4

A Multi-threaded Higher-order User Interface Toolkit

EMDEN R. GANSNER and JOHN H. REPPY
AT&T Bell Laboratories

ABSTRACT

This article describes eXene, a user interface toolkit implemented in a concurrent extension of Standard ML. The design and use of eXene is inextricably woven with the presence of multiple threads and a high-level language. These features replace the object-oriented design of most toolkits, and provide a better basis for dealing with the complexities of user interfaces, especially concerning such aspects as type safety, extensibility, component reuse and the balance between the user interface and other parts of the program.

4.1 INTRODUCTION

In two previous papers [Rep86, Gan92], we have advocated an approach to the design of a foundation for graphical user interfaces and interactive applications. In our view, such a foundation includes support for concurrency, strong static typing, higher-order programming (i.e., functions as values), support for modular program construction, and automatic memory management. This article describes a multi-threaded X window system toolkit, called eXene, built on such a foundation.

The base for eXene is Concurrent ML (CML) [Rep91a], a concurrent extension of Standard ML (SML) [Mil90]. We employ CML as more than just the implementation language for eXene: it provides the semantic framework for the toolkit and permeates its design and use. This is particularly true concerning three aspects of eXene: concurrency, higher-level programming and memory management. Taken together, these aspects induce greater simplicity and a high degree of modularity in applications constructed using eXene.

The most significant characteristic of eXene is the fundamental role that concurrency plays in its design and implementation. Concurrency is critical in allowing the programmer to cleanly

User Interface Software, Edited by Bass and Dewan
© 1993 John Wiley & Sons Ltd

structure an application and its interface to handle the asynchrony and multiple contexts of interactive use. One need only consider a sample of common programming scenarios:

- A computationally intensive program that provides periodical updates to a graphical display of its status. In addition to displaying new information, the user interface must also handle external asynchronous events such as being resized by the user.
- A program for editing and analyzing multiple views of graphs, which must allow editing on one view while applying a potentially expensive layout algorithm to another.
- A language-based editor that uses incremental attribute evaluation to give the user immediate feedback about static semantic errors [Reps84]. Since a user's editing operation can result in an arbitrarily large number of attribute re-evaluations, we must structure the evaluator so that those attributes affecting the user's view will be evaluated first and those remaining will be evaluated in the background.

The majority of user interface toolkits are based on sequential languages and must emulate concurrency using event loops. While this allows features such as multiple views to be supported, it also biases the architecture of the application towards the user interface. Because the application is driven by the user interface, the event loop must be built to manage system events as well as graphics events, and computationally intensive code must be written in such a way as to divide up the work into small pieces that can be interleaved between the handling of external events. In effect, the event loop is a poor man's concurrency.

In eXene, the system architecture is not hobbled by this user interface bias. Graphical components are implemented as independent threads, separate from each other and the application code. This increases the modularity of the components and allows them and the application to be implemented more naturally, not as finite state machines. Obviously, the interleaving of computation happens automatically. This makes it simple to import code without worrying whether its execution will cripple the user interface.

The high-level language features of CML provide many of the mechanisms for creating and tailoring graphical components in eXene. Higher-order functions, parametric polymorphism, abstract event values and parameterized modules are powerful tools for building reusable, modular components. EXene promotes an applicative style programming. This increases the clarity and reliability of the code, and is especially important in a concurrent system where the possibility of interference arises. We also note that programming using eXene is type safe. Because of the complexity of building user interfaces, the safety afforded by strong static typing is too valuable to be thrown away. Most of the advantages touted for weakly or dynamically typed languages in building user interfaces are provided by the features such as polymorphism and higher-order functions mentioned above.

In a typical user interface, graphical components and system resources are heavily distributed and shared, making it difficult to determine when memory should be freed and resources released. EXene provides garbage collection and object finalization[1] to free the programmer from these decisions. As components can freely refer to other components without worrying about them disappearing, components become simpler and more modular.

It is possible to address the problems solved by eXene, or provide similar features, using conventional languages, libraries and toolkits. This, however, typically involves making the application code more complex, bending or breaking the type system, relying on programmer

[1] In *object finalization*, a value can be associated with a finalization function to be called on the value before the value's memory is freed. With this mechanism, we can extend the model of automatic storage collection to system objects such as bitmaps, fonts, etc.

discipline, or ignoring the problem. The thrust of the work on eXene is that, by building a user interface toolkit on top of a concurrent, high-level foundation, one achieves a system that is simpler, safer, more uniform, and more modular.

4.1.1 Related Work

CML and eXene are actually second generation systems emerging from these ideas. They follow from our earlier work with the Pegasus system [Rep86, Nor87, Rep88, Gan92], which used the PML language for its foundation. The work most closely related to Pegasus and eXene in spirit is Newsqueak [Pik89b, Pik89a] and Montage [Haa90]. EXene differs in being a more fully developed system, with a higher-level model of concurrency (CML events values), a richer graphics model, more support for programming at the component level, and, in comparison with Newsqueak, a richer base language. The NeWS window system [Gos89] also relies on concurrency in its design. NeWS requires that the user interface code be split between the client (typically C code), and the display server (PostScript code). The PostScript code running on the server can be multi-threaded. While this allows concurrency in interactive applications, exploiting it increases the complexity of the client-server interaction.

User interface design seems to be one area where an “object-oriented” approach has a clear utility. As a result, most graphics toolkits use an object-oriented approach. Examples include InterViews [Lin89], Xt [Nye90b], NeWS, Iris [Gan88], Trestle [Man91] and Garnet [Mye90]. EXene eschews an explicit object-oriented approach for several reasons. Structural polymorphism, first-class function values and a sophisticated module system provide similar interface inheritance. Most importantly, our experience, and that of others [Pik89a, Haa90], suggests that concurrency and delegation provide many of the same advantages as object-oriented programming.[2] Threads provide localization of state and clean well-defined interfaces. Delegation and wrapper functions provide implementation inheritance.

In general, some form of concurrency is available or could be added to any of these toolkits. Treating concurrency as an afterthought, however, prevents possibilities for simplification and component sharing.[3] We feel that concurrency should be a design principle, and be exploited at all levels of an interactive system.

4.1.2 Summary of the Paper

In the next section, we briefly describe the important features of CML. This is followed by Sections 4.3–4.5, which respectively describe drawing, user interaction and widgets in eXene. Section 4.6 presents some examples of eXene applications. We conclude with some future directions. Throughout the article, we assume that the reader has some familiarity with the X window system and its terminology as can be found, for example, in [Sch92]. Although we use a small amount of SML and CML notation, knowledge of these languages is not necessary in understanding the important concepts in the paper.

[2] This should not come as a surprise, since *delegation* was originally a concept of concurrent *actor* systems.
[3] This is also true for garbage collection.

4.2 AN OVERVIEW OF CML

Both the implementation and the user's view of eXene rely heavily on the concurrency model provided by CML.[4] CML is based on the sequential language SML [Mil90] and inherits the useful features of SML: functions as first-class values, strong static typing, polymorphism, datatypes and pattern matching, lexical scoping, exception handling and a state-of-the-art module facility. An introduction to SML can be found in [Pau91] or [Har86]. The sequential performance of CML benefits from the quality of the SML/NJ compiler [App87]. In addition CML has the following properties:

- CML provides a high-level model of concurrency with dynamic creation of threads and typed channels, and *rendezvous* communication. This distributed-memory model fits well with the mostly applicative style of SML.
- CML is a higher-order concurrent language. Just as SML supports functions as first-class values, CML supports synchronous operations as first-class values [Rep88, Rep91a, Rep92]. These values, called *events*, provide the tools for building new synchronization abstractions. This is the most significant characteristic of CML.
- CML provides integrated I/O support. Potentially blocking I/O operations, such as reading from an input stream, are full-fledged synchronous operations. Low-level support is also provided, from which distributed communication abstractions can be constructed.
- CML provides automatic reclamation of threads and channels, once they become inaccessible. This permits a technique of speculative communication, which is not possible in other threads packages.
- CML uses preemptive scheduling. To guarantee interactive responsiveness, a single thread cannot be allowed to monopolize the processor. Preemption insures that a context switch will occur at regular intervals, which allows "off-the-shelf" code to be incorporated in a concurrent thread without destroying interactive responsiveness.
- CML is efficient. Thread creation, thread switching and message passing are very efficient (benchmarks results are reported in [Rep92]). Experience with eXene has shown that CML is a viable language for implementing interactive systems.
- CML is portable. It is written in SML and runs on essentially every system supported by SML/NJ (currently seven different architectures and many different operating systems).
- CML has a formal foundation. Following the tradition of SML [Mil90, Mil91], a formal semantics has been developed for the concurrency primitives of CML (see [Rep91b] or [Rep92]).

4.3 BASIC EXENE FEATURES

Before we describe the more radical features of eXene, a discussion of some of the basic features is in order. These features for the most part follow the Xlib model, but we have attempted to provide an interface that is both cleaner, and more in keeping with the SML programming style. For example, we use immutable objects where possible (such as immutable *tiles* for specifying textures, instead of pixmaps), and we perform more client-side error checking, instead of relying on the X server for error checking.

[4] Conversely, the development of CML was strongly motivated by the desire to be able to support user interface systems comparable to eXene.

Following the X model, eXene supports the notions of a *display* (a connection to a server), a *screen* (a monitor driven by the server), and a *window* on a screen. An application might have multiple displays (e.g., a multi-player game), multiple screens for a given display, and multiple windows per screen. In Xlib, programmers must specify the display as an argument to most operations; in eXene, we avoid this by incorporating the display in the representation of most graphical objects (e.g., screens and windows).

Internally, eXene uses a small collection of threads to implement each display connection. These threads manage buffering and sequencing of communications with the X server. The interface to these threads is a collection of CML channels that are bundled into an abstract display value. A similar scheme is used for screens. Since these abstract values encapsulate both the state and thread of control, supporting multiple displays and screens in an application is trivial.[5]

EXene uses a cleaner, and slightly stripped-down, version of the X graphics model [Sch92]. In X, drawing operations are performed with respect to a *graphics context*, which is a server-side object that specifies the color, font, texture, etc., of the drawing operation. There are a number of drawbacks to the way that graphics contexts are supported: they are fairly heavy-weight, requiring communication with the server to create and update; the number of contexts supported by the server may be limited (e.g., if the server is an X terminal); and they are created with respect to a particular visual, and can only be used with that visual. Furthermore, the Xlib interface to graphics contexts is not uniform: the clipping mask is part of the context, but is specified independently of the other attributes.[6] Another wart is that fonts are specified both as part of the graphics context, and as an argument to the `XDrawText` operation. Not only does `XDrawText` take fonts in its argument list, it actually updates the graphics context that it uses, which makes sharing of graphics contexts more difficult. In addition to these problems, multi-threaded toolkits face the additional problem that graphics contexts are shared mutable objects, thus some form of concurrency control is required.

In eXene, we address these problems by providing a higher-level, but lighter-weight, client-side value, called a *pen*, for specifying drawing attributes. Pens provide a cleaner, more uniform interface to specifying the attributes of a drawing operation: they are immutable, they are independent of any visual, and they collect together all the drawing attributes uniformly. Pens include the clipping region, but not fonts, which are specified as an argument to the text drawing operations. Internally, concurrency control is provided by graphics-context servers (one per visual), which map pens to server-side graphics contexts. Since pens are lightweight, immutable and independent of any visual, the user is freed from having to manage graphics contexts to reduce server memory usage: the toolkit does it for her. The use of a behind-the-scenes manager allows better modularity in the application code. While Xt also provides management of graphics contexts to promote sharing, its mechanism is weaker: it only supports the read-only use of shared graphics contexts, and does not provide security against interference.

To further improve modularity, eXene supports automatic reclamation of server objects such as fonts, colors, and pixmaps. This is implemented by a finalization scheme built on top of SML/NJ's *weak pointer* mechanism. When the client-side version of the object becomes

[5] This encapsulation property appears elsewhere in eXene (cf., Section 4.5), and can be exploited to easily support multiple views in an application.

[6] This is because the arguments to the graphics context operations are integers, while a clipping region is represented as a list of rectangles.

garbage, a finalization procedure is invoked, which sends a request to deallocate the server-side object. This frees the user from explicit management of these resources.

The use of internal servers to multiplex server-side resources is exploited repeatedly in eXene. Another example arises in the handling of fonts. There is a per-display font server that maintains a table of open fonts, and checks requests for new fonts against the table. This reduces the amount of client-server traffic (e.g., if ten independent button widgets all attempt to open the same font). Although not part of Xlib, a similar optimization is frequently used in toolkits built on the library.

Another place in which eXene provides a higher-level model is in the support for color.[7] Colors are specified abstractly using color names or RGB values (we plan to extend this to the X11R5 device independent color spaces [Sch92]). Color specifications are mapped to abstract colors by a color server (one per screen), again reducing client-server traffic and improving modularity. Although inappropriate for color-intensive applications,[8] this model, used with finalization, is adequate for most uses of color.

EXene provides an abstract *drawable* type that gives a common interface to windows and off-screen pixmaps. Internally, a drawable is represented as a connection to a *draw-master* server that buffers drawing operations and handles the interaction with the graphics-context server. This representation allows filters to be interposed that modify the drawable's behavior (e.g., coordinate translations).

The client-server communication in X is asynchronous, and this often shows in the programming model. Ideally, drawing operations should not require exposing the asynchrony of the protocol, but there are a few places where this breaks down. In eXene, we have attempted to hide this asynchrony wherever possible, and provide a higher-level interface in those cases where, for performance reasons, the asynchrony must be exposed.

Because it requires a system call to send a message to the X server, Xlib buffers client requests so that the system call overhead is reduced. One of the most common mistakes made by the neophyte Xlib programmer is failing to flush the buffer after a sequence of drawing operations; without this, the graphics never appear on the screen. In a program structured around a central event loop (or one using the built-in event loop provided by many toolkits), this is not as big of a problem, since the output buffer is flushed prior to reading the next X event. As we noted in Section 4.1, however, structuring programs around a central event loop is not always desirable. Our approach in eXene is to have the output-buffer thread periodically flush itself out to the wire. This removes the need for the user to explicitly flush the buffer, which leads to better modularity. Unfortunately, this periodic buffer flushing does not provide fast enough turn around when real-time feedback is required (e.g., when using the mouse to adjust a scrollbar). To handle this problem, we provide a function for creating a *feedback* drawable from a drawable. The feedback drawable uses the same drawing surface, but is unbuffered for immediate visual feedback. A better solution may be to tie the mouse stream and drawable together in these situations, such that the output buffer is flushed whenever the client requests input.[9]

Another feature of the X protocol that often bites users is a race condition between the client's first drawing operation and when the server actually maps the window [Gaj90]. To

[7] We currently only provide access to the default read-only colormap of a screen.

[8] Since we do not expose the notion of *pixels*, applications are prevented from exploiting color-plane tricks, which may not be a bad thing.

[9] This was suggested by one of the referees.

avoid this race, an application must wait for the first exposure event[10] on the window before attempting to draw graphics. In eXene, we hide this required synchronization internally.

A third place in which the user is exposed to the asynchrony of the protocol in graphics operations is the interface to the `CopyArea` operation. When `CopyArea` is used to copy a rectangle of pixels from an on-screen window to some destination (e.g., when scrolling), it is possible that a portion of the source rectangle will be unavailable (i.e., because it is obscured by another window). In this case, the corresponding portion of the destination rectangle will have to be repainted. Conceptually, this can be viewed as a client request followed by a server reply, but if implemented this way, the round-trip delays cause noticeable screen flicker. What is needed is an asynchronous remote procedure call (RPC), sometimes called a *promise* [Lis88]. Providing an asynchronous RPC interface to the `CopyArea` operation is not possible in a language like C, so this is implemented by using X events to deliver the reply. The client receives either a `GraphicsExposure` event or a `NoExpose` event as an acknowledgment of a `CopyArea` operation. In eXene, we exploit the first-class synchronous operations provided by CML to provide a true asynchronous RPC interface to the `CopyArea` operation. When a client executes a `CopyArea` operation, it receives a CML event value that is the promise of a list of exposure rectangles (the empty list signifies `NoExpose`). The client can proceed with drawing, and later synchronize on the event to check for any needed repairs (see [Gan91] or [Rep92] for more details).

4.4 USER INTERACTION

The most significant departure in eXene from traditional user interface toolkits is in our approach to handling user input. As we argued in [Gan92], the multiplexing required by graphical applications maps naturally onto a concurrent programming model. Instead of a centralized event loop for processing user input, eXene uses a distributed hierarchy of threads to route user input to the appropriate place. The hierarchy basically mirrors the window hierarchy of the application. Each component in the hierarchy has an *environment*, consisting of three streams of input from the component's parent (mouse, keyboard and control),[11] and one output stream for requesting services from the component's parent. For each child of the component, there are corresponding output streams and an input stream. A component is responsible for routing messages to its children, but this can almost always be done using a generic router function provided by eXene. This event-handling model, with its top-down decentralized routing, is similar to those of [Pik89a] and [Haa90]. It is substantially different from the bottom-up approach used by most toolkits. In particular, it allows parents to interpose filters on the event streams of their children, which is an important mechanism in the composition of widgets in eXene (cf., Section 4.5.3).

To illustrate this approach, consider the implementation of a simple drawing application in eXene. The application presents the user with a window for drawing and a reset button (see Figure 4.1). When the user depresses a mouse button in the drawing window, a triangle is drawn at the cursor location, and when the user clicks on the reset button, the drawing window is cleared. This application's implementation consists of three components: a top-level component with two subcomponents (one for the button, and one for the drawing window). Each component has an associated X window. Figure 4.2 gives the thread network for these

[10] An exposure event notifies the application that a region of its windows has been damaged and requires repainting.
[11] Note that we translate X events into the appropriate types of eXene messages.

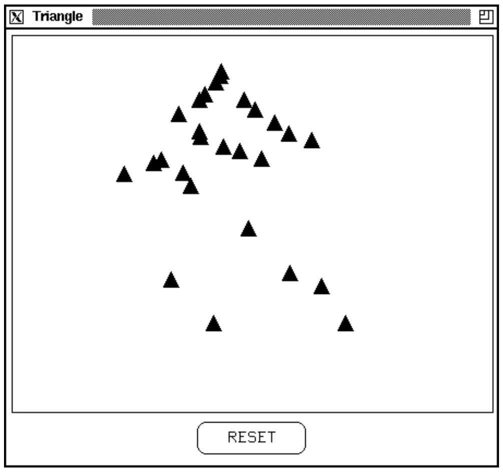


Figure 4.1 A simple drawing application

components; to simplify the picture we have omitted the output streams for requesting services, since they are not used by the application. The small unlabeled squares in this picture represent message sinks (threads that just consume input forever). For example, the top-level component ignores mouse and keyboard messages and has no graphics to redraw, so it has sinks for all three of its input streams. In addition to the sinks, the top-level component has a *router* thread associated with it. The router thread looks at the addresses of incoming messages and forwards them to the appropriate destination.

The drawing component has three threads: a sink for keyboard messages, a mouse-stream thread, and a command-stream thread that also maintains the drawing state. The mouse-stream thread looks for button-press messages, discarding all others. When a button-press message is received, it sends a drawing command to the command-stream thread. The command-stream thread is responsible for drawing the triangles on the drawing window, and for redrawing the window if it is damaged; the code for this thread is given in Figure 4.3. The thread is implemented as a tail recursive function. It receives messages from three sources: command messages from the top-level component's router (`cmdEvt`); draw messages from the mouse thread (`drawEvt`); and reset messages from the button component (`resetEvt`). For each source there is a corresponding handler function (`handleCmd`, `draw`, and `reset`). The thread also maintains a state, consisting of a list of the points on the display where triangles have been drawn. The function `drawTriangle` (not shown) draws a triangle at the specified point; it is used both to draw new triangles (in `draw`), and to redraw the screen to repair damage (in `handleCmd`).

The button component also consists of three threads: a sink for keyboard messages, a mouse-stream thread, and a command-stream thread. The mouse-stream thread looks for button-press messages; when it receives one, it sends a reset message to the drawing component's command-stream thread. The button's command-stream thread is responsible for redrawing the button when it is damaged.

When the user clicks the mouse on the button, the X server sends an X event to the application; the eXene library code routes this to the top-level component's mouse stream as

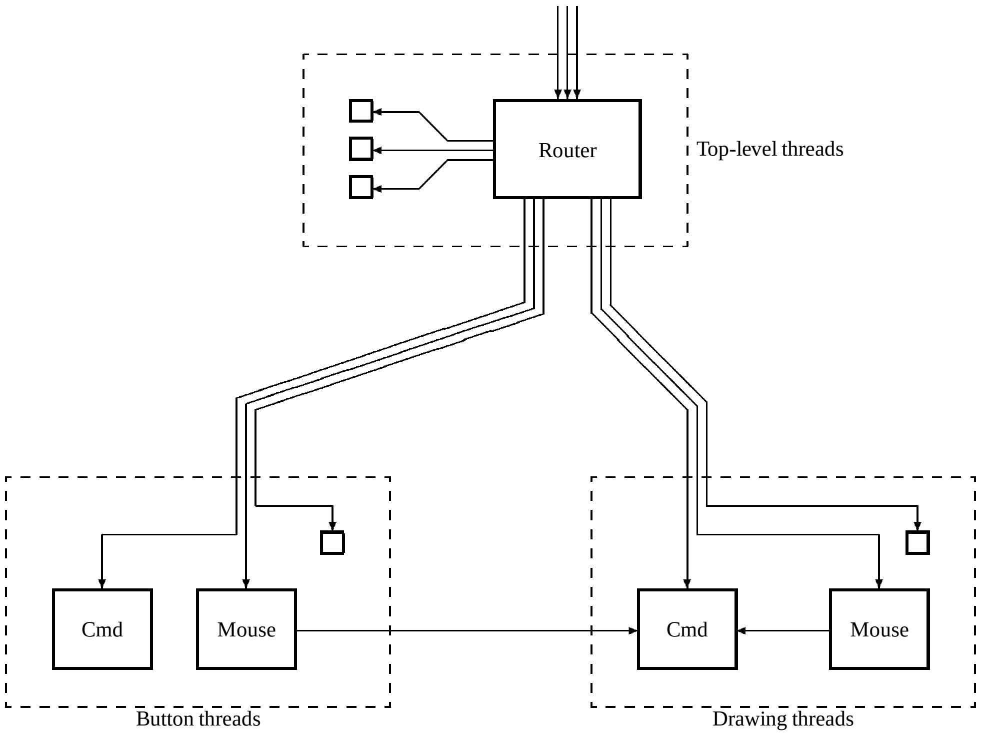


Figure 4.2 The application's communication network

a mouse message addressed to the button component. The router in the top-level component then passes the message on to the button component, which then sends a reset message to the drawing component.

The use of several threads per component, typically one for each input stream as well as one or more threads for managing state and coordinating the other threads, is standard in eXene applications. By breaking the code up this way, each individual thread is quite simple. This heavy use of threads is made possible by the lightweight nature of threads in CML. A thread typically incurs less than 100 bytes of space overhead [Rep91a], which makes them comparable in size to Smalltalk objects [Ung84].

4.5 EXENE INTRINSICS

The basic eXene features described above do not provide a general framework in which pieces of an interface can be built by various people at various times and then modified and integrated into a single user interface. Support for this is provided by the widget[12] layer in eXene. Widgets are the basic building blocks for constructing interfaces. The widget layer

[12] For want of a better term, we borrow the X term for a graphical object composed of a drawing area and its interaction semantics.

```
fun drawCmdLoop state = let
      fun handleCmd (CI_Redraw _) = (
            clearDrawable drawable;
            app drawTriangle state;
            drawCmdLoop state)
        | handleCmd CI_OwnDeath = ()
        | handleCmd _ = (drawCmdLoop state)
      fun draw pt = (drawTriangle pt; drawCmdLoop(pt::state))
      fun reset () = (clearDrawable drawable; drawCmdLoop[])
      in
        select [
            wrap (cmdEvt, handleCmd),
            wrap (drawEvt, draw),
            wrap (resetEvt, reset)
          ]
      end
```

Figure 4.3 The drawing command thread code

also provides the additional protocols necessary for cooperation among widgets, as well as their reuse and extension.

The widget level reifies the underlying eXene approach to building graphical interfaces.

- The inherent concurrency in the user interface is made explicit. The user interface is just a part of an application: it does not dictate the architecture or control structure. Interfaces are built as networks of simple components connected by streams and event values. Each widget has its own threads, which separate it from other widgets and from the application code. A programmer can also use concurrency to simplify the internal structure of widgets, as described in the previous section.
- Few things are as full of state as graphical objects. With threads, the function call structure naturally encodes much state information, without the programmer having to maintain state explicitly. Additional state is encapsulated in channels.
- Input is distributed hierarchically. Events are passed from the root widget down the hierarchy to the appropriate widget. This allows the programmer to interpose widgets at any level to modify widget characteristics or alter the distribution of events.
- Higher-order functions, parametric polymorphism and parametric modules powerfully expand the programmer's tools for tailoring and combining interface components safely and simply.

As with most interface toolkits, a program using eXene creates at runtime a variety of widgets, which are combined into one or more hierarchical structures. At some point in the program, these structures are made visible and active. With most toolkits, after the widget hierarchies have been instantiated, the program gives up control to an event loop supplied by the toolkit. Alternatively, an application must be willing to provide its own event distribution mechanism. In eXene, however, the program can continue to go about its business, whether performing computations, reading input or interacting with the widgets. As an example, Figure 4.4 contains the code for a simple program that uses widgets. This example creates a button labeled "`Goodbye, Cruel World!`". If the user clicks on the button with any mouse button, then the function `quit` is called, terminating the program. The program will

```
fun goodbye display = let
      val root = mkRoot display
      fun quit () = (delRoot root; RunCML.shutdown())
      val quitButton = mkCmdButton root {
                         label="Goodbye, Cruel World!",
                         action=quit
                       }
      val shell = mkShell root (
                    quitButton,
                    NONE,
                    {win_name = NONE, icon_name = NONE}
                  )
      fun loop () =
            if input_line std_in = "quit\n"
              then quit ()
              else loop ()
      in
        init shell;    (* make button visible *)
        loop ()
      end
```

Figure 4.4 Goodbye

also quit if the user enters “`quit`” on standard input. This is similar to the `xgoodbye` example in Chapter 2 of [Nye90b], but differs in one major way. In the `xgoodbye` example, control is passed off to the Xt event loop; in our version, the application retains control.

4.5.1 Widgets in EXene

As in most toolkits, a widget is a graphical object that corresponds to some control or feedback element of the user interface. But in eXene, widgets have a very “thin” interface. In addition, information is mostly distributed, with little that is global or centralized. This gives eXene widgets a very distinctive flavor. A widget only knows about the window it was handed and how it divided the window among its children. The parent widget[13] controls the external view of a child. The parent provides the child’s window; it positions the window; it changes its size; it deletes it. If the child needs any of these actions performed, it asks the parent to perform the action. A child does not directly alter the external configuration of its window; it should only deal with what is inside its window.

A widget in eXene has three important attributes: a `root`, a `boundsOf` function, and a `realize` function. The `root` value corresponds to the screen on which a widget lives. A widget uses the `boundsOf` function to specify its size constraints. This function is usually used by the widget’s parent when determining how much display space to allocate for the widget. The `bounds_t` type provides a fairly general mechanism for specifying geometry requirements in terms of natural size, increments and upper and lower bounds.

At the time of instantiation, a widget is passed, through its `realize` function, a window

[13] A *parent* widget is just the widget that provides the child’s window and event streams. The parent widget need not correspond to a parent window of the child widget’s window. In particular, a parent and child widget may share the same window.

on which to draw, the size of the window, and an input environment. This environment was described in Section 4.4. It is the widget's only built-in connection to the outside world. In addition to providing streams for the widget's mouse and keyboard events, it also supplies control channels for service requests and status events between parent and child. From its parent, a child receives notification that its window size has changed, that its window has been damaged and needs repair, and that its window has been deleted, among others messages. In turn, the child can use a control channel to request that its parent delete its window or reallocate space for it.

When its `realize` function is called, the widget configures itself corresponding to the given size and arranges to service events on its input environment. If the widget has any children, it must also layout its children, provide their windows, and call their `realize` functions. A parent widget is responsible for distributing the user events it receives to the appropriate child widget, if any, and monitoring requests from its children.

In addition to specifying how widgets interact and communicate, the widget layer provides mechanisms for writing and tailoring widgets. In eXene, we have taken the approach that one should construct small, lightweight components, with well-defined functions, and use a collection of powerful techniques to combine them to create a widget with the desired properties. These techniques can be roughly divided into two categories, as discussed in the following two sections.

4.5.2 Parameterization

The principal mechanism for specializing values in any library is parameterization. The library designer provides various hooks by which the programmer or user can tailor the library components to a particular use or appearance. In this regard, eXene is no different, though its design provides some atypical approaches to parameterization.

When created, a widget accesses a style database to determine various values affecting its display or its actions. These values typically involve fonts, background and foreground colors, layout parameters, and event actions, and are set to reflect the needs of the application or the preference of the user. EXene use a naming scheme for widgets and widget resources based on a logical hierarchy. As noted by Gettys [Get91] and others, there are problems with the standard naming system based on the physical widget hierarchy because it exposes too much of the underlying structure. To protect against changes to the widget structure, users would end up relying on loose bindings in resource specifications almost to the exclusion of strict bindings. Logical naming is more robust, supporting major changes to the widget hierarchy with no effect on resource naming, and more flexible, allowing resource names to diverge from the physical hierarchy. The logical naming scheme is also necessary in eXene as the widget hierarchy is created bottom up. When necessary, the application program can arrange that its choices override any user settings.

Function values form one of the most important classes of parameters used to tailor widgets. They are typically used by the programmer to specify action or callback functions that are invoked by the widget in response to some event or condition. Frequently, the application requires these functions to be executed in the context of certain values outside the purview of the widget. In addition, different widgets invoke callback functions with different types of arguments. With first-class function values available, the eXene programmer can use callback functions without subverting the type system or introducing an onerous multiplicity of types whose sole purpose is to mimic function closures.

There are other problems typically associated with callback functions as they occur in standard libraries. A client must explicitly register callback functions with widgets; it must explicitly unregister the functions when it no longer wishes the services performed. The code for managing a collection of callback functions is built into the widget. From the client's standpoint, its callback functions are invoked asynchronously. The client code must be able to accept the effects of a callback at any time. In addition, the callback function cannot involve time-consuming operations, lest it block the widget from handling other clients and potentially lock up the entire application.

In eXene, we use CML events to provide the means of communication between widgets[14]. A widget's interface can contain an abstract event value. There is no registering or unregistering of callback functions; the client just synchronizes on the event value. When the appropriate conditions occur, the widget synchronizes on the event value. The client then has the opportunity of continuing other activities until it reaches a state at which an action associated with the event is appropriate. The action executes in the context of the client's data and, thanks to the ability to spawn threads, can involve extensive computation. These semantics scale nicely to multiple clients by the interposition of a multicast channel between the widget and its clients. No change to the widget is necessary.

Widgets in eXene are often built out of even lighter weight components, offering another opportunity for parameterization. A common form of decomposition reflects a separation of the view and control aspects of a widget, somewhat in analogy to the *Model-View-Controller* idea [Kra88]. As an example, the eXene widget library supplies a `mkLabelView` function that, given a font, a string and an alignment, returns the size required to display the string plus a draw function. The draw function takes a drawable, a rectangle and a pen, and draws the text with the specified alignment within the rectangle in the drawable. This higher-order function is employed many places where the display of a text string is required. Another place where this technique arises is in the construction of button-like objects. The library defines a protocol for button views, corresponding to various possible button states, such as being active and set. There is a collection of button views following this protocol, such as check boxes, text buttons, and rocker switches. EXene also supplies various controller functions that implement some form of user interaction and, given a button view, use the view for visual feedback of the control state. A programmer creates a button-like object in eXene by composing a button behavior with a button view. For example, Figure 4.5 shows functions providing two of the standard button behaviors, one delivering a continuous stream of events while the button is "pressed" (mkButton) and one maintaining an on/off state for the button (mkToggle). The figure also shows functions implementing two of the standard button views, along with examples of the views. The lines indicate the four possible ways of composing the functions to get four different types of buttons. Views and controllers can either be supplied by the programmer, or taken from the eXene library.

Certain widgets are most generally parameterized over multiple types and values, more than can be easily accommodated by using SML's parametric polymorphism and function values. In these cases, the eXene programmer can use SML's parameterized modules, called *functors*. In the SML module system, functors are written as functions that create new modules when applied to a collection of modules compatible with certain specified signatures. This is precisely what is needed to factor out the dependencies of a widget over a complex structure of interrelated types and values. For example, the eXene text widget is written as a functor taking

[14] With the availability of threads and synchronization primitives, the concurrency problems with callbacks disappear. One simply writes a callback function to spawn a new thread or to synchronize on a CML event.

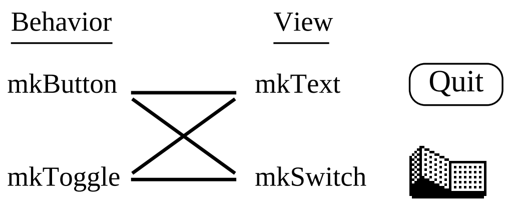


Figure 4.5 Composing buttons

a text buffer module as an argument. Such a module provides an abstraction for creating a two-dimensional layout of text. This abstraction includes a submodule that defines a typeball type, which is used to specify how (font face and size, color, spacing) a text string should be drawn. The role of the text widget is to provide controlled access to creating and modifying a text buffer while maintaining a projection of the buffer layout on the screen.

EXene also uses functors to parameterize viewport widgets, which allow the programmer to create a changeable “window” on a (potentially) much larger drawing area. The functor parameters are used to specify the constraints on the viewport window, how the drawing area coordinates project onto the viewport window, and how a view is implemented. For a pixel-based drawing canvas, a viewport may be an unconstrained projection of the drawing. For a text-based widget, the viewport is probably constrained by line or character boundaries. A simple view can be created by having the drawing area implemented as a subwindow of the viewport window. The programmer can construct more efficient views by using direct calls to the canvas drawing routines, circumventing the window system, and by maintaining drawing caches such as backing store, text buffers, or display lists.

4.5.3 Widget Wrappers

The hierarchical routing of events in eXene allows the programmer to wrap one widget within another, thereby allowing the wrapping widget to interpose its behavior between the wrapped widget and a prospective parent, essentially producing a derived widget. The wrapping widget subverts the wrapped widget’s interface, providing a new `boundsOf` or `realize` function or intercepting the event stream. In a simple case, the wrapping function might do nothing more than translate keystrokes. This technique is also used to fix the size of a widget, widgets typically being written to adjust to whatever size they are given. The programmer wraps a widget in another widget, whose `boundsOf` function returns a fixed size, with equal lower and upper bounds. The text widget described above is an output-only widget: it does not respond to any mouse or keyboard events, but it handles redisplay correctly. To make a text editor or a virtual terminal, a programmer would wrap the text widget with code to handle user input, using the text widget as a simple output device. As another example of this approach, reconsider the application described in Section 4.4. If the programmer decides to attach a pop-up menu to program, she can wrap the top-level component with a function that responds to mouse button presses by putting up the menu but forwards all other events down the hierarchy.

Graphical composition occurs when a widget’s window is a subwindow of that of its parent widget. This technique, standard in all interface toolkits, is also handled by widget wrapping. Thus, in eXene, a single mechanism is used to support subwindowing and derived widgets.

Because of the desire that basic components should be as simple as possible, eXene, like InterViews, relies on composition for features that might ordinarily be built into a widget. For example, to get a border for a widget, one inserts a widget into a frame widget, whose role is to create a border about its child widget.

The design of eXene induces an elegance in the use of composition. For example, one can create a new widget giving textual feedback on a slider widget[15] by composing the slider with a text label widget in a box layout widget, and spawning a thread that monitors changes to the slider's value, as captured by a CML event, and resets the label widget's text accordingly.

4.6 APPLICATIONS

EXene has served as the basis for a variety of applications, several of significant size and sophistication. We describe some of these uses below.

As a basic "proof of concept," we have ported a number of standard sample graphics applications, such as a hand calculator (Chapter 12 of [Nye90a]) and a bitmap editor (Chapter 4 of [Nye90b]), to eXene. Typically, the eXene versions require a third to a quarter of the code needed for the Xt versions, largely because of the expressiveness, concurrency and memory management provided by the underlying language.

Video game-like programs make good test cases for graphics toolkits. We mention three that show the utility and efficiency of threads in eXene. In *bounce*, the user uses the left mouse button to send a new ball bouncing around within a window. If the user clicks the middle mouse button on a ball, it disappears. The right mouse button brings up a pop-up menu that allows the user to start over, or to quit. Because of the distributed user input handling, the balls continue to move while the menu is being displayed. In the implementation, each ball has its own thread, which calculates the next position and passes this information to a display manager thread. Another application using animation-like graphics is an arithmetic teacher. The game has buttons for picking the operation (addition, subtraction, etc.) and the level of difficulty, and a window for displaying the problem and the tentative answer. Another window displays a scene in which a small figure climbs some distance up a pole on each correct answer. With enough correct answers, the figure makes it to the top and waves a flag; a wrong answer sends the figure into a pool of water with a big splash. With concurrency, the figure can still be climbing while the user is working on the next problem. We have also ported Cardelli's *badbricks* game to eXene.[16] The notable point here is that a typical game involves on the order of 1600 threads.

Other sample eXene applications include an implementation of a *Graphical Fisheye Viewer* [Sar92], and an implementation of the *DeltaBlue* incremental constraint solving algorithm [Fre90]. The latter is an interesting experiment because of the importance of incremental constraint solving in high-level user interface management systems. This experiment suggests that constraints may provide a high-level mechanism for specifying the interconnection of widgets [Yan92].

EXene has been used to provide the user interfaces for two interactive theorem proving systems. PAM [Lin91] is a general proof tool for process algebras. Its interface consists of a main window, which is used to compile process algebra calculi and create proofs, plus a window for each proof. At the same time, the user can work on several problems in the same

[15] A slider widget provides a valuator by which the user can set a scalar value in some range.
[16] Badbricks is a demo included in the Trestle distribution.

calculus, or the same problem in different calculi. Proofs complete in one proof window can be bound as a named theorem and then used in other proofs. The system developed by Griffin and Moten [Gri92] provides as a library a logic-independent implementation of tactic trees. It is designed to be incorporated into any SML-based interactive theorem prover. It provides a structure editor for tactic trees, in which the user can move the focus to any subtree, alter the view of the tree to global, local or elided views, and modify a subtree by deletion or the application of a tactic.

4.6.1 Graph-o-matica

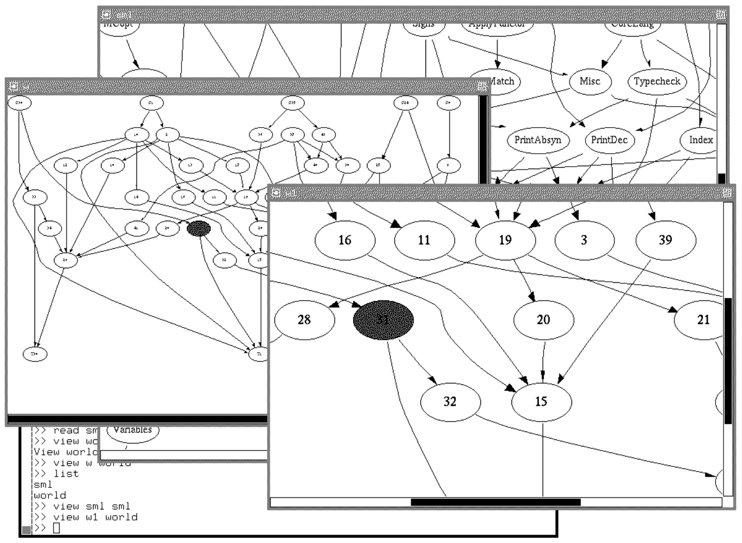
In order to discuss in more detail how the various features of eXene come together in an application, we focus on a single application, called *Graph-o-matica*, in the remainder of this section. Graph-o-matica is an interactive tool for analyzing and viewing graphs, which has been implemented on top of eXene.

There are two main types of windows used in Graph-o-matica: *command windows*, which provide a terminal-style, textual interface for manipulating graphs and their views, and *view windows*, which provide a view on a 2D layout of a graph. There can be multiple command windows, a given abstract graph can have more than one layout, and a given layout can have more than one view. A layout allows the user to modify the 2D embedding of the graph, including elision of subgraphs. A view allows the user to pan and zoom (using menus and the scrollbars) on a given embedding of the graph. Dialogue boxes and other standard graphics paraphernalia are employed in the interface. Except for the graph drawing component, Graph-o-matica uses standard eXene widgets.

Figure 4.6 illustrates a sample session using Graph-o-matica. The bottom window is a command window. Future versions might include more graphically oriented mechanisms for some of the main types of operations; however, the command window will probably always be available to the user. As a general rule, a program should have an underlying text language interface, even when the principal user interactions are graphical. Lying on top of the command window is a view of the graph of modules in the SML/NJ compiler. The top two windows provide two views of a different graph but each view uses the same layout. If a graph is edited, either using a graphical view or from a command window, this information needs to be propagated to the layouts and views of the graph. The system uses a multicast channel abstraction to manage the propagation of update notifications between graphs and layouts and between layouts and views. This simplifies the implementation of the graph object, since it does not need to know anything about multiple layouts. The layout objects, if they decide a given change affects them, can query the graph object for more detailed information. Similar simplification occurs in the layout objects. The modular style supported in eXene, based particularly on explicit, abstract concurrency and high-level memory management, facilitates this type of layering and multiplication of objects.

The need for both communication abstraction and selective communication also arises in the *virtual terminal* used in the command window. At any time, the virtual terminal must be able to handle both input from the user and output from its client (the command shell). EXene provides an abstract interface to the input stream, but since it is event-valued, it can still be used in selective communication.

Because the pieces are written to operate independently, using their own threads, the user interface need not block. For example, when Graph-o-matica presents a dialogue box to the user it may continue computation, including drawing graphics and handling user input, while



```
>> read sm
>> view wo
View worl
>> view w world
>> list
sml
world
>> view sml sml
>> view w1 world
>> 
```

Figure 4.6 Graph-o-matica

the dialogue box is displayed. Internally, selective communication is used by components to multiplex waiting for the result of the dialogue box with other interactions. Similarly, Graph-o-matica avoids having intensive computation, such as running the graph layout algorithm, lock up the system. These computations are run in separate threads, with completion marked by a CML event.

It should be emphasized again that similar asynchronous behavior can be achieved in many standard toolkits. Typically, an event loop plays the role of a scheduler, and the application can register and unregister functions to be called in response to user events, file system events, timer events or when no events need processing. This introduces the user interface bias discussed above. It becomes necessary to structure the application, including non-interface code, as a state machine. This approach is cumbersome and introduces distortions in the structure of the software. Imagine being required to write a potentially time-consuming algorithm that can only run in quanta of "small fraction[s] of a second" ([Nye90b], p. 239) before returning. As a result, applications written using standard toolkits tend not to provide the level of concurrency users expect.

The approach we have taken in eXene is to make the presence of concurrency explicit, and to build the interface toolkit to take advantage of this concurrency. This allows the pieces of the application to be written in whatever style is appropriate, provides better separation of the components of the application, and makes it simple and inexpensive to use concurrency when required.

4.7 FUTURE WORK

Although eXene is quite usable in its current state, it is still very much a work in progress. We are already planning various changes, some at the implementation level, others providing enhancements to the user's view.

The hierarchical routing used in eXene provides the basis for the programmer's ability to wrap an old component in a function providing new behavior. Most of the time, though, events are routed through most paths unchanged. We would like to explore means of maintaining the semantics of hierarchical routing while providing more efficient direct routing when possible.

At present, eXene provides no facility by which a widget can specify interesting mouse events. It is possible that something akin to cages in Trestle [Man91] may provide an elegant solution to this problem. Essentially, a cage is a region surrounding the cursor position; the system generates an event when the cursor leaves the cage. This mechanism generalizes the X notions of mouse motion (a one pixel square cage) and window enter and leave events (a cage corresponding to a window or its screen complement).

In its current state, eXene relies heavily on X for its rendering model, and for the implementation of windows and user event routing. Using the X rendering model has the benefit that the full generality of X drawing primitives is available to the programmer. We feel this advantage is more than offset by the complexity and low-level detail of this model. In addition, having the X model visible inhibits implementing eXene on any other graphics base. Tying a widget drawing context to an X window is a mistake: it is almost a truism that X windows are too heavyweight to be used extensively. This is particularly annoying in eXene, where the flavor is that of lightweight objects, as exemplified by function closures and CML threads and channels.

We hope to overlay the present library with one that supports a higher-level rendering model and a lighter-weight window model. We also wish to explore how well a constraint system can be integrated within eXene. These goals will require that eXene provides its own window management. The library will use the underlying graphics system, such as X, solely to provide primitive graphics services and a raw stream of user input.

ACKNOWLEDGMENTS

Some of this work was done while the second author was a graduate student at Cornell University, and was supported, in part, by the NSF and ONR under NSF grant CCR-85-14862, and by the NSF under NSF grant CCR-89-18233.

We wish to thank L. Augustsson, T. Breuel, T. Griffin, C. Krumvieda, H. Lin and T. Yan for testing the initial versions of eXene, pointing out bugs and suggesting various useful changes. We also thank the anonymous referees for their comments on the substance and style of the paper.

REFERENCES

[App87] A. W. Appel and D. B. MacQueen. A Standard ML compiler. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, vol. 274 of Lecture Notes in Computer Science, pages 301–324. Springer-Verlag, September 1987.

[Fre90] B. N. Freeman-Benson, J. Maloney, and A. Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, 1990.

[Gaj90] H. Gajewska, M. S. Manasse, and J. McCormack. Why X is not our ideal window system. *Software – Practice and Experience*, 20(S2):137–171, 1990.

[Gan88] E. R. Gansner. Iris: A class-based window library. In *Proc. USENIX C++ Conference*, pages 283–292, Denver, October 1988. USENIX Association, Berkeley, CA.

[Gan91] E. R. Gansner and J. H. Reppy. eXene. In R. Harper, editor, *Proc. Third International Workshop on Standard ML*, Pittsburgh, September 1991. Carnegie Mellon University, Pittsburgh, Pennsylvania.

[Gan92] E. R. Gansner, and J. H. Reppy. A foundation for user interface construction. In B. A. Myers, editor, *Languages for Developing User Interfaces*, pages 239–260. Jones & Bartlett, Boston, 1992.

[Get91] J. Gettys. Customization - Rope for a Noose, or a Lifeline for the Drowning? In *Proc. 5th X Technical Conference*, Boston, January 1991. MIT X Consortium, Cambridge, Massachusetts.

[Gos89] J. Gosling, D. Rosenthal, and M. Arden. *The NeWS Book*. Springer-Verlag, 1989.

[Gri92] T. Griffin and R. Moten. Tactic trees in eXene. Included in the eXene distribution, 1992.

[Haa90] D. Haahr. Montage: Breaking windows into small pieces. In *USENIX Summer Conference*, pages 289–297, June 1990. USENIX Association, Berkeley, CA.

[Har86] R. Harper. Introduction to Standard ML. *Technical Report ECS-LFCS-86-14*, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, August 1986.

[Kra88] G. Krasner and S. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.

[Lin89] M. A. Linton, J. M. Vlissides, and P. R. Calder. Composing user interfaces with InterViews. *IEEE Computer*, 22(2):8–22, 1989.

[Lin91] H. Lin. PAM: A process algebra manipulator. In *Proc. Third Workshop on Computer Aided Verification*, July 1991.

[Lis88] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proc. SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 260–267, Atlanta, June 1988. ACM, New York.

[Man91] M. S. Manasse and G. Nelson. Trestle reference manual. *Technical Report 68*, DEC Systems Research Center, December 1991.

[Mil90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Massachusetts, 1990.

[Mil91] R. Milner and M. Tofte. *Commentary on Standard ML*. The MIT Press, Cambridge, Massachusetts, 1991.

[Mye90] B. A. Myers, D. A. Giuse, R. B. Dannenberg, B. V. Zanden, D. S. Kosbie, E. Pervin, A. Mickish, and P. Marchal. Comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer*, 23(11):71–85, 1990.

[Nor87] S. C. North and J. H. Reppy. Concurrent garbage collection on stock hardware. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, vol. 274 of Lecture Notes in Computer Science, pages 113–133. Springer-Verlag, September 1987.

[Nye90a] A. Nye. *Xlib Programming Manual*, vol. 1. O'Reilly & Associates, Inc., 1990.

[Nye90b] A. Nye and T. O'Reilly. *X Toolkit Intrinsics Programming Manual*, vol. 4. O'Reilly & Associates, Inc., 1990.

[Pau91] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, New York, 1991.

[Pik89a] R. Pike. A concurrent window system. *Computing Systems*, 2(2): 133–153, 1989.

[Pik89b] R. Pike. Newsqueak: A language for communicating with mice. *Technical Report 143*, AT&T Bell Laboratories, April 1989.

[Rep86] J. H. Reppy and E. R. Gansner. A foundation for programming environments. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 218–227, Palo Alto, California, December 1986. ACM, New York.

[Rep88] J. H. Reppy. Synchronous operations as first-class values. In *Proc. SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 250–259, Atlanta, June 1988.

ACM, New York.

[Rep91a] J. H. Reppy. CML: A higher-order concurrent language. In *Proc. SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 293–305, Toronto, June 1991. ACM, New York.

[Rep91b] J. H. Reppy. An operational semantics of first-class synchronous operations. *Technical Report TR 91-1232*, Department of Computer Science, Cornell University, August 1991.

[Rep92] J. H. Reppy. *Higher-order concurrency*. PhD dissertation, Cornell University, Department of Computer Science, January 1992. Available as Technical Report TR 92-1285.

[Reps84] T. W. Reps. *Generating Language-based Environments*. The MIT Press, Cambridge, Massachusetts, 1984.

[Sar92] M. Sarkar and M. H. Brown. Graphical fisheye views of graphs. *Technical Report 84*, DEC Systems Research Center, March 1992.

[Sch92] R. W. Scheifler and J. Gettys. *The X Window System*. Digital Press, 3rd edition, 1992.

[Ung84] D. Ungar. Generation scavenging: A non-disruptive high-performance storage reclamation algorithm. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, Pittsburgh, Pennsylvania, April 1984. ACM, New York.

[Yan92] T. Yan. An incremental constraint solver for eXene. Included in the eXene distribution, 1992.