REVITALIZING EXENE

by

Matthew Hoag

B.S., Kansas State University, 2007

A THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Science
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2009

Approved by:

Major Professor
Alley Stoughton

Abstract

This thesis covers the process leading up to the release of eXene 2.0, a User Interface Management System (UIMS) toolkit. Since its inception, eXene has provided a unique way to create meaningful graphical user interfaces (GUIs) for Standard ML applications. Additionally, it has gone through several quality revisions which have both enhanced the toolkit and corrected many deficiencies that were present. Even with these improvements, however, the full potential of eXene has become increasingly difficult for developers to utilize. That is, in spite of the natural innovation that eXene brings to GUI construction, its current lack of extensibility, usability, and functionality, has caused Standard ML developers to choose simpler, more familiar UIMS toolkits, despite their limitations, for the creation of their applications. In light of this fact, eXene needs an internal and cosmetic overhaul to extend its usage and appeal. First, to improve its extensibility, formerly weakened by organic growth, eXene requires some restructuring of its architecture. Second, to improve its overall usability, previously stifled by sparse documentation, eXene requires the implementation of an interactive electronic document for its API. Finally, to improve its functionality, several new multi-purpose widgets are introduced. It is the author's hypothesis that a revised structure, improved documentation, and additional multi-purpose widgets sufficiently elevate eXene's extensibility, usability, and functionality such that eXene can be considered a fully featured UIMS toolkit. With these changes and the release of eXene 2.0, eXene is more likely to be adopted as the primary UIMS toolkit for Standard ML developers.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphical User Interfaces (GUI) toolkits are often considered to be a major selling point to application developers when choosing a programming language. They provide a means for developers to create, arrange, and modify a set of pre-made widgets to provide a graphical front-end to an application. In doing so, GUI toolkits can make a programming language more attractive by providing the developer with a straight-forward method to create an interaction environment for the users of their application. While GUI toolkits are not essential to the usefulness or popularity of a programming language, they often give it much needed appeal.

1.1 Standard ML

Higher order language such as Standard ML (SML)[RMM97] can benefit from a comprehensive and well-documented GUI toolkit. By itself, SML is a highly usable, feature rich programming language. It provides a strong typing system (with type inference), higher order functions, polymorphism, and a high degree of modularity. The standard distribution of SML also contains several extended libraries which greatly increase its applicability to various real world applications. One of these extended libraries, Concurrent ML (CML)[Rep99], deserves special mention for its contribution to the overall capability of SML.

1.2 Concurrent ML

Leveraging first-class continuations and the UNIX style signal handling integrated into the Standard ML of New Jersey compiler, the CML library constructs a set of concurrency primitives that can be used in conjunction with any SML application. More specifically, CML allows for the creation of one or more processes (or threads) that run in parallel and facilitates synchronous communication between said threads via typed channels. CML also provides a means for selective communication; that is, it gives a thread the nondeterministic choice to synchronize over one of several different channels. It is important to note that unlike many other concurrent languages, CML is a higher order concurrent language, where the communication and synchronization operations are themselves first-class values. This not only provides an abstraction of the synchronization operation, but also a mechanism to separate the description of the synchronization operation from the act itself[Rep99]. This concurrency concept, among many others inherent in CML, provides the basis for the communication of widgets and windows in eXene.

1.3 EXene

EXene[GR93a,GR93b], a multi-threaded User Interface Management System (UIMS) toolkit designed to create GUIs for SML applications, is quite unique and visionary in its implementation. The standard approach for handling user input (e.g., keyboard, mouse) in UIMS toolkits is to employ the event-based paradigm. This approach consists of a main loop that checks for incoming events from users or other threads and event handlers that provide a reaction or sub-routine to events specified by the developer. In this approach, the main loop and the input checking code are often built into the programming language itself because their computation is not dependent on the application. Conversely, the application developer is required to provide event handling functions for instances where the application needs to react with particular events. For those with experience in concurrency, it is easy to see that this simple event-based paradigm is only a small subset of the functionality provided by a full fledged concurrent system. Unlike almost every standard UIMS toolkit, eXene's interaction with the user and its inter-window/widget communication are implemented using the full set of concurrency primitives provided by CML.

1.4 Advantages of eXene

By virtue of its extensive concurrent backend, eXene reaps several substantial rewards. First, eXene developers are not forced to rely on the event loop embedded in the UIMS toolkit of choice (although it is theoretically possible to implement such a loop for use in eXene). Instead, the application developer maintains full control over the application's computation. Second, eXene remains free to execute additional computation over multiple threads where an event-based UIMS can be largely inactive waiting for the next input. Thirdly, while it is always possible for an individual GUI element to fail or deadlock in eXene, this does not cause the whole system to fail or deadlock. More specifically, the inherent modularity provided by CML allows GUI elements in eXene to fail gracefully without affecting the rest of the application. Finally, and possibly most important, eXene forgoes the imperative style of most UIMS toolkits and allows developers to utilize SML's powerful type system when providing communication between GUI components. The SML type system can be used to create both terse and descriptive communication depending on the situation and ultimately gives GUI construction a very functional feel. Overall, the usage of eXene allows the developers to maintain greater control over the execution of their application and provides a development environment that can exploit the functional nature and strong type system of SML.

1.5 The Current State of eXene

In combination, SML/NJ, CML, and eXene make a powerful application development environment. SML/NJ's strong typing system and functional nature coupled with eXene's high computational and concurrent versatility provide developers with a reliable and succinct means to create a variety of meaningful graphical applications. Nevertheless, eXene still contains a number of deficiencies. More specifically, eXene provides the "proof of concept" for a UIMS, but lacks some needed implementation. While great strides have been made to enhance the functionality of eXene[Gan95,DeB05], little attention has been addressed to making it accessible to developers.

1.6 EXene's Competition

It is important to note that eXene is not the only choice for GUI development in SML. Several SML bindings to well known UIMS toolkits exist, such as GTK+[LN01] and TCL/TK[LW04]. While the GTK+[1] and the TCL/TK bindings are widely sufficient for the construction of meaningful GUIs, they provide a fairly imperative approach to GUI development. Moreover, it is a common occurrence in application development to trade innovation for simplicity. Thus, many developers may trade the natural innovation and functional capabilities of eXene for a simpler and/or currently more fully featured UIMS toolkit.

1.7 The Fully Featured UIMS Toolkit

UIMS toolkits come in various forms and are used in a variety of languages. Yet, to be successful, they must provide a level of extensibility, usability, and functionality expected by the typical developer. That is, if a typical developer expects a certain set of features in a UIMS toolkit, and these features are missing, it is likely that no amount of innovation or uniqueness will persuade the developer to use the toolkit. However, with a sufficient level of functionality, usability and extensibility, a UIMS toolkit may be considered fully featured by developers.

[1]Currently, the bindings for GTK+ are only usable with the Moscow ML compiler and not with SML/NJ compiler. It is foreseeable, however, that they may be extended to SML/NJ.

1.7.1 Functionality

For a developer, functionality is often the primary determinant when choosing a UIMS toolkit. This is because a toolkit's functionality determines the range of GUI applications that can be constructed with its use. A UIMS toolkit with high functionality usually contains numerous low-level or modular GUI components that can be used in tandem to construct complex and meaningful GUIs, as well as several high-level GUI components[2] which are used consistently in generic application development. In their entirety, the low-level and high-level GUI components govern the type GUI applications that can be created along with the time and work required to do so. As a result, developers will choose a UIMS toolkit that possesses a level of functionality such that it is possible and efficient to create the envisioned GUI application.

1.7.2 Usability

Usability is of paramount concern to the developer when choosing a UIMS toolkit. It is often employed as the metric to determine a UIMS toolkit's ease of use. More specifically, usability defines the extent to which the overall structure and individual elements of functionality are explained to the developer. Highly usable UIMS toolkits generally have manuals that describe the basic usage and structure of the toolkit and also contain easily accessible, up-to-date documentation describing the functionality of the toolkit. In its extremes, the usability of a UIMS toolkit ranges from a guiding hand to an impassable barrier for the developer. As such, regardless of the functionality provided, usability dictates the developer's willingness to use the toolkit.

1.7.3 Extensibility

An element of foundational importance in a UIMS toolkit is extensibility. While extensibility is not a direct or immediate concern of the developer, it defines the longevity of the toolkit.

[2]High-level GUI components are generally fabricated completely from low-level GUI components.

If a UIMS toolkit lacks extensibility, it becomes increasingly difficult to add enhancements and correct certain deficiencies that are present. Thus, as time passes, a toolkit reaches a plateau in its functionality and is surpassed by more extensible toolkits. Consequently, developers will choose toolkits that demonstrate a high degree of extensibility.

1.8 The Revitalization of eXene

The subject of this thesis covers three areas of improvement needed to draw out eXene's full potential: a revised structure, improved documentation, and additional multi-purpose widgets. While the overall architecture of eXene has long been viable, through various improvements and additions, a number of structural problems have become apparent. Dealing primarily with lack of abstraction and incomplete integration, these structural problems are addressed in Chapter 2. Next, even though there are several manuals[GR93a,GR93b] and documents that provide an overview of the functionality of eXene, they represent earlier versions of eXene and give an incomplete list of eXene's current functions and their usage. The insufficient documentation of eXene has been a deterrent to its adoption. Therefore, a documentation solution for eXene is introduced in Chapter 3. Lastly, the addition of new multi-purpose widgets is proposed to extend the usability of eXene. In Chapter 4, widgets created by several developers, including two by the author, and their subsequent integration are described.

In order to maintain eXene as a viable option for GUI development in SML, the developer's perspective must be taken with regard to extensibility, usability, and functionality. It is the author's hypothesis that a revised structure, improved documentation, and additional multi-purpose widgets sufficiently elevate eXene's extensibility, usability, and functionality such that eXene can be considered a fully featured UIMS toolkit. In light of this revitalization, the natural advantages inherent in eXene are likely to become visible and appealing to the SML developer.

Chapter 2

Restructuring eXene

There are a number of structural elements that must be considered when revising the architecture of eXene. First and foremost, eXene is an X client, and thus interfaces with the X window system in order to provide services as a UIMS toolkit. Consequently, eXene is significantly reliant on the features provided by the X window system, which in turn, affects its base architecture. The next structural elements of importance are the eXene libraries. EXene contains two major libraries: the interface library, and the widget library. The interface library provides the means for eXene to communicate with the X window system, while the widget library provides the developer with a high-level tool for creating graphical objects. There are two additional minor libraries, the graph utility library[1] and the styles library[2], present in eXene. The last structural element is the compilation and library manager[Blu02], a tool used to modularize and compile large projects in SML/NJ. Its consideration is important because any physical change in the directory hierarchy of eXene must also be mirrored in compilation and library manager source files. Given eXene's architecture, the structural decisions and subsequent revisions of eXene can have far reaching effects. Thus, it is the goal of this thesis to make such changes only with the proper justification.

[1]The graph utility library was introduced with the initial release of eXene[GR93a]

[2]The styles library was introduced in the release of eXene 1.0[Gan95]

2.1 The X Window System

The X window system (X) is a client-server application that allows for the creation of GUIs using bitmaps. The server side of X, or the X server, provides the functionality for sending user input to the client-side of X and also rendering text and graphics in windows on a physical display. The X client, then, has the complementary abilities to request information about the state of the X server, receive information about user requests, and most importantly request the X server perform a particular action (e.g., draw an object, place text). Therefore, eXene must, in some capacity, provide this functionality in order to serve as an X client.

2.2 eXene's Interface Library

The interface library of eXene includes nearly all of the functionality required of an X client. In general, it follows the X protocol conventions set forth in X[Nye90], and in that way, eXene's interface library is operationally comparable to the widely used X client, Xlib. The interface library is broken up into eight structures (or modules). These modules are categorical abstractions of eXene's various low-level components which facilitate communication with the X server. Because these low-level components are numerous in size and functionality, the abstractions provide a clean way for eXene developers to operate in the X environment without an in-depth knowledge of the interface library's structure[GR93a].

Due to the strong typing system present in SML, there were additional considerations when making the interface library abstraction. Many of the low-level components use the same types and structures, but after these types and structures are absorbed into the abstraction, the declarations remain independent of each other. For example, the same `Geometry` structure (seen in Figure 2.1) is instantiated in the various low-level components that compromise `EXeneBase`, `Drawing`, `ICCC`, `Interact`, and `EXeneWin`, but this fact is unknown to the abstracted components and would raise a type error during compilation. Thus, it is necessary in eXene to use the `sharing` keyword to explicity state that these `Geometry`

```
structure EXene :> sig

    structure Geometry  : GEOMETRY
    structure EXeneBase : EXENE_BASE
    structure Font      : FONT
    structure Drawing   : DRAWING
    structure ICCC      : ICCC
    structure Interact  : INTERACT
    structure EXeneWin  : EXENE_WIN
    structure StdCursor : STD_CURSOR

    sharing Geometry  = EXeneBase.G = Drawing.G = ICCC.G = Interact.G
                      = EXeneWin.G
    sharing EXeneBase = Font.EXB = Drawing.EXB = ICCC.EXB = Interact.EXB
                      = EXeneWin.EXB = StdCursor.EXB
    sharing ICCC      = EXeneWin.ICCC
    sharing Interact  = EXeneWin.Interact

    sharing type Font.font = EXeneBase.font
    sharing type EXeneWin.window = Drawing.window = EXeneBase.window
    sharing type Drawing.pixmap = EXeneBase.pixmap
    sharing type Drawing.tile = EXeneBase.tile
    sharing type Drawing.font = EXeneBase.font
    sharing type Drawing.color = EXeneBase.color
    sharing type ICCC.atom = EXeneBase.atom

  end = struct
```

Figure 2.1: *shared types and structures in abstract.sml*

structures are all the same despite having slightly different heredity.

For ease of development the structural names of the low-level components are removed and their types and functions are opaquely ascribed to one of the eight modules[3]. In the case of `EXeneBase`, the types and functions of `XProtTypes`, `DrawTypes`, `FontBase`, `Display`, `Cursor`, `ColorServer`, and `Pixmap` are all completely absorbed by the abstract module

[3]Clearly, not all of the functions from the low-level components are ascribed to one of the eight components, as that would defeat the purpose of the abstraction. Therefore, it is important to note that each of the eight modules has a signature which determines which functions from the low-level components will be available to the developer.

```
structure EXeneBase : EXENE_BASE =
struct
    structure G = Geometry
    structure XTime = XTime

    open EXeneVersion

    exception BadAddr = XDisplay.BadAddr

    open XProtTypes DrawTypes FontBase Display Cursor ColorServer Pixmap
         Image Tile
    open HashWindow
    ...
end
```

Figure 2.2: *EXeneBase in abstract.sml*

(shown in Figure 2.2). As noted previously, the name of the low-level components no longer needs to be addressed, post abstraction. That is, if a developer needs to reference the type `Display.display` through the abstraction he/she would only be required to reference `EXeneBase.display`.

In the end, the abstracted version of the eight modules are neatly packaged as substructures in the `eXene` structure. To access a module one need only `open` or reference the `eXene` structure for it to become visible. While the low-level components of eXene's interface library are quite complex, the interface abstraction is straight-forward and easy to use. Additionally, the whole of the interface library is abstracted through the interface abstraction, and contains little to no functionality beyond communication with the X server.

2.3 The Widget Library

The widget library, the second major library of eXene, is built on top of the interface library. It further abstracts the window, drawing, and bitmaps routines of the interface library into extensible graphical components known as widgets. Widgets come in many different types, varying from simple widgets (like a button widget) to composite widgets (which encapsulate multiple widgets and can control their position, dimension, and visibility). With the latter widget type, it becomes possible to create a widget hierarchy, where a set of widgets can exist at the same time. In order for this widget hierarchy to exist, widgets must adhere to a basic uniformity to promote, among other things, internal communication[4]. This internal communication is made possible using CML's selective communication and event abstraction and yields nicely to the creation of a communication handler in each widget. Because this internal communication in widgets is highly concurrent, it is possible to filter or further abstract message events at each level of the widget hierarchy. Ultimately, an eXene developer is able to create highly complex and stable GUIs easily by adding various simple widgets to a composite widget.

2.4 The *Other* Libraries

The graph utility and styles libraries both contribute greatly to eXene, albeit in different ways. The graph utility library provides eXene users with the ability to draw various shapes, such as ellipses, splines, and rounded rectangles. The styles library interfaces with the X Resources provided by X and allows for the specification of various attributes[5] from multiple sources, be it the developer or the user. Moreover, the styles library has the ability to resolve multiple styles to a singular style, based on an internal precedence. Given their collective functionality, both libraries live in a realm somewhere between the interface library and the widget library. That is, both the graph utility and styles library use certain functionality provided by the interface library and are subsequently used by the widget library. For example, the `Ellipse` structure present in the graph utility library uses the `Geometry` and `Drawing` structures from the interface library to create a graphical ellipse, which in turn may be used by the widget library to create an eliptical button.

[4]This uniformity deals with various attributes and functions that widgets must provide in order to interact correctly the widget hierarchy. Also, an eXene developer must follow some internal protocols to actually use widgets, such as making the root widget and attaching a shell widget to the widget heirarchy.

[5]Currently, eXene only supports styles for widgets', however, the library is extensible to other high-level objects.

```
Library
...
   signature EXENE_BASE
   structure EXeneBase
...
is
   lib/sources.cm
```

(a) (Library) "eXene.cm"

```
Group (../eXene.cm)
...
   signature EXENE_BASE
   structure EXeneBase
...
is
   lib/base.cm
   lib/user/build.sml
```

(b) (Compilation Unit) "lib/sources.cm"

Figure 2.3: *Library and Compilation Unit Comparison*

2.5 The Compilation and Library Manager

The compilation and library manager (CM)[Blu02] is a tool provided by SML/NJ that helps a developer structure and compile large projects. Its basic function is to construct libraries from the SML/NJ source files, compilation units and/or libraries provided to it. The libraries define a set of visible structures that are available when compiling a `.sig` or `.sml` file. Alternatively, the compilation units provide structure visibility only to the library in which it is grouped.

Both libraries and compilation units are created using `.cm` source files. A library is constructed using the `Library` and `is` keywords, while a compilation unit is constructed using the `Group` and `is` keywords (as seen in Figure 2.3). When creating libraries, the signatures and structures that will be visible in the library are preceded by the `Library` keyword, and the source files, compilation units and/or libraries needed to generate those signatures or structures are preceded by the `is` keyword (as seen in Figure 2.3a). Likewise, when creating compilation units, the signatures and structures that will be visible in the compilation unit are preceded by the `Group` keyword, and a collection of source files, compilation units and/or libraries needed are preceded by the `is` keyword (as seen in Figure 2.3b). It is important to note that the compilation units also have the name of the library target directly following the `Group` keyword to identify the library to which they belong.

When referencing structures and signatures in CM, compilation units and libraries, as opposed to source files, are handled differently. That is, in a specific library or compilation unit, a source file may only be referenced once throughout the chain of dependency[6], whereas other libraries or compilation unit may be referenced as many times as needed. Thus, the chain of dependency in compilation units and libraries can be viewed as an incomplete directed acyclic graph[7] (DAG) instead of a standard dependency tree. Ultimately, the DAG dependency functionality provided by the CM allows for the creation of terse compound libraries with dependencies (libraries and compilation units) that may themselves be dependent on one another.

2.6 Refactoring eXene

The goal in the redefinition of the eXene structure is to consolidate the physical location of the four libraries without loss of functionality. The interface library (see Section 2.2) and the widget library (see Section 2.3) have no overlapping functionality, so it follows that these two libraries remain separate, both on an abstract and physical level. The graph utility and styles library (see Section 2.4) are a different matter entirely. These libraries provide useful functionality to the widget library in the very same way the interface library provides functionality to the widget library. This similarity in utility forms a link between the libraries. Additionally, the graph utility library and the styles library are abstracted in such a way that they could potentially be applicable to another high level library other than the widget library. Specifically, an improved conceptual model for creating GUIs in the X window system (besides widgets) could potentially benefit from these libraries. Thus, one can conclude that these libraries and their sub-parts, should not be encapsulated in the widget library, but could indeed be tied in some way with the interface library.

It could be argued that the link between the graph and styles libraries and the interface library warrants the refactorization of eXene. When entering the directory structure it is

[6]More specifically, if a library or compilation unit directly includes a source file as a dependency, none of its dependent compilation units or libraries may include that source file.

[7]The chain of dependency must be considered an incomplete rather than complete directed acyclic graph, because (as mentioned previously) source files can only be referenced once.

easy to assume that any of the supporting functionality for widgets comes from the `lib`[8] directory. It is not readily apparent, however, that the `graph-util` and `styles` directories have supporting functionality for widgets. Therefore, on a purely physical level, it made sense to move the physical location of the `graph-util` and `styles` directories inside the `lib` directory (this change in the directory tree of eXene can be seen in Figure 2.4).

As one might expect, the migration is not complete with a simple `mv`[9] of the two directories. This change must also be reflected in the CM source files. Originally, the graph utility and the styles libraries were part of their own compilation unit grouped to the overall eXene library. After the directories were moved to the `lib` directory, the contents of their respective CM source files (the signature and structure definitions and the files where they are defined) needed to be migrated to the `lib` directory as well. Furthermore, because the libraries became a part of the `lib` directory, it was advisable to force their compilation in the `lib/sources.cm` compilation unit rather than having them compiled in separate compilation units.

In addition to the library refactorization, there was a need for further compartmentalization of some of the source files present in the `graph-util` directory. A new `xauth` directory was created (seen in Figure 2.4) to house all of the source files that dealt with X authorization. Previously, the X authorization files were located in `graph-util`, causing the directory to become bloated and preventing an eXene developer from finding them easily. Consequently, moving all of the X authorization files to a directory that was correctly labeled was advantageous to the developer. It is important to note that again this move was reflected in the `lib/sources.cm` file so that everything compiled correctly.

This refactorization of eXene has added a structural benefit to the inner working of eXene by developing the `lib` directory into a complete library. In fact, as a product of the changes, the `widgets/sources.cm` compilation unit need only include the `lib/sources.cm`

[8]The lib directory currently houses the entirety of the interface library.
[9]mv is a *nix command used for moving files or directories.

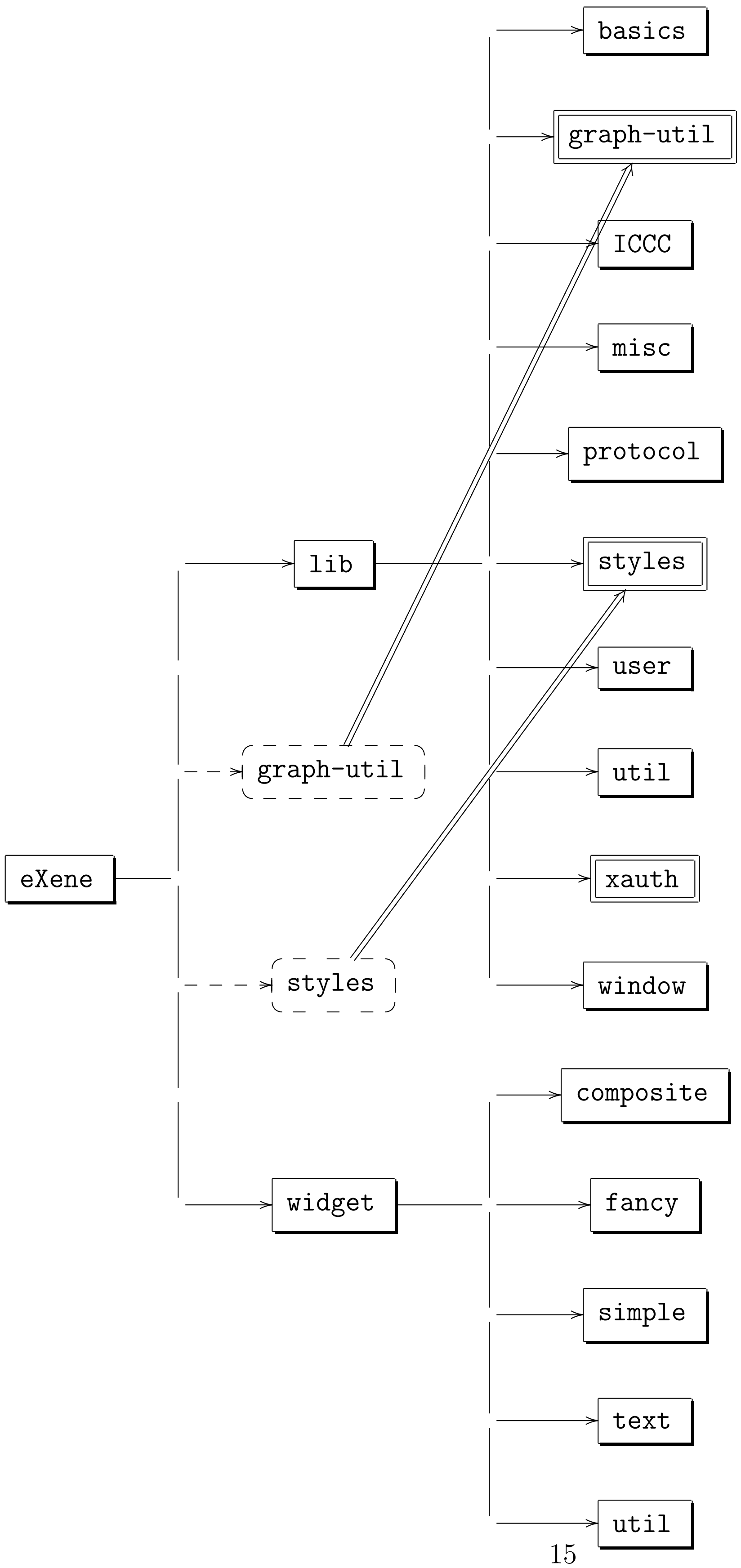

Figure 2.4: *eXene directory refactorization*

compilation unit to compile successfully[10]. Furthermore, the reorganization of the directory structure has provided some much needed transparency for the eXene developer. In the recent iterations of eXene, there has been much effort to improve the existing X authorization problems[DeB05]. Now, with the transparency, it will be easier to make additional improvements to X authorization. Overall the refactorization was substantial, but was needed in order to consolidate the structure of eXene for future development.

2.7 Improving the Abstraction Model

Abstraction, with regard to application development, is always important to consider. It is the basis for providing modularity in programming, and hiding the inner workings of a module. In this way, it allows for future revision of the basic structure of a module without the subsequent loss of functionality. It also decouples the internal relation between various programming components so that they can be used independently of one another.

2.7.1 The state of eXene's abstraction model

In the development of eXene, abstraction was a constant consideration. This fact can be seen in the structure of eXene itself, and also in the initial choice of a functional and concurrent programming model. That being said, there are areas in the development of eXene where the usage of abstraction was slightly ignored or deficient.

Like many programming projects, eXene has gone through a period of organic growth. More specifically, certain extended features were desired on the widget level of eXene, yet these features did not necessary apply only to the widget level. As a result, when they were implemented on the widget level, there was a loss of abstraction and potential functionality. In order to solve this problem, these features needed to be moved to the correct location both on a physical and abstract level.

[10]This contrasts with the previous implementation which required the inclusion of both the graph-util/sources.cm and styles/sources.cm.

2.7.2 Widget types

One area for improved abstraction is in the definitions for basic widget types. As mentioned in Section 2.3, the concept of the widget is derived from the X window. Like windows, widgets have physical dimensions, alignment, and gravity. Unlike windows, however, widgets have "widget state", in which they can be active or inactive[11]. Yet in eXene, there is no definition for a window's basic types, only a widget's. In fact, the idea that X windows have basic types is conceptually removed from eXene. Instead the entirety of a window's attributes is specified in `widget-base.sml` (as shown in Figure 2.5). In the event that a new conceptual model for GUI design is created in eXene, the definition of base widget types would be completely useless. For this reason, a definition for basic window types was made.

To begin dealing with this problem, a suitable location for the definition of base window types had to be found. Conveniently, in the newly arranged library there is still a `window` directory that seems suitable for the new source file, `window-base.sml`. Unfortunately, all of the source files present in the `window` directory are part of the interface library abstraction, making `window-base.sml` the only source file outside of the interface abstraction. To rectify this problem, it is possible to pull the definition of the base window types behind the interface abstraction, but this is unnecessary because the base window types are already at a sufficient level of abstraction. A more reasonable solution, and the one chosen by the author, was to create a new directory named `window-aux` which can house the window related library components that are outside the interface library abstraction.

Next, it became necessary to separate the definition of the base window types from the base widget types. As noted previously, the only type that does not belong to the base window types is the widget state (or `wstate`). As a result when migrating the source code, nearly all of `widget-base.sml` was moved to `window-base.sml`, with the exception of the `wstate` datatype. To complete the move, the line:

[11]The concept of a window in X does not include "state", thus a widget has more functionality than its predecessor.

```
signature WIDGET_BASE =
  sig
    ...
    datatype valign = VCenter | VTop | VBottom
    datatype halign = HCenter | HRight | HLeft
    datatype gravity = Center | North | South | East | West |
                       NorthWest | NorthEast | SouthWest | SouthEast
    datatype arrow_dir = AD_Up | AD_Down | AD_Left | AD_Right
    datatype wstate
      = Active of bool      (* state may be affected by user actions *)
      | Inactive of bool    (* state cannot be affected by user actions *)

    type shades = ShadeServer.shades
    exception BadIncrement

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

    type bounds
    val mkBounds : { x_dim : dim, y_dim : dim } -> bounds

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

    type win_args

    val wrapCreate : (EXeneBase.window * G.rect * win_args) -> EXeneBase.window
    val wrapQueue : 'a CML.event -> 'a CML.event
    val wrapFlushableQueue : 'a CML.event -> ('a CML.event) * (unit CML.event)

  end
```

Figure 2.5: *(widget-base.sml) Initial signature definition for basic widget types*

```
include WINDOW_BASE
```

was added to the `widget-base.sml` signature and the line:

```
open WindowBase
```

was added to the `widget-base.sml` structure. Finally, `window-base.sml` was added to `lib/sources.cm` and the definition of its signature and structure was added to `lib/sources.cm` and `eXene.cm` for it to become completely visible. In the end, eXene retains the entirety of its functionality with a greater degree of abstraction.

2.7.3 Widget utilities

A similar area for improved abstraction is found in the `widgets/util` directory. Within this directory there are a number of components that aid in the creation of widgets, but are not technically tied to widgets themselves. One set of components deals with the creation of image, shade, tile servers. Another set deals with reading and writing bitmaps and drawing three-dimensional objects. Because these components are clearly not widget-specific, they do not belong in the widgets library and therefore a new location needed to be found.

On a functional level, the components used for the creation of image, shade, and tile servers are most similar to the low-level components found in the `lib/window` directory, but the components in the `lib/window` directory remain hidden behind the the interface library abstraction. The newly created `window-aux` directory already houses window components that are not part of the interface abstraction, which made it a good location to put the image, shade, and tile server components. Alternatively, the components used to read and write bitmaps, as well as draw three-dimensional graphical objects, are most similar to the mid-level components found in the now consolidated `lib/graph-util` directory. Unlike the previous components in `widget/util`, these components are at the same level of abstraction as those with which they are grouped, leaving no ambiguity about the move.

After the migration of the previously mentioned components from `widget/util`, only one component, `attrs.sml`, remains out of place. The source file `attrs.sml` provides a list of definitions for attributes in widgets and appears to be widget-specific. With the changes made to `widget-base.sml` and the creation of `window-base.sml`, however, these attributes are no longer widget-specific, and are instead window-specific. Consequently, after modifying `attrs.sml` so that it reflects this fact, a more appropriate location for the component was the `lib/styles` directory, as a generic attribute list for window descendants. Lastly, a new definition for widget attributes was declared, namely `widget-attrs.sml`[12], so that the `Style` structure created for widgets can still be parameterized by an attribute set specific to widgets.

After all of the components, previously in `widget/util`, were physically moved to a more appropriate place, this migration was also reflected abstractly in the compilation unit in which they were originally placed and in the compilation unit where they were moved.

2.7.4 The Interact abstraction

The final area of deficiency in the current abstraction model is in the `Interact` abstraction. The `Interact` abstraction provides a structured list of datatypes and functions to keep track of user input. Previously, `Interact` internally kept track of user mouse clicks through the `MOUSE_Up` and `MOUSE_Down` types. As a whole, these types kept track of the mouse button being pressed, the mouse position in window coordinates, the state of the other mouse buttons, and the time that they were pressed. Although this user information is sufficient for most eXene applications, it fails to determine if modifier keys (shift, control, etc.) are depressed when a mouse button is clicked. As a result, it is impossible to issue a shift-click or control-click command to an eXene widget.

On many levels, the interface abstraction removes a great deal of hardship when dealing with the X server. However, in the case of mouse clicks, the abstraction is too strict and actually removes potential functionality. To address this problem, changes must be made to the lower level components that collect and define user events and also to the

[12]The definition of attributes in widget-attrs.sml can easily be constructed by opening the structure present in attrs.sml.

`Interact` abstraction. First, the low-level component, `window-env`, which specifies the different types of user events of interest to eXene, needed to be expanded so that modifier key state was included as a part of the `MOUSE_Up` and `MOUSE_Down` definitions. Next, the `toplevel-win.sml`, which collects mouse and keyboard events from X server and then wraps them up as `MOUSE_Up` and `MOUSE_Down` types, was altered so that it routes the modifier key state into the instantiation of the mouse click types. Finally, the `Interact` abstraction was changed to reflect the expansion of the mouse click types.

2.7.5 Benefits of the revised abstraction model

On both a low and high level, eXene suffers slightly from problems dealing with lack of abstraction and over abstraction. However, with the revised abstraction model proposed by the author, many of these abstraction problems are mitigated. The redefinition of `window-base.sml` and the migration of `widgets/util` strengthens the modularity of eXene and opens up the base functionality eXene to future high-level libraries. Moreover, the refinement of the `Interact` abstraction gives an eXene developer the means to interact more fully with the users of his/her application. Both changes in the abstraction model strengthen the capabilities of eXene without drastically changing the way eXene is used by the developer.

2.8 Integrating the Enhancements

As mentioned in Chapter 1, many enhancement have been proposed in eXene. These enhancements include the introduction of styles[Gan95], customizable input focus, and customized X resources; as well as the implementation of X selection, and the revision of X authorization[DeB05]. All of these enhancements greatly improved the usability of eXene in principle. Some, however, require extensive implementation to reap any benefit. The proposal for input focus is one such enhancement.

With regard to eXene, input focus determines which widget receives the keyboard input

```
structure SimpleEdit : SIMPLEEDIT =
struct
...
   fun handleKey (KEY_Press key, me as
                  {str,selpos=ss,sellen=sl,selrf,selre,wid,ht}) =
                  (case key of
                     (KEYSYM(65289),_,xt) => (* tab *)
                        (send(focChan,S.Next xt); me)
                  | (KEYSYM(65056),_,xt) => (* shift+tab *)
                       (send(focChan,S.Previous xt); me)
                  ...
   end
...
end
```

Figure 2.6: *(simple-edit.sml) Cycling/Releasing focus in the simple string edit widget*

from the user of the X application. By default, the position of the mouse determines which widget ultimately receives the input. Before the proposal for a customizable input focus, this behavior could be rather annoying to the user, because he/she would have to move the mouse over a widget for it to receive a key press. With the proposal, the application would determine the default widget for the input focus and the application could cycle, release, and set input focus on particular widgets, based on user input (the Tab key or mouse clicks).

For this addition of customizable input focus, a two-tiered implementation is required. First, a focus manager must be added to the shell which was accomplished by Dusty DeBoer[DeB05]. Next, for every widget to be recognized by the focus manager, two additional functions are required: the `takeFocus` function[13] and the `focusableOf` function[14]. The widgets must also have a focusable message channel so that they can capture certain key strokes and relay them to the focus manager.

The latter tier of the implementation remains largely unimplemented. As such, the only widget that can truly gain and release focus is the `TextEdit` widget implemented by Dusty

[13]This function tells the widget to take input focus by sending on the widget's request channel.

[14]This function is used by the focus manager to determine the channel on which the widget will communicate with the focus manager (that is, tell the focus manager that the tab key has be pressed). It also contains an internal function that tells the focus manager how to make widget take focus.

DeBoer[DeB05]. Moreover, the way in which a widget cycles or releases focus is hard-coded into widget itself and cannot be modified without making a change to the source code (see Figure 2.6). While providing the ability to control input focus is advantageous to the developers and users of eXene applications, it must be fully integrated into eXene for the benefits to be realized.

As a solution, the focusable functionality was added to various widgets that would benefit from its implementation (such as `Button` widgets and the text editing widgets), in parallel with a new widget convention for customizing the cycle/release of focus. The new widget convention establishes two new style attributes among the widgets, `KeyFocusNext` and `KeyFocusPrev`, which are set to the default values of a tab key press and a shift-tab key press respectively. Ultimately, the developer or user is able to modify how an application using these select widget cycled/released input focus.

Chapter 3

eXene Documentation

Application Programming Interface (API) Documentation has historically been considered central to the usability and correct application of a programming language and its libraries. In most cases, it provides essential information about the usage and potential side effects of various functions and language constructs. Without extensive, up-to-date documentation, it can be difficult for a developer to use a language and its libraries correctly, as well as find the functions that are important for the development of his/her specific application.

While the documentation in SML/NJ and CML is extensive and up-to-date, eXene's documentation is in a state of disrepair. Originally, two manuals[GR93a,GR93b] outlined the basic usage of its interface, widget, and graph utility libraries. At that time, the manuals were sufficient to describe eXene's functionality, but as improvements were made to eXene over the years, these manuals were not updated and currently provide an incomplete list of eXene's functions and their usage. Even though additional documents were added to complement[Gan95,DeB05,DS05] the existing set, it remains difficult for eXene developers to locate specific functionality of interest among the documents.

One way to efficiently provide API documentation for a programming language and its libraries is a documentation generator. A documentation generator has the ability to parse the source code and comments of a set of files in given programming language and produce an API of the contents. Although it varies greatly depending on the document generator, the API can be presented as a plain document (RTF) or as an interactive document (HTML or PDF). Generally, the interactive API documents are preferred over those with plain text, because they contain a number of hyperlinks that allow a developer to quickly navigate to related information on different pages of the document.

By the virtue of design, documentation generators help keep an API up-to-date. Because API documentation is derived from the latest version of the source code, any changes in the structure of the source files are automatically reflected in the most recently generated documentation. This has an added benefit to developers providing comments to the APIs because they are no longer required to mirror their source code comments in an external API document. Instead, the API document is generated automatically from the comments already present in the source code.

3.1 An interactive documentation solution for eXene

ML-Doc[Rep07] is a documentation generator for SML that can create an interactive API document. It has been used to generate the API documentation for both the SML/NJ basis library and CML. As such, it can also be used with eXene to produce an interactive API document.

Unlike some documentation generators, ML-Doc does not create the API document directly from the SML/NJ `.sig` and `.sml` source files and their comments. Instead, it generates the documentation from intermediate ML-Doc (`.mldoc`) files, implemented in the Standard Generalized Markup Language (SGML). This creates a problem of consistency because the signatures in the eXene source files and the `.mldoc` files can be edited independently. While this inherent dissonance in the design of ML-Doc can potentially keep the documentation out-of-date, it is unavoidable given the functional nature of SML/NJ, and the expanded hyperlink functionality available in ML-Doc.

It is possible, however, to eliminate this dissonance between the signatures and the `.mldoc` files in eXene. While it is not possible to make the continued generation of `.mldoc` file dependent on the file containing the definition of a signature, the reverse is entirely possible. That is, ML-Doc files contain sufficient information to completely generate the signature of an SML/NJ component. In fact, the `extract-sig` function provided by ML-Doc can be used to create a signature from an ML-Doc file. This process is explained in more detail in Section 3.4. It is important to note, however, that forcing the dependency and removing the previously mentioned dissonance is not without side-effects. It requires that all future developers of eXene's API become proficient in editing `.mldoc` files, in view of the fact that they will no longer be able to directly edit the `.sig` files containing the eXene signatures.

3.2 Setting up ML-Doc

In order to use ML-Doc with eXene several changes needed to be made to the structure of eXene and the host development environment. First, in order to use ML-Doc, a developer must add an SGML parser[1] and ML-Doc[2] to their development environment. Following the installation of ML-Doc, the eXene directory tree was modified to hold the `.mldoc` files, the configuration files, and the generated document. At the root level of eXene a `manual` directory was created. Within this directory, the `HTML`, `Info`, and `Sigs` directories was created. These directories hold the `.html`, `.info`, and `.sig` files repectively, which are generated from the `.mldoc` files in eXene. An `ML-Doc` directory was also be created to hold all of the `.mldoc` files present in eXene.

With the new directory tree in place, it was necessary to create the configuration files which ML-Doc uses to generate the API document. The configuration files include: `Config.cfg` (the document's configuration file), `CATALOG` (the document's catalog), and `Entities.sgml` (a list of the document specific components). Initially, a mostly empty `Entities.sgml` file was created to house all of the references to components that will be in the finished document. Next, the `CATALOG` was created that includes a path to

[1]The defacto SGML parser and validator can be retrieved at `http://sourceforge.net/projects/openjade/files/opensp/OpenSP-1.5.1.tar.gz/download`

[2]ML-Doc can be retrieved at `http://people.cs.uchicago.edu/~jhr/tools/downloads/ml-doc.tgz`

the `Entities.sgml` and to the ML-Doc catalog. Finally, the standard template for the `Config.cfg` file was used[Rep00], with additional definitions for the location of the `CATALOG` and the path to the SGML parser.

For ML-Doc to function correctly the `CATALOG` and `Config.cfg` must include *hard* paths to ML-Doc catalog and the SGML parser. This fact can be slightly inconvenient if multiple developers are working on eXene at the same time, because eXene is under version control, specifically subversion. As it is, the hard paths prevent development from being workstation independent. To fix this problem a script (see Appendix A) was created for eXene that generates the `CATALOG` and a symbolic link to the location of the SGML parser so they do not have to be under version control. With the generation of these files it is possible for eXene to specify the *hard* paths required by ML-Doc and remain workstation independent.

3.3 Creating ML-Doc files

In order to begin the construction of the eXene API document, `.mldoc` files were created. To do this it was necessary to use the ML-Doc command `mkdoc` with a file containing an SML/NJ signature as an argument. After an `.mldoc` file is created, it was moved to the newly create `ML-Doc` directory. The ML-Doc documentation generator is not currently able recognize a directory structure in the `ML-Doc` directory so all of the `.mldoc` files must exist together and cannot be grouped in directory structure similar to their ancestors. Finally, for each `.mldoc` file created, the line:

```
<!ENTITY NAME SDATA "name.sig">
```

was added to Entities.sgml where the `NAME` was replaced with the component's signature and the `name.sig` was replaced with the component's signature file.

This process can be a difficult to remember and must be done for each new component created in eXene. So, a script, `mkdoc-tool` (see Appendix B), was created to expedite the process. To run the script one need only provide it with a `.sig` or `.sml` file where the signature is defined[3]. After receiving the file as an argument, it makes the `.mldoc` file, moves it to the `manual/ML-Doc` directory, and adds the appropriate entry into `Entities.sgml`. For the benefit of compilation, the script also creates a symbolic link to the `.mldoc` in the directory where signature is located.

3.4 ML-Doc and CM

As mentioned previously, the way in which the CM compiles eXene must be changed. That is, the CM must be made aware that the signatures in eXene are generated from `.mldoc` files. To accomplish this, two separate tasks was completed. First, a script to generate the signature (from the `.mldoc` file)[4] and subsequently moves it to the correct location, was constructed. Second, the script was registered to the CM so that the CM can use it during compilation.

The script, `mldoc-tool`, is fairly straight forward in its construction, but in order for the compilation to proceed correctly, a symbolic link was made in each directory where there is a compilation unit or library (`.cm` files). In CM, it is also possible to register simple shell commands with the `Tools` library. Using CM's `Tool` library, the `mldoc-tool.sml` source file was created, which registers and runs the `mldoc-tool` during compilation. As a result it is possible to replace all of the references to files (which provide the definition of a signature) in the compilation units with `.mldoc` files (as show in Figure 3.1). When considering the CM source files, the eXene developer can view the `.mldoc` files as a definition of the signature for a component and the `.sml` files as a definition of the structure for a component.

At this point, any further changes that need to made to the signature of a component should only be made to the `.mldoc` file which now defines the signature. Although this requires eXene API developers to learn how to use SGML as it relates to ML-Doc[5], it strengthens the documentation of eXene overall.

[3]One must also call the script from the directory where the signature file is located.

[4]To generate the signature the extract-sig tool is used which is provided by ML-Doc

[5]A more verbose coverage of ML-Doc[RB07] is available at `http://www.cse.unsw.edu.au/~tbourke/software/ml-doc.1.html`

```
Group (../sources.cm)
...
is
...
   bitmap-io-sig.sml
   bitmap-io.sml
   ellipse-sig.sml
   ellipse.sml
...
```

(a) (Previous) “lib/graph-util/sources.cm”

```
Group (../sources.cm)
...
is
...
   bitmap-io.mldoc
   bitmap-io.sml
   ellipse.mldoc
   ellipse.sml
...
```

(b) (Current) “lib/graph-util/sources.cm”

Figure 3.1: *Previous and current versions of “lib/graph-util/sources.cm”*

3.5 Populating the eXene Documentation

Using the functionality provided by ML-Doc and the tools constructed in this thesis, it became possible to create an interactive API document for eXene. All that was left to do was edit the skeleton `.mldoc` files such that they included a brief description of each of the eXene modules as well as an explanation of the individual functions provided by the modules.

To accomplish this task a number of sources were referenced and their information was consolidated into the appropriate `.mldoc` files. These sources included the library[GR93a] and widget[GR93b] manuals, Dusty DeBoer’s thesis[DeB05], and the source files themselves along with their comments. When the information in the sources was contradictory, precedence was generally given to the most recent documents as well as to the comments or functionality in the source files over the documents themselves. It is important to note that some of the functionality provided by eXene had little to no initial documentation. In these instances, the source code was analyzed and an appropriate description was given.

In all cases, the author strived to make the descriptions of the modules and explanations of the functions as terse and meaningful as possible. Additionally, where applicable, an X resources attribute table was included in the widget descriptions which summarized the resources parameterization of the widget. In the end, with the population of the `.mldoc` files completed, the subsequently generated interactive API document was both usable and comprehensive, containing nearly all of the information needed by the eXene developer.

Chapter 4

New eXene Widgets

Of the four libraries in eXene, the widget library is the most useful to developers attempting to create a GUI for a SML/NJ application. The widget library provides a set of high-level components (widgets) that can be used in tandem to create complex GUIs. While eXene's functional and concurrent nature may spark interest in application developers, the utility provided by the widget library determines whether it is viable for a particular GUI application. As a result, the number, variety, and usefulness of the widgets provided in the widget library ultimately affect the overall appeal of eXene.

Currently, eXene contains many of the basic widgets needed for GUI development. Its simple widget collection consists of a highly configurable button[1], a drawing canvas, an item list, a text list, a colored rectangle, and a scrollbar widget, as well as various label and message widgets. The composite widget collection contains several widgets that can be used to encapsulate other widgets, create a scroll port around a widget layout, and make a simple menu. EXene also provides a set of additional text widgets for text editing.

Some of these widgets are extensible and can serve many different functions. For example, the simple menu widget can easily be configured as a tooltip for other widgets. Many widgets, however, are either less versatile or deficient in some way. Consequently, it is necessary to improve existing widgets, or even add new widgets in the widget library.

[1]The button widget can be configured as a labeled or toggle button; as well as take the form of a variety of different shapes.

4.1 Selectable List

One widget that has a significant deficiency is the text list widget. The text list is an extension of the item list widget, which maintains a set of items using widget state. In general, the text list widget is useful to a developer attempting to represent a toggle-able text list in his/her GUI. The text list, however, is very limited in its ability to toggle the items it represents. Using the text list it is only possible to toggle a *single* item on or off using a mouse click; there is no way to select or deselect several items using a shift-click or control-click. Since selecting and deselecting multiple items in a text list is a common feature in modern GUI toolkits, it would be beneficial to have a text list widget in eXene with expanded selection capabilities. The creation of the selectable list was an attempt by the author to solve this problem.

Structurally, the selectable list is much different from the text list. Instead of using the item list as its basis, the selectable list uses an internal mechanism to maintain the list of items. The internal mechanism also has expanded functionality to handle multiple selection[2]. Additionally, the selectable list makes use of the toggle button widget parameterized with a specialized list view[3]. This allows for a significant amount of code reuse (not present in the original text list), because the selectable list does not need its own `realize` and `draw` functions, and uses the toggle button's `realize` and `draw` functions instead.

It is important to note that the selectable list would not have been possible without the modified `Interact` abstraction outlined in the latter part of Section 2.7. These changes to the `Interact` abstraction allowed for the identification of modifier key state on `ButtonUp` events in the toggle button. As a result, it became possible to tell if the user depressed a shift or control key while issuing a mouse click to the toggle button.

The selectable list has three different methods of selection: `SingleSelect`, `SelectRange`,

[2]Although it may have been possible to modify the item list widget with this expanded functionality, the author initially believed that the expanded functionality was structurally divergent from the original construction of the item list widget.

[3]The list view was created by the author.

Name	Type	Default	Semantics
background	color	white	background color of widget's window
foreground	color	black	foreground color of widget's window
selectBackground	color	grey	selected background color of widget's window
selectForeground	color	white	selected foreground color of widget's window
font	font	9x15	font of widget's window
borderWidth	int	NoValue	borderWidth surrounding text
halign	HAlign	Left	horizonal alignment of the text items
hpad	int	5	horizonal justification for text (includes borderWidth)
vpad	int	0 (pixels)	inter-button padding

Table 4.1: *Attribute List for Selectable List*

`ToggleSelect`. `SingleSelect` is activated using a single mouse click. It selects only the item which received the click, deselecting all other in the selectable list. `SelectRange`, is activated using a single mouse click while the shift key is depressed. It selects all of the items which are in between the previous mouse click and the current mouse click. Lastly, `SelectToggle` is activated using a single mouse click while the control key is depressed. It toggles the selection of the item being clicked without altering the state of the other items in the selectable list. With regard to multiple modifier keys being depressed when a mouse click occurs, the shift key (or `SelectRange`) maintains precedence.

The selectable list uses `Styles` to add parameterization to the widget. With the `Styles` it is possible to set all of the values outlined in Table 4.1. The selectable list also provides two *modes* which can be specified by the developer. If the application using the selectable list does not warrant multiple items being selected at once, the developer can specify the `SingleSelect` *mode*. Therefore, when using this *mode* it is only possible to select singular files. This *mode* declaration was intentionally left out of `Styles` parameters because it seemed more intuitive to declare the pivot aspect of the selectable list's functionality with the widget's creation.

The selectable list represents a significant improvement over its predecessor. While the primary advantage is clear through the selection scheme, it also abides by most of the new widget conventions[DeB05] and provides extensive parameterization. In combination, these factors provide the user with familiar functionality and the developer with a highly configurable widget.

4.2 File Chooser

Any GUI application that deals with file input and output (IO) generally has need of a file chooser. A file chooser allows users to graphically open directories and select file(s) for IO operations. They are constructed from several low-level widgets, such as buttons, labels, selectable lists, text fields, and scroll bars. Typically, file choosers are considered stand alone widgets, i.e., they are not used as a component in another high-level widget. That being said, file chooser widgets have widespread usage in application development and would be a welcome addition to eXene.

Along with their standard functionality, modern file chooser implementations often come coupled with a *file action*. That is, the file chooser allows for the selection of files and a button to `Open`, `Save`, or `Delete` said files. In this case, the developer need only parameterize the instantiation of the file chooser with the *file action*. The eXene file chooser, however, was created with modularity in mind and does not implement a parameterized *file action*. As a result, the eXene file chooser maintains its purity and can only navigate the file system and select files. It is the duty of the developer to couple the file chooser with a *file action*.

The file chooser implemented by the author has a layout consisting of three tiers and several different widget components. The first tier is the directory navigation tier. It contains a text edit widget (of the directory path)[4], an `Update` button[5], a `Home` button[6], and a `Parent` Button[7]. The second tier is a selectable list wrapped in a scroll port. The selectable list contains a vertical list of all of the names of the sub-directories and files in the current directory. Finally, the third tier consists of a text edit widget (listing all of the selected files), an unlabeled *file action* button (the contents of which are specified by the

[4]The text edit widget reveals the current directory path by default
[5]The Update button will attempt to navigate to a new directory if one is specified in the text edit widget
[6]The Home button will navigate to the home directory
[7]The Parent button will navigate to the parent directory of the current directory

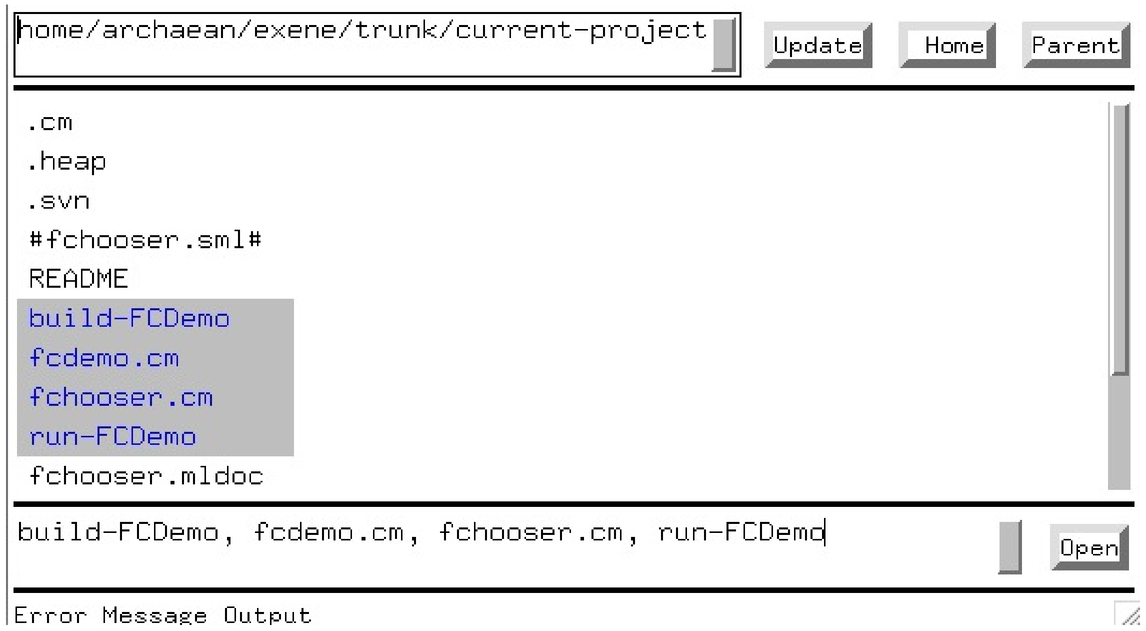

Figure 4.1: *eXene file chooser screen shot*

Name	Type	Default	Semantics
background	color	pink	background color of widget's window
foreground	color	black	foreground color of widget's window
hdir	string	OS.FileSys.getDir()	Home Directory of the Widget

Table 4.2: *Attribute List for File Chooser*

developer or parent widget) and a error label. Together these components allow for the full navigation of the host's file system and selection of files. Additionally, the unlabeled *file action* button is supplied to the developer for customization.

As with the selectable list, the file chooser uses `Styles` to add parameterization to the widget. While the parameterization of the `background` and `foreground` of a widget are standard (shown in Table 4.2), the file chooser has a special attribute `hdir`. The `hdir` specifies what the widget will consider the home directory. It is useful to allow both the developer and the user to potentially specify the home directory because it enhances the usability of the file chooser with no foreseeable side-effects.

Since no functional alternative exists, the file chooser widget is clearly a boon to eXene's widget library. It allows user to graphically navigate the file system, select files, and perform developer defined *file actions*. It may not be as simple to use as file choosers from other GUI toolkits, but it maintains a high level of modularity and takes advantage of eXene's functional nature.

4.3 Box Layout

Any application that contains more than one widget requires the use of a layout widget. Layout widgets are composite widgets, which facilitate the organization and visual dimensions of several widgets grouped within it. The primary layout widget used in eXene is the `box_layout` widget. The `box_layout` widget can organize a set of widgets utilizing the recursively defined datatype `box_item` (shown in Figure 4.2). The `box_item` can be used to encapsulate a widget and represent empty space, as well as vertically or horizontally align a list of `box_item` widgets. This datatype is extremely powerful and allows the `box_layout` widget to align widgets in almost every conceivable way.

There are, however, some deficiencies in the current implementation of the `box_layout` widget. In its initial construction, the `box_layout` widget had the ability to insert, append, delete, and map/unmap within the `box_layout`. However, these functions only allowed for the modification of widgets on the top-level of the `box_layout`. That is, if the `box_layout` consisted of a set of widgets contained as a list of `box_items` within a list of `box_items`, it became impossible to perform modifications on the encapsulated set of widgets[8]. Additionally, the functionality of the `box_layout` widget has long been sparsely documented, making the inner workings of the `box_layout` widget[9] largely unknown to developers.

To rectify these problems, a new `box_layout` widget was created by Jonathan Hoag. While interfacing completely with the functionality of the original `box_layout`, the new `box_layout` emphasizes transparency in the computation of resizing the layout and complete

[8]In this case, the only modifications that can be made are to the box item list which encapsulated them.

[9]Specifically, the calculations which determine the dimensions of the widget during a resize event are difficult to understand.

```
datatype box_item =
  G of (bounds)
| W of (widget * (bounds option) ref)
| HB of (halign * box_item list)
| VB of (halign * box_item list)
```

Figure 4.2: *Box Item datatype*

customization of the layout after it is realized[Hoa06]. In the improved version, the layout is represented as a tree rather than a list. Consequently, modification operations, such as mapping/unmapping, are applicable to individual widgets on any level of the layout rather than just the top level. Moreover, the way in which the `box_layout` widget resizes itself and its children can no longer be viewed as arbitrary. Instead, the specifications for resizing the dimensions of the `layout` widget are all stated and documented explicitly[Hoa06].

The improved `box_layout` widget brings new life to the construction of complex applications. Along with an improved understanding of how the `box_layout` widget functions internally, eXene developers will have functional control over all of the widgets encapsulated in the layout. Overall, the improved `box_layout` widget is a much needed advancement in eXene.

Chapter 5

Conclusion

With its unique design, eXene maintains many advantages over the standard UIMS toolkit. It provides a highly concurrent and computationally effective UIMS toolkit that blends perfectly with SML, a feature rich, higher order, functional language. Nevertheless, many developers may forego the natural advantages gained by using eXene for a seemingly simple and familiar UIMS toolkit. After becoming familiar with the elegance of selective communication and event abstraction in CML, however, one could argue that eXene is far more practical given its harmony with the functional paradigm. Yet, with all its benefits, eXene continues to suffers from a number of deficiencies which have limited its functionality, crippled its usability, and reduced its extensibility. These continued deficiencies give true legitimacy to using eXene alternatives for GUI construction in SML. Therefore, it has been the goal of this thesis to transform eXene into an increasingly functional, highly usable, and extensible UIMS by mitigating the deficiencies present.

The physical structure of a UIMS toolkit can have profound effects on its usage; eXene is no exception. In the design of a directory structure, certain expectations are made by developers about the location and placement of components. Because of the developer's expectations, it is important to group components based on their utility, as well as maintain a level of transparency on each level of the directory tree. Therefore, to improve the physical structure of eXene, it was necessary to consolidate the low level components in the `lib` directory and move various other components to more appropriate and intuitive locations within the directory structure. Although the changes were minimal, they made meaningful contributions to the usability and extensibility of eXene.

The design of the abstraction model can also greatly impact the utility of eXene. Because it defines the API, the abstraction model can either enhance or detract from overall functionality of the toolkit. To provide the best possible development environment, the abstraction model must always resolve to the highest level of modularity and should not remove functionality for simplicity. To improve the abstraction model in eXene, several components that were historically placed in the widget library were modified and moved to more appropriate libraries. Additionally, the `Interact` structure and its sub-components were modified to expand the capabilities stifled in the previous abstraction model. With these changes to the abstraction model, eXene has become increasingly extensible and functional.

With regard to its previous enhancements, another important aspect in the development of eXene is integration. While enhancements that expand functionality can be helpful, some of their benefit cannot be realized without complete integration. As such, the widget focus concept, developed previously, was fully integrated into eXene. Consequently, eXene has gained a substantial amount of hidden functionality.

For the every UIMS, documentation is a pivotal concern. Even as functionality was added to eXene, the lack of documentation rendered it useless to the developer. That is, without extensive, up-to-date documentation, grave mistakes can be made in the design and application of widgets in a GUI application. This can leave a developer disenchanted and thus with a motive to seek an alternative UIMS. To improve the documentation in eXene, an interactive API document was created and populated using ML-Doc. In addition, various scripts were made to keep the documentation and source files consistent with each other. More so than any other change made to eXene, the improved documentation significantly increases its usability and subsequent appeal to the developer.

Finally, from a developer's standpoint, eXene is only as useful as the widgets that it provides. Thus, improvements made to the widget library will undoubtably enhance the usefulness of eXene. With a revised selectable list and layout manager, as well as a new file chooser, eXene has become more functional.

There is no question that much remains to be done to enhance eXene's viability. With further development, eXene's inherent advantages will become even more apparent to developers. With the improvements outlined in this thesis, however, eXene has achieved a level of functionality, usability, and extensibility that is on par with any other UIMS toolkits available for SML and is indeed fully featured. As such, the need for using other UIMS toolkits is drastically reduced and eXene is one step closer to widespread use.

Bibliography

[Blu02] M. Blume. *The SML/NJ Compilation and Library Manager*. Lucent Technologies, Bell Labs, 2002.

[DeB05] D. B. DeBoer. Enhancements to exene. Master's thesis, Kansas State University, 2005.

[DS05] D. DeBoer and A. Stoughton. On the future of exene. `http://www.cis.ksu.edu/~stough/eXene/future.pdf`, 2005.

[Gan95] E. M. Gansner. Notes on the new exene widgets. Included as part of version 1.0 of the eXene distribution, 1995.

[GR93a] E. M. Gansner and J. H. Reppy. *The eXene library manual*. AT&T Bell Laboratories, February 1993.

[GR93b] E. M. Gansner and J. H. Reppy. *The eXene widgets manual*. AT&T Bell Laboratories, February 1993.

[Hoa06] J. Hoag. Exene's layout widget. 2006.

[LN01] K. Larsen and H. Niss. mgtk, sml bindings for gtk+. `http://mgtk.sourceforge.net/`, 2001.

[LW04] C. Luth and B. Wolff. sml tk. `http://www.informatik.uni-bremen.de/~cxl/sml_tk/`, 2004.

[Nye90] A. Nye. *X Protocol Reference Manual*, volume 0. O'Reilly & Associates, Inc., 1990.

[RB07] J. H. Reppy and T. Bourke. *ML-Doc man page*. FreeBSD, 2007.

[Rep99] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[Rep00] J. H. Reppy. *ML-Doc README*. Lucent Technologies, Bell Labs, 2000.

[Rep07] J. H. Reppy. *ML-DOC*, 2007. http://people.cs.uchicago.edu/~jhr/tools/ml-doc.html.

[RMM97] R. Harper R. Milner, M. Tofte and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

Appendix A

linkToMLDOC source code

```
!/bin/bash

# Script to find and link locations of ml-doc/lib/catalog and nsgmls/onsgmls

echo "Locating for ml-doc/lib/catalog"
catalog=‘locate /ml-doc/lib/catalog‘
echo $catalog

if [ -n $catalog ]
then
     echo "ml-doc/lib/catalog found"
else
     echo "Error: ml-doc/lib/catalog not found; you may need to install ml-doc"; exit
fi

echo "Determining if CATALOG exists"
if [ -f CATALOG ]
then
     echo "CATALOG found... editing file to add correct catalog entry"
     sed -i "" -e '/^CATALOG[ ]*\"/ c\
             CATALOG                      \"'$catalog'\"
            ' CATALOG

else
     echo "CATALOG not found; creating from scratch"
     # Create from scratch
     echo "-- Catalog for eXene --" > CATALOG
     echo "ENTITY   %document-entities  \"Entities.sgml\"" >> CATALOG
     echo "CATALOG                      \"$catalog\"" >> CATALOG
     echo "-- Catalog for eXene 2.0 --" >> CATALOG
fi

echo "Determining if nsgmls.source exists"
if [ -f nsgmls.source ]
then
     echo "nsgmls.source does exist... doing nothing"
else
     echo "nsgmls.source does not exist... making link file"
     nsgmls=‘which onsgmls || which nsgmls‘
     if [ -n $nsgmls ]
     then ln -s $nsgmls nsgmls.source
     else echo "Error: Can't find nsgmls or onsgmls;  One of the two are need for ML-Doc"
     fi
fi
```

Appendix B

mkdoc-tool source code

```
#!/bin/bash

# script to generate a MLDoc skeleton file
# and create a symbolic link to said file in original directory

pwd=$PWD
base=${1%\.*}
eXeneDir=`dirname $0`
echo "making $base.mldoc from $1"
rm -f $base.mldoc
cd $eXeneDir/manual
if mkdoc $pwd/$1 > ML-Doc/$base.mldoc;
then chmod 774 ML-Doc/$base.mldoc;
     ln -s $eXeneDir/manual/ML-Doc/$base.mldoc $pwd/$base.mldoc;
else exit 1;
fi;

echo "Attempting to write entry into Entities.sgml"

sigID=`grep 'SIGBODY SIGID' ML-Doc/$base.mldoc | sed -e 's/.*FILE=//' -e 's/>.*//'`
if grep $base.sig Entities.sgml;
then    echo "Entities.sgml contains an entry for $base.sig";
else    if grep SIGID ML-Doc/$base.mldoc;
        then echo "<!ENTITY $sigID SDATA \"$base.sig\">" >> Entities.sgml; echo "Entity added"
        else echo "SIGID not present";
        fi;
fi;
```

Appendix C

Selectable List source code

```
(* selectable-list.sml
   The Selectable list Widget is a slight modification of the button bar
   that uses the button abstraction to simulate the selection of one or
   more items (or buttons) from a list of buttons
*)
signature SELECTABLE_LIST =
sig

structure W: WIDGET

type selectable_list

(* creates a selectable_list widget with items labeled by the supplied strings;
   a boolean value is also required to identify whether or not the selectable
   list will operate in "single select" mode; if the mode is set to true, then
   then only one item from the list can be selected at a time; any type of
   interaction with the list will result in one and only one selection;

   otherwise if the mode is set to false, the items in the list may be
   selected by there different methods:

   SINGLE_CLICK: An item will be selected and all of the previously
                 selected items will become unselected;

   SINGLE_CLICK
     WITH SHIFT: A range of items will be selected starting from the last click
                 to the present click ADDING to the current selection;

   SINGLE_CLICK
     WITH CNTRL: A item will be toggled from the currently selected list; that
                 is, if an item is currently selected it will become unselected
 and if it is unselected int will become selected;

   the widget may be customized using the following resource attributes:

     Name             Type    Default      Semantics
     ------------------------------------------------------------------------
     background       color   white        background color of widget's window
     foreground       color   black        foreground color of widget's window
     selectBackground color   grey         selected background color of widget's window
     selectForeground color   white        selected foreground color of widget's window
     font             font    9x15         font of widget's window
     borderWidth      int     NoValue      borderWidth surrounding text
     hpad             int     5            horizonal justification for text (includes borderWidth)
     vpad             int     0 (pixels)   inter-button padding *)
```

```
val selectableList : (W.root * W.view * W.arg list) -> (string list * bool) -> selectable_list

(* returns the widget of a selectable list *)

val widgetOf : selectable_list -> W.widget

val selEvtOf : selectable_list -> (int * string) list CML.event

(* getSelectedList returns a tuple list of the strings and their corresponding index which
   are currently selected by the widget; the index is constructed based on the initial
   order of the supplied string list *)

val getSelectedList : unit -> (int * string) list

end

structure SelectableList =
struct

open CML

(* This is a unique Toggle Button that is not constructed in the Toggle structure it uses
   a view that was previously unavailable *)
structure SpecialTextToggle = ToggleCtrl (ListView)

(* Structures used more than once requiring a short hand *)
structure A = Attrs
structure TT = ToggleType
structure Q = Quark
structure SV = SyncVar
structure W = Widget
structure I  = Interact

val attr_hpad = Q.quark "hpad"
val attr_vpad = Q.quark "vpad"

(*attributes used specifically for this widget*)
val nativeAttrs =
[([], attr_vpad,       A.AT_Int,   A.AV_Int 0)]

(*attributes used for each individual list item*)
val passedAttrs =
[([], A.attr_background,       A.AT_Color, A.AV_Str "white"),
 ([], A.attr_foreground,       A.AT_Color, A.AV_Str "black"),
 ([], A.attr_font,             A.AT_Font,  A.AV_Str "9x15"),
 ([], attr_hpad,               A.AT_Int,   A.AV_Int 5),
 ([], A.attr_halign,           A.AT_HAlign,A.AV_HAlign W.HLeft),
 ([], A.attr_selectBackground, A.AT_Color, A.AV_Str "grey"),
 ([], A.attr_selectForeground, A.AT_Color, A.AV_Str "blue")
]

(*the type of actions that will modify the selected list*)
datatype clickType = SingleSelect | SingleToggle | AddMulti

(*identity for the type and operation of a selectable list *)
type selectable_list =
{widget : W.widget,
 selEvt : (int * string) list event,
 getSelected : unit -> (int*string) list}

(*varialbe constructor for a selectable list *)
fun selectableList (root, view as (name, style), args) (names, mode) =
let
    (*modified view such that new W.findAttr will works properly *)
    val conView     = (Styles.extendView (name, "selectable-list"),style)
    val attrs       = W.findAttr(W.attrs(conView, passedAttrs, []))

    val bg     = A.getColor(attrs A.attr_background)
    val fg     = A.getColor(attrs A.attr_foreground)
            val font        = A.getFont(attrs A.attr_font)
    val hpad        = A.getInt(attrs attr_hpad)
    val sel_bg      = A.getColor(attrs A.attr_selectBackground)
    val sel_fg      = A.getColor(attrs A.attr_selectForeground)
    val halign      = A.getHAlign(attrs A.attr_halign)

            val nativeAttrs = W.findAttr(W.attrs(conView, nativeAttrs, []))
    val vpad     = A.getInt(nativeAttrs attr_vpad)
    val padGlue     = Box.Glue{nat = vpad, min = vpad, max = SOME vpad}

            (*findClickType unrolls the ToggleType.Toggle datatype and classifies it as a clickType *)
    fun findClickType (TT.Toggle(_,SOME(TT.BtnUp (mbut, modkeys)))) : clickType option =
        if mode
then SOME SingleSelect
else
if I.shiftIsSet modkeys
then SOME AddMulti
else
if I.cntrlIsSet modkeys
then SOME SingleToggle
else SOME SingleSelect
      | findClickType _ = NONE

            (*isToggle unrolls the ToggleType.Toggle datatype and determines if there is a BtnUp msg*)
    fun isToggle (TT.Toggle (_,SOME(TT.BtnUp _)))  = true
      | isToggle _           = false

            (*item_data is the internal representation for each of the labeled toggle buttons
      each item has a label containing its name, a item which is the event itself and
      a upEvt which recieves the button click information*)
            type item_data =
      {  lab   : string,
 item   : TT.toggle,
 upEvt    : TT.toggle_act event}

            (*makeItem given a string constructs an instance of an item_data*)
    val makeItem : string -> item_data =
        fn lab =>
     let val itemArgs =
[([], A.attr_label,       A.AV_Str      lab),
 ([], A.attr_background,     A.AV_Color    bg),
 ([], A.attr_foreground,    A.AV_Color    fg),
 ([], A.attr_selectBackground,   A.AV_Color    sel_bg),
 ([], A.attr_selectForeground,   A.AV_Color    sel_fg),
 ([], A.attr_font,          A.AV_Font     font),
 ([], A.attr_halign,             A.AV_HAlign   halign),
 ([], attr_hpad,           A.AV_Int      hpad)
]
    val item =  SpecialTextToggle.toggle (root, view, itemArgs)
    (* the flushEvt is currently not used by this widget *)
    val (valEvt, flushEvt) = FilterEvt.filterEvt isToggle (TT.evtOf item)
    val itemData = {lab      = lab,
    item     = item,
    upEvt    = valEvt}

in itemData
end

            (*itemsToBoxes : item_data list -> Box.WBox list
      takes in a list of items and constructs a vertical list of widgets*)
    fun itemsToBoxes nil       =  nil
```

```
      | itemsToBoxes [b]       =
                  [Box.WBox(TT.widgetOf b)]
              | itemsToBoxes (b :: bs) =
                  Box.WBox(TT.widgetOf b) :: padGlue :: itemsToBoxes bs


    val itemDatas = map makeItem names
    val items   = map #item itemDatas
    val box   = Box.VtLeft(itemsToBoxes items)

    val layout = Box.layout(root, view, args) box
    val widget = Box.widgetOf layout

            val selCh : (int * string) list chan = channel()

            (* the rqstChan is used to to make requests to the server thread *)
            val rqstChan: ((int list) SV.ivar)chan = channel()

            (* indexedData constructs (int * item_data) list effectively indexing the already
       constructed itemDatas *)
            val indexedDatas = let
        fun makeIndex (num, []) = []
  | makeIndex (num, item::tl) = [num] @ makeIndex (num+1, tl)
in ListPair.zip(makeIndex (0,itemDatas), itemDatas)
end

            (* doSingleToggle:  (selList: int list * index: int) -> int list
       doSingleToggle adds the index to selList if it is not present and removes
       it if it is present *)
            fun doSingleToggle (selList,index) =
        if (List.exists (fn x => x=index) selList)
then (List.filter (fn x => not(x=index)) selList)
else ([index]@selList)

            (* doSingleSelect: (index:int)->  int list *)
    fun doSingleSelect (index) =
        [index]

            (* makeMultiList: (last: int option * index : int) -> int list *)
            fun makeMultiList (NONE, index) = [index]
              | makeMultiList (SOME(last), index) =
        let
fun makeIndex top bottom =
    if top >= bottom
    then (makeIndex (top-1) bottom)@[top]
    else []
in
                    if index >= last
    then makeIndex index last
    else makeIndex last index
end

            (* doAddMulti: (last: int option * selList: int list * index: int) -> int list *)
    fun doAddMulti (last, selList, index) =
        let
val addList = makeMultiList (last, index)
fun addUniqueItemsToList [] = []
  | addUniqueItemsToList(h::tl) =
        if (List.exists (fn x => x=h) selList)
then addUniqueItemsToList tl
else [h]@(addUniqueItemsToList tl)
in
    selList@(addUniqueItemsToList addList)
                end
            (* greaterThan : (int * int) -> bool *)
    val greaterThan = fn (x, y) => x > y

            (* doAction: (clickType option * int option * int list * int) -> int list *)
            fun doAction (NONE,last, selList, index) = selList
      | doAction (SOME(SingleToggle), last, selList, index) =
       ListMergeSort.sort greaterThan (doSingleToggle (selList, index))
      | doAction (SOME(SingleSelect), last, selList,index) = doSingleSelect (index)
      | doAction (SOME(AddMulti), last, selList, index ) =
       ListMergeSort.sort greaterThan (doAddMulti (last,selList, index))

    fun getIndexedItem num =
                let
            fun grab [] = []
                      | grab ((index,{lab,...}:item_data)::tl) =
                  if num = index
                  then [(num,lab)]
                  else grab tl
        in grab indexedDatas
                end

    fun getIndexedItems [] = []
      | getIndexedItems (hd::tl) = (getIndexedItem hd)@(getIndexedItems tl)

   (* unCoverClickAndServ (int option * int list * ( )*)
    fun unCoverClickAndServ (last,selList, serv,(index,{upEvt, ...}:item_data)) =
        [wrap(upEvt, fn click =>
let
val selList' = (doAction((findClickType click),
 last,
 selList,
 index
)
)
val evt = sendEvt(selCh, getIndexedItems(selList'))
fun loop () =
    select
    [wrap(evt, fn () => serv (SOME(index)) selList')]
in loop ()(*serv (SOME(index)) selList'*)
end
     )
]

            fun makeSelect (last,selList,serv,[]) = []
      | makeSelect (last,selList,serv,indexedData::tl) =
           (unCoverClickAndServ (last,selList,serv,indexedData))
  @ (makeSelect (last,selList,serv,tl))

    fun syncButtonStateWithList selList =
        let
    fun syncList [] = ()
      | syncList ((index,{item,...}:item_data)::tl) =
           if (List.exists (fn x => x = index) selList)
   then (TT.setState(item,true) ; syncList tl)
                  else (TT.setState(item,false); syncList tl)
in
syncList indexedDatas
end


            fun server last selList = (syncButtonStateWithList selList;
                            select ([wrap(recvEvt rqstChan, fn iVar => (SV.iPut(iVar, selList);
         server last selList ))]@
               (makeSelect (last,selList,server,indexedDatas))
      )
     )

    fun getSelectedList () =
     let

    val iVar : (int list) SV.ivar = SV.iVar()

     in send (rqstChan, iVar); getIndexedItems (SV.iGet iVar)
     end

    in
     spawn(fn () => server NONE []);
     {widget = widget,
 selEvt = recvEvt selCh,
       getSelected = getSelectedList}
    end


fun widgetOf({widget,...}:selectable_list) = widget
fun selEvtOf({selEvt,...}:selectable_list) = selEvt
fun getSelectedList({getSelected,...}:selectable_list) = getSelected ()
end;
```

Appendix D

File Chooser source code

```
(*fchooser.sml*)

signature FILE_CHOOSER =
sig

structure W: WIDGET

type filechooser

(* a filechooser widget allows you to browse through the file system and
   perform operations on that file system.  It is always in an active mode.
   It requires a root, view, and args list as well as a button that will
   provide the primary operation for the file chooser and a boolean value
   that specifies the selection mode (true - Single Selection
      false - Multiple Selection).

   The widget may be customized using the following resource attributes:

     Name          Type    Default             Semantics
     ----------------------------------------------------------------------
     background    color   pink                background color of widget's window
     foreground    color   black               foreground color of widget's window
     hdir          string  OS.FileSys.getDir() Home Directory of the Widget
*)

val fileChooser : (W.root * W.view * W.arg list) -> (Button.button * bool) -> filechooser

(*returns widget of a filechooser*)
val widgetOf : filechooser -> W.widget

(*getCurrentDirectory returns the current Directory of the filechooser*)
val getCurrentDirectory : filechooser -> string

(*getCurrentText returns the current text (the selected files and anything that
  the user has typed in*)
val getCurrentText : filechooser -> string

(*getCurrentSelection returns a list of the currently selected files and
  with a corresponding directory identifier. i.e. true  -> directory
  false -> file *)
val getCurrentSelection : filechooser -> (string*bool) list

(*setDirectory is a function that takes in a string and attempts to
  sets the filechooser's current directory to that string*)
val setDirectory : filechooser -> string -> unit
```

```
(*setError is a function that takes in a string and sets the text of the
  error label to the string *)
val setError : filechooser -> string -> unit

end

structure FileChooser =
struct

open CML Widget Geometry Interact

structure Q = Quark
structure A = Attrs
structure W = Widget
structure SV = SyncVar

val attr_hDir = Q.quark "hDir"
val homeDir = OS.FileSys.getDir()

val attrs =
       [([], A.attr_background,  A.AT_Color,  A.AV_Str "white"),
        ([], A.attr_foreground,  A.AT_Color,  A.AV_Str "black"),
([], attr_hDir,          A.AT_Str,  A.AV_Str homeDir)]

datatype setRequest   = SetDirectory of string
      | SetError of string

(*internal state consists of a directory list, a file list, the current
  SelectableList and the current navigational directory*)
type state =     {dirList      : string list,
 fileList     : string list,
      selList      : SelectableList.selectable_list,
      currentDir   : string}

(*the filechooser type consist of a widget, and 5 functions that allow for
  informational retrieval and state modification*)
type filechooser      = {widget           : W.widget,
        currentDirectory : unit -> string,
 currentText      : unit -> string,
        currentSelection : unit -> (string*bool) list,
        setDirectory   : string -> unit,
 setError         : string -> unit}

fun fileChooser(root, (name, style), args) (actionBttn, selectMode) =
let val view = (Styles.extendView (name, "file-chooser"), style)
    val attrs  = W.findAttr(W.attrs(view,attrs,[]))
    val bg = A.getColor(attrs A.attr_background)
    val fg = A.getColor(attrs A.attr_foreground)
    val hDir = A.getString(attrs attr_hDir)

            val focusMgr = FocusMgr.mkFocusMgr ()

    (*Widget Construction*)
    val dirPath  = TextView.textView (root, view, [([], A.attr_maxLines, A.AV_Int 1),
   ([], A.attr_rows,   A.AV_Int 1)])
     ("")
    val _ = TextView.tvSetShowScrollbars (dirPath) (false)
    val dirPathFF = FocusFrame.focusframe (root, view, [])
  ((TextView.widgetOf dirPath),(TextView.focusableOf dirPath))

           (* val _ = FocusMgr.addFocusable focusMgr (FocusFrame.focusableOf dirPathFF)*)

    val updArgs =
        [([], A.attr_label,A.AV_Str "Update"),
 ([], A.attr_borderWidth,A.AV_Int 5)]
    val updBttn = Button.textBtn(root,view, updArgs)
    val updEvt  = Button.evtOf updBttn

    val homeArgs =
     [([], A.attr_label,  A.AV_Str " Home "),
 ([], A.attr_borderWidth,A.AV_Int 5)]
    val homeBttn = Button.textBtn(root, view, homeArgs)
    val homeEvt  = Button.evtOf homeBttn

    val prevArgs =
     [([], A.attr_label,  A.AV_Str "Parent"),
 ([], A.attr_borderWidth,A.AV_Int 5)]
    val prevBttn = Button.textBtn(root, view, prevArgs)
    val prevEvt  = Button.evtOf prevBttn

    (*The "error label" is actually a button with no border width.
      it was implemented as such to circumvent the problems of rigidity
      caused by the label widget*)
    val errArgs  =
     [([], A.attr_label,  A.AV_Str "Error Message Output"),
 ([], A.attr_borderWidth,A.AV_Int 0),
                 ([], A.attr_halign,            A.AV_HAlign W.HLeft)]
            val errorLabel = Button.textBtn(root, view, errArgs)


            val listLayout = Box.layout (root, view, [])
        (Box.VtLeft[Box.Glue{nat=100,min=100,max=SOME(100)}])

    val scrollSelList = ScrollLayout.mkSBLayout root
      { widget = (Box.widgetOf listLayout),
     hsb    = NONE,
     vsb    = SOME {sb=(Scrollbar.widgetOf (Scrollbar.mkVScrollbar root {color=NONE, sz=16})), pad=0, left=false}}

    val scrollLayout  = Box.layout (root, view, [])
        (Box.VtLeft[Box.WBox(ScrollLayout.widgetOf scrollSelList)])

    val fileList = TextView.textView (root, view, [([], A.attr_maxLines, A.AV_Int 1),
   ([], A.attr_rows,   A.AV_Int 1)])
     ("")
    val _ = TextView.tvSetShowScrollbars (fileList) (false)
    val fileListFF = FocusFrame.focusframe (root, view, [])
   ((TextView.widgetOf fileList),(TextView.focusableOf fileList))

    (*val _ = FocusMgr.addFocusable focusMgr (FocusFrame.focusableOf fileListFF)*)

            val dividerArgs = [([], A.attr_color, A.AV_Str "black"),
       ([], A.attr_width, A.AV_Int 3)]

    (*horzDivider: unit -> W.widget
      used to make all of the horizonal divider in the widget construction*)
    fun horzDivider () = Divider.horzDivider(root,view,dividerArgs)

            val padGlue = Box.Glue{nat = 10,min = 10, max = SOME(10)}

    val topSect  = Box.layout (root, view, [])
           (Box.HzCenter[Box.WBox(FocusFrame.widgetOf dirPathFF),
    padGlue,
    Box.WBox(Shape.mkRigid(Button.widgetOf updBttn)),
    padGlue,
    Box.WBox(Shape.mkRigid(Button.widgetOf homeBttn)),
    padGlue,
    Box.WBox(Shape.mkRigid(Button.widgetOf prevBttn))])
```

```
    val midSect  = Box.layout (root, view, [])
           (Box.HzCenter[Box.WBox(Box.widgetOf(scrollLayout))])

    val btmSect  = Box.layout (root, view, [])
           (Box.HzCenter[Box.WBox(FocusFrame.widgetOf fileListFF),
           padGlue,
    Box.WBox(Shape.mkRigid(Button.widgetOf actionBttn))])

            val errSect  = Box.layout (root, view, [])
      (Box.VtLeft[Box.WBox(Button.widgetOf errorLabel)])

            val padGlue = Box.Glue{nat = 4,min = 4, max = SOME(4)}

    val overall  = Box.layout (root, view, [])
           (Box.VtLeft  [padGlue,
           Box.WBox(Box.widgetOf(topSect)),
    padGlue,
    Box.WBox(horzDivider ()),
    padGlue,
    Box.WBox(Box.widgetOf(midSect)),
    padGlue,
    Box.WBox(horzDivider ()),
    padGlue,
    Box.WBox(Box.widgetOf(btmSect)),
    padGlue,
    Box.WBox(horzDivider ()),
    padGlue,
    Box.WBox(Box.widgetOf(errSect))])

    val widget = (Box.widgetOf(overall))
    (*end widget construction*)

    (*racErrorLabel: string -> unit
      racErrorLabel stands for remove and create error label
              since it is impossible to change the text of a button racErrorLabel
              recreates the button with a new message*)
    fun racErrorLabel msg =
let  val errArgs  =
     [([], A.attr_label,  A.AV_Str msg),
   ([], A.attr_borderWidth,A.AV_Int 0),
                   ([], A.attr_halign,            A.AV_HAlign W.HLeft)]

                  val errorLabel = Button.textBtn(root, view, errArgs)
in
     (Box.delete errSect [0];
      Box.insert errSect (0, [Box.WBox(Button.widgetOf errorLabel)]);
 Box.mapBox errSect [0])
end

            (*racDF: state -> state
      racDF stands for remove and create Directorys and Files
      racDF takes in the internal state and attempts to open and read
              the currentDir.  Once that is complete its reconstructs the
      selectable list and remounts it on the list layout*)
            fun racDF(state:state) =
     let
val _ = OS.FileSys.chDir (#currentDir state)
val dirStream = OS.FileSys.openDir(#currentDir state)

val fileAndDirList =
    let
        fun createList dList fList =
     (case OS.FileSys.readDir(dirStream) of
  NONE       => (dList, fList)
| SOME file   => if (OS.FileSys.isDir(file))
 then (createList (dList@[(file)]) fList)
 else (createList dList (fList@[(file)])))
    in createList [] []
    end

(* createSelectableList: (string list * string list) -> state
 *)
fun createSelectableList(dList, fList) =
    let
        val _ = OS.FileSys.closeDir (dirStream)
     val selectableList = SelectableList.selectableList (root,view,[])
   (dList@fList,selectMode)

                        (*val scrollSelList = ScrollLayout.mkSBLayout root
      { widget = (SelectableList.widgetOf selectableList),
     hsb    = NONE,
     vsb    = SOME {sb=(Scrollbar.widgetOf
  (Scrollbar.mkVScrollbar root
   {color=NONE,
    sz=16}
  )),
  pad=0,
  left=false
   }
    }*)


(*fun sleep n = CML.sync(CML.timeOutEvt(Time.fromMilliseconds n))*)
    in
     ((if (List.null( (#dirList state)@(#fileList state)))
  then ()
  else (Box.delete listLayout [0](*; sleep 100*)));

 Box.insert listLayout (0, [Box.WBox(SelectableList.widgetOf selectableList)]);
 Box.mapBox listLayout [0];
 racErrorLabel "Error Message Output";
 TextView.setString dirPath (#currentDir state);
 TextView.setString fileList "";
 {currentDir = #currentDir state,
  dirList    = dList,
  fileList   = fList,
  selList    = selectableList})
    end
in
    createSelectableList(fileAndDirList)
end


    (*channel for getting the selection from the selectable list*)
    val getSelChan : ((string*bool) list SV.ivar) chan = channel ();
    (*channel for getting the current directory from the state*)
    val getDirChan : (string SV.ivar) chan = channel ();
    (*channel for getting the current text from the text edit widget*)
    val getTexChan : (string SV.ivar) chan = channel ();
            (*channel for setting the directory and setting the error message *)
    val setReqChan : (setRequest) chan = channel ();

    (*loop: state -> unit
      main loop*)
            fun loop(state:state) =
     let
    val _ = OS.FileSys.chDir (hDir)
    val parentDir = OS.Path.getParent (OS.Path.toUnixPath(#currentDir state))
    val updateDir = TextView.getString dirPath
    val selEvt    = SelectableList.selEvtOf (#selList state)

    (*function used to extract the file/directory names and identify their corresponding types*)
            fun extract ((x,m)::tl) = ([(m,OS.FileSys.isDir ((#currentDir state)^"/"^m))] @ (extract tl))
              | extract []          = []

                    (*handle the press of the home button
      change the current directory to the home directory*)
    fun handleHome (Button.BtnUp _)   = let
                val state' = {currentDir = hDir,
   dirList = #dirList state,
fileList = #fileList state,
selList = #selList state}
             in
                        loop(racDF(state'))
                  end

      | handleHome _                  = loop(state)

    (*handle the press of the parent button
      attempts to change the directory to the parent directory
             failure to do so raises an error message
     *)
    fun handleParent (Button.BtnUp _) = ((let
                val state' = {currentDir = parentDir,
   dirList = #dirList state,
fileList = #fileList state,
selList = #selList state}
        in
   loop(racDF(state'))
   end)
handle OS.SysErr (msg1, msg2) =>
(racErrorLabel msg1;
loop(state)))


      | handleParent _        = loop(state)

                    (*handle the press of the update button
      attempts to change the directory to the text in the dirPath
      failure to do so raises an error message
     *)
    fun handleUpdate (Button.BtnUp _)  =((let
                val state' = {currentDir = updateDir,
   dirList = #dirList state,
fileList = #fileList state,
selList = #selList state}
         in
  loop(racDF(state'))
         end)

 handle OS.SysErr (msg1,msg2) =>
  (racErrorLabel msg1;
 loop (state)))

      | handleUpdate _        = loop(state)

                    (*handle a selection made on the selectable list
     *)
    fun handleSelect selection        =(let     (*unpack:(int*string) list -> string
   unpack constructs a string of the list of
   files selected and set them as the text of
   the fileList widget*)
         fun unpack [] = ""
    | unpack ((_,hd)::[]) = hd
    | unpack ((_,hd)::tl) = hd^", "^(unpack tl)
    val sel = unpack selection
 in
(TextView.setString fileList sel;
 loop(state))
 end)

    (**)
    fun handleChangeDir name        =((let
                val state' = {currentDir = name,
   dirList = #dirList state,
fileList = #fileList state,
selList = #selList state}
 in
  loop(racDF(state'))
 end)

 handle OS.SysErr (msg1,msg2) =>
  (racErrorLabel msg1;
loop (state)))

    fun handleGetSelection (iVar)   =   (SV.iPut(iVar, extract (SelectableList.getSelectedList (#selList state)));
      loop(state))

    fun handleGetDirectory (iVar)   =   (SV.iPut(iVar,(#currentDir state));
      loop(state))

    fun handleGetText    (iVar)   =   (SV.iPut(iVar,(TextView.getString fileList));
 loop(state))

    fun handleSetRequest (req)        = (case req of
       SetDirectory x =>
   ((let

                val state' = {currentDir = x,
         dirList = #dirList state,
      fileList = #fileList state,
      selList = #selList state}
   in
   loop(racDF(state'))
   end)

   handle OS.SysErr (msg1,msg2) =>
     (racErrorLabel msg1;
   loop (state)))
| SetError x =>
  (racErrorLabel x; loop(state)))



    in select
     [wrap(homeEvt, handleHome),
 wrap(prevEvt, handleParent),
 wrap(updEvt,  handleUpdate),
 wrap(selEvt,  handleSelect),
 wrap(recvEvt getDirChan, handleGetDirectory),
 wrap(recvEvt getTexChan, handleGetText),
 wrap(recvEvt getSelChan, handleGetSelection),
 wrap(recvEvt setReqChan, handleSetRequest)]
    end

fun getCurrentDirectory() =
    let val iVar : string SV.ivar = SV.iVar ()
    in  (send(getDirChan, (iVar));
        (SV.iGet iVar))
    end

fun getCurrentText() =
    let val iVar : string SV.ivar = SV.iVar ()
    in   (send(getTexChan, (iVar));
 (SV.iGet iVar))
    end

fun getCurrentSelection() =
    let val iVar : (string * bool) list SV.ivar = SV.iVar ()
    in (send(getSelChan, (iVar));
       (SV.iGet iVar))
    end

fun setDirectory s = send(setReqChan, (SetDirectory(s)))

fun setError s     = send(setReqChan, (SetError(s)))

        in

spawn(fn () => loop(racDF{currentDir = hDir,
          dirList    = [],
          fileList   = [],
          selList    = SelectableList.selectableList (root,view,[]) ([],false)}));

    {widget           = widget,
     currentDirectory = getCurrentDirectory,
     currentText      = getCurrentText,
     currentSelection = getCurrentSelection,
     setDirectory     = setDirectory,
     setError         = setError}
end

fun widgetOf({widget,...} : filechooser) = widget
fun getCurrentDirectory({currentDirectory,...} : filechooser) = currentDirectory ()
fun getCurrentText({currentText,...} : filechooser) = currentText ()
fun getCurrentSelection({currentSelection,...} : filechooser) = currentSelection ()
fun setDirectory({setDirectory,...} : filechooser) = setDirectory
fun setError({setError,...} : filechooser) = setError

end
```

Appendix E

File Chooser Demo source code

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

structure FCDemo : sig

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

  end = struct

    structure EXB = EXeneBase
    structure S = Styles
    structure A = Attrs

(* set up the option spec table. *)
val optSpec =
 [(S.OPT_NAMED("help"), "-help",   S.OPT_NOARG("on"),  A.AT_Bool),
  (S.OPT_NAMED("help"), "-nohelp", S.OPT_NOARG("off"), A.AT_Bool),
  (S.OPT_NAMED("res"),  "-res",    S.OPT_RESARG,       A.AT_Str),
  (S.OPT_NAMED("skip"), "-skip",   S.OPT_SKIPARG,      A.AT_Str),
  (S.OPT_NAMED("ign"),  "-ignore", S.OPT_SKIPLINE,     A.AT_Str),
  (S.OPT_RESSPEC("*background"), "-bg", S.OPT_SEPARG,  A.AT_Str),
  (S.OPT_RESSPEC("*foreground"), "-fg", S.OPT_SEPARG,  A.AT_Str),
  (S.OPT_RESSPEC("*borderColor"),"-bc", S.OPT_SEPARG,  A.AT_Str)]

(* set up application resource defaults. *)
val appResources =
   ["*background: white",
    "*foreground: black"]

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

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

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

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

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

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

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

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

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


    val quitBttn = Button.textBtn (root, view, [([], Attrs.attr_label, Attrs.AV_Str "Quit"),
        ([], Attrs.attr_borderWidth, Attrs.AV_Int 5)])
    val quitEvt  = Button.evtOf quitBttn

    val openBttn = Button.textBtn (root, view, [([], Attrs.attr_label, Attrs.AV_Str "Open"),
        ([], Attrs.attr_borderWidth, Attrs.AV_Int 5)])
    val openEvt  = Button.evtOf openBttn

    val attr_singleSelect = Quark.quark "singleselect"
    val fileChooser = FileChooser.fileChooser (root, view, []) (openBttn, false)

    val layout =
        Box.layout (root, view, []) (Box.VtCenter[
    Box.WBox(Button.widgetOf quitBttn),
    Box.WBox(FileChooser.widgetOf fileChooser)
        ])

    val shellArgs =
        [([], Attrs.attr_title, Attrs.AV_Str "SelectableText Widget Demo"),
         ([], Attrs.attr_iconName, Attrs.AV_Str "demo-res")]
    val shell = Shell.shell (root, view, shellArgs) (Box.widgetOf layout)

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

    fun printList [] = (TextIO.print "\n")
      | printList ((hd,_)::[]) = (TextIO.print (hd); printList [])
      | printList ((hd,_)::tl) = (TextIO.print (hd^", "); printList tl)
```

```
    fun findNumOfFD selection =
let
fun aux ([],(dir,fil)) = (dir,fil)
  | aux ((_,t)::tl,(dir,fil)) = aux (tl,(if t then (dir+1,fil) else (dir,fil+1)))
in
aux (selection,(0,0))
end

    fun openFiles selection = (TextIO.print "Opening Files\n";printList selection)

    fun openDirs [] = FileChooser.setError fileChooser "No Directory to Open"
      | openDirs ((selection,_)::[]) = FileChooser.setDirectory fileChooser
    ((FileChooser.getCurrentDirectory fileChooser)
      ^"/"^selection
    )
      | openDirs (selection::tl) = (FileChooser.setError fileChooser  "Cannot open multiple directories")

    fun openDirOrFile selection =
let val (dir,fil) = findNumOfFD selection
    fun choose (0,n) = openFiles selection
      | choose (n,0) = openDirs selection
      | choose _ = (FileChooser.setError fileChooser "Cannot select directories AND files")
in choose (dir,fil)
end


    fun loop():unit =
        let
fun handleOpen (Button.BtnUp _)   = (openDirOrFile (FileChooser.getCurrentSelection fileChooser); loop ())
          | handleOpen (_)           = loop()
fun handleQuit (Button.BtnUp _)   = (TextIO.print " [demo-res quitting]\n"; quit())
          | handleQuit (_) = (loop())
        in CML.select
           [CML.wrap(openEvt, handleOpen),
    CML.wrap(quitEvt, handleQuit),
            CML.wrap(cmEvt, quit)
          ]
        end
    in
        Shell.init shell;
        loop()
    end

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

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

end
```