of `setInputFocus`.

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

We are also building a `FocusableFrame` composite widget that wraps around a widget and its focusable object, and that draws a border around the child widget when that widget has focus. This is done by monitoring the `focusableEvt` of the focusable object. Of course, the `FocusableFrame` widget has a method that may be used to turn it into a focusable object, enabling it to be added to the focus manager of the shell.

[5]The current eXene release doesn't annotate keyboard messages with timestamps; this will have to be rectified if, e.g., Tab is to cause focus to be assigned to the next focusable object.

4 Customization using X Resources

Xlib provides for user customization of applications by means of "resource specifications". For example, an application may allow background and foreground color, window geometry, and font settings to be configured by the user [8, p. 339]. Some of these resource settings might be passed as arguments to the application on the command line, such as "`-background white`". On the other hand, some resource specifications may be general to several applications or to all instances of a given application, and these may be stored in a configuration file. Xlib provides support for both of these methods, with a `XrmParseCommand` function for loading resource settings from a list of arguments into a resource "database", and a `XrmGetFileDatabase` function for loading a resource configuration file into a resource database.

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

Finally, as resource specifications may originate from several sources (say, from command line options or the `XA_RESOURCE_MANAGER` property), application developers must have a way of combining resource databases in such a way that one database takes preference over another. Xlib provides the `XrmMergeDatabases` function for this purpose.

EXene currently provides support for user customization of widgets [3]. Widgets are passed the following resource-related information:

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

- An "args" list, consisting of a list of attribute/value pairs.

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

As noted in [3],

> The idea of the argument list used at widget creation is right, but its form is not. A flat name-value list does not allow a programmer to control the resources of specific internal widgets. The name component of an argument needs to reflect the hierarchy. This suggests replacing the type of name with a list of names ...

Our view is slightly different: although a widget may be built out of various widgets, it's not necessarily that case that this physical structure should be exposed to the application developer. But it is true that there may be various instances of a given attribute, say `font`, that a widget wants to expose. It is this logical structure that we will expose. As a result, we do plan to makes the keys of the args and attrs lists be lists of names, terminated by attributes.

EXene provides support for creating a style from a list of strings with the function `styleFromStrings`. This is sufficient, in our opinion, for loading resource specifications from a configuration file, as an application may simply read a file, then pass a list of lines to this function. However, eXene currently lacks the ability to create a style from the `XA_RESOURCE_MANAGER` property of the X server, lacks the ability to merge styles together, and has no helper functions to assist the application developer in sorting through command line arguments.

To rectify this, we have added several helper functions to the eXene `WIDGET` signature. First, `mergeStyles`, when applied to $(style_1, style_2)$, returns a style where all specifications of $style_1$ have been inserted into $style_2$, effectively giving priority to the specifications of $style_1$ (this function was trivial to write, given the previously existing support for updating styles).

```
val mergeStyles : style * style -> style
```

The `styleFromXRDB` function takes a widget root (the widget-level abstraction for an X display) as an argument, and returns a style created from the specifications in the `XA_RESOURCE_MANAGER` property of the X server.

```
val styleFromXRDB : root -> style
```

Finally, several functions have been added to facilitate sorting through command line arguments.

```
val parseCommand   : optSpec -> string list -> optDb * string list
val findNamedOpt   : optDb -> optName -> root -> attr_value list
val styleFromOptDb : root * optDb -> style
```

The function `parseCommand` takes an option specification, type `optSpec`, and a list of strings. It returns an "option database", type `optDb`, and a list of strings not recognized as arguments.

The option specification is a list of possible arguments that may be provided on the command line. Each of these individual argument specifications includes an `optName` to identify the argument (either a simple string such as "`background`" or a resource name such as "`*background`") and the argument string that identifies the option in the command line (such as "`-bg`"). Each individual argument specification also includes an option "kind", similar to Xlib's option kind furnished to `XrmParseCommand`, such as `OPT_STICKYARG` signifying that the option value will be the next string in the argument list.

Once command line arguments are organized into an option database by `parseCommand`, they may be retrieved by their `optName` tag, or by converting the options with resource names to a `style`. A value named with a string tag may be retrieved with `findNamedOpt`; an `optDb` may be converted into a `style` with `styleFromOptDb`.

5 X Authorization

While testing eXene applications, especially applications to be displayed over a tunneled SSH connection to the X server, the authors encountered a few issues with eXene's X authorization code. The file specified in a user's `XAUTHORITY` environment variable contains a series of records, each with family (Local (Unix Socket), Internet, etc.), host address, display number, authorization name, and authorization data fields. When an application is given a hostname and display number as an argument to connect to, or takes this information from the user's `DISPLAY` environment variable, this X authorization file is searched for an authorization method (and authorization data) to supply in opening the connection.

For records whose family is Internet, the address is stored as a four-byte packed IP address. It is therefore necessary to convert this IP address to a hostname for comparison with the hostname string of the argument to the application.[6] In addition, if a hostname of `""` or `"localhost"` is given as an argument, we must consider any family in the list of X authorization methods, rather than just methods with family

[6]While it is true that the IP address of the argument hostname could be obtained for comparison, the given hostname might have multiple interfaces leading to an incorrect match if IP addresses were compared.

Local. We have modified eXene's X authorization code to convert IP addresses to hostnames for comparison, and to search all authorization families when the empty string or `"localhost"` is provided as the host to connect to, thus correcting the authorization issues that we have encountered so far.

6 X Selections

The Inter-Client Communication Conventions [15] of the X window system provides two ways for X clients (applications) to share information: the selection and the cut buffer.

The selection mechanism is the more complex and general of the two methods. An application that wants to share information with other applications may acquire a selection, either the primary selection or another one, by making a request to the X server. Then, other applications may request that the value of the selection be converted to specified target types, where the set of these types is extensible, but includes different kinds of text and images. An application's request is transmitted by the X server to the owner of the selection, who can honor it (but may convert the value to a different type if it wishes) or reject it. If it honors the request, the supplied value is stored by the selection owner in a window property supplied by the requester, and the requester is notified that the value is now available. Afterward, the requester is responsible for deleting the property. There are three target types that all clients are required to handle: `TARGETS`, `MULTIPLE` and `TIMESTAMP`. `TARGETS` is used to ask the selection owner what target types it is willing to convert the value of the selection to. And, `MULTIPLE` and `TIMESTAMP` are used for batching requests and determining the time of the X event that caused the selection owner to acquire the selection, respectively. When the value in a specified type of a selection is big, it can be transmitted incrementally. Finally, there is a way of passing data to a selection owner so as to influence the value that it supplies.

In contrast, the cut buffer mechanism is very simple. It simply consists of eight properties of type string on the root window. An application can add a string to the cut buffer by shifting the values of the first seven properties to the last seven properties, and then setting the first property to that string. An application may obtain the value in the first property of the cut buffer, or may rotate the cut buffer, simultaneously moving the value in the first property to the last, and shifting the values of the last seven properties to the first seven.

At present, eXene provides only partial support for selections. If a client requests

ownership of a selection, it gets back a selection handle. From this handle, it can obtain a CML event with which it should synchronize to obtain selection requests, as well as another CML event with which it should synchronize to learn if it has lost (or never obtained) the selection. There is also a method that may be applied to the selection handle to give up ownership of the selection. EXene also provides a method for requesting the value of a selection. Currently, there is no support for the `TARGETS`, `MULTIPLE` and `TIMESTAMP` targets, or for incremental transfers, and we plan to rectify this. But we also plan to provide a higher-level interface to the selection mechanism. When acquiring ownership of a selection, one will supply a function that takes in a requested target type and returns an option value that is `NONE`, if it's not possible to convert the selection's value to the requested target, and is `SOME` v, if that value was successfully converted to the result v. If necessary, this function can communicate via CML channels to determine the current value of the selection. But more commonly, this value will be fixed, as long as the selection is owned; in such cases, we can speak of simply setting the selection to a given value. And, for common target types like `STRING`, we will provide curried functions that take in values of the given target type, and return the functions that know how to convert these values to requested types. We will provide support for passing data along with a selection request. Finally, because the property supplied along with a request for a selection is supposed to be currently unused (except when data is supplied along with the request), and because the requester is supposed to delete the property after the request is completed, we plan to hide the allocation/deallocation of the property within the eXene function for requesting the value of a selection.

We will also provide support for the cut buffer. In fact, because many applications use the cut buffer as a substitute for the primary selection, when the latter is unowned, we will provide support for doing this. I.e., we will provide support for simultaneously setting the primary selection to a string and adding the string to the cut buffer. And we will provide support for requesting the value of the primary selection as a string, and, if that fails, obtaining the head of the cut buffer instead.

7 Widget Bounds and the Layout Widget

In EXene, a widget or other graphical entity (like a box of the layout widget—see below) indicates its bounds constraints (which don't have to be honored) to its parent in two dimensions, using elements of the following datatype, which corresponds to the hints on a window's desired size that are supplied to window managers. Here,

`dim` and `DIM` stand for "dimension", `incr` stands for "increment", and `nat` stands for "natural".

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

A (horizontal or vertical) dimension

$$\texttt{DIM\{base = } b\texttt{, incr = } i\texttt{, min = } m\texttt{, nat = } n\texttt{, max = } x\texttt{\}}$$

means that the widget will be happy with a window with $b + ki$ pixels in the given dimension, where $m \leq k$ and, if $x = \texttt{SOME j}$, then $k \leq j$. But it would prefer its window to have $b + ni$ pixels, which is its natural size.

EXene provides a very general layout widget for displaying *boxes*, a recursive data structure consisting of different kinds of horizontal and vertical lists of boxes, whose leaves consist of widgets and glue. Although very useful, the existing widget produces odd results in some cases. E.g., putting inflexible glue around a flexible widget results in an overall widget that's inflexible. Part of the problem is that the existing behavior of the layout widget is largely undocumented, and its code is complex and hard to fathom. With a student collaborator, we are attempting a complete redesign of the layout widget, learning from the existing one.

We plan to start by carefully specifying how the bounds of a box is determined from the bounds of its components, and how the components of a box are sized and positioned, relative to a rectangle in which the overall box is to be rendered. As an example of the issues involved in doing this, consider a case when several boxes are put into a horizontal list. Let's restrict out attention to the horizontal dimension. How should the dimensions of the individual boxes be combined into an overall dimension for the list? (If any of the boxes are inflexible, their fixed sizes can simply be included in the base value for the list; in what follows, let's assume that all of the boxes are flexible.) If each of the component boxes has the same increment, the answer is clear. But if these boxes have different increments, it seems there is no real alternative but to fall back on setting the increment for the list to 1. Then, given a rectangle in which to display the list, it will sometimes be necessary to distribute some extra pixels between the widgets.

8 Other Widgets

K-State students have developed, or are developing, widgets for displaying images, displaying rich text, and letting a user browse the filesystem for a file. Suggestions of widget projects are welcome.

9 Documentation

We plan to develop up-to-date documentation for eXene using Reppy's MLDoc utility—which has been used to good effect for the Standard ML Basis Library manual.

Acknowledgments

We have benefited from a large number of (mostly email) conversations with John Reppy concerning SML/NJ, CML and eXene. Jonathan Hoag and the second author are currently working on a redesign of the eXene layout widget, and it's a pleasure to acknowledge Jonathan's contributions to Section 7. Thanks are also due to the following K-State students who have contributed to the local development of eXene: Jan Antolik, Dominic Gélinas, John Homer, Kevin Jones, Georg Jung, Joseph Lancaster, Jan Miksatko, Alan Reinhold, Ryan Shelton, Charles Thornton and Julie Thornton.

References

[1] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag, 1991.

[2] B. N. Freeman-Benson, J. Maloney, and A Borning. An incremental constraint solver. *Communications of the ACM*, 31(1):54–63, 1990.

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

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

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

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

[7] H. Lin. PAM: a process algebra manipulator. In *Third Workshop on Computer Aided Verification*, July 1991.

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

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

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

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

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

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

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

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

[16] M. Sarkar and M. H. Brown. Graphical fisheye views of graphs. Technical Report 84, DEC Systems Research Center, March 1992.

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

[18] A. Stoughton. Infinite pretty-printing in eXene. In *Trends in Functional Programming*, volume 3, pages 13–24. Intellect, 2002.