EX11 – A GUI in a Concurrent Functional Language

Joe Armstrong
SICS
Box 1263
SE-164 29 Kista Sweden
+46 8 633 1538
joe@sics.se

ABSTRACT
In this paper, I describe how GUIs can be made from collections of communicating parallel processes. The paper describes EX11 which is an Erlang binding to the X protocol. I describe the X windows programming model and show how X protocol messages can be naturally mapped onto Erlang messages. The code to perfom this mapping makes extensive use of the Erlang bit syntax and as such provides a good example of the use of the bit syntax to implement a reasonably complex protocol. I give code examples which make use of the EX11 widget library and show how the widget library itself is implemented.

Categories and Subject Descriptors
D.1.3 [**Concurrent Programming].** D.2.2 **[Design Tools and Techniques]**: *User interfaces*

General Terms
Algorithms, Design.

Keywords
GUI, Erlang, X windows, X protocol, concurrency.

1. INTRODUCTION
This paper describes a system for programming GUIs called *EX11*. EX11 is an Erlang [1] binding for the X window system [2]. With EX11 you can easily program complex GUIs. EX11 models all widgets as concurrent processes. This results in extremely compact GUI programs which are simple to program and easy to understand.

EX11 is written in 100% Erlang and talks *directly* to the X server using the the X Protocol [3].

EX11 has two parts - a low-level library which is used to communicate with the X-server and a high level widget set intended for GUI programming. The low-level part of the implementation deals directly with the X protocol itself, talking directly to the X windows server through either a Unix domain socket or through one of the X windows control ports. The low-level software has to encode and decode packets according to the X protocol specification and it makes extensive use of Erlang binaries for bit level manipulation of the protocol packets. Later in the paper we will see how Erlang binaries can be used for packing and unpacking the X protocol packets. Fortunately there is an almost one to one correspondence between the packets as defined in the X Protocol specification and the Erlang code used to create the packets. This fact greatly simplified the implementation of the system.
The low-level libraries provide about as much functionality as the Xlib library [4,5] – but the code is very much shorter.

The higher level widget library provides an abstraction layer that makes it easy to program complex widgets. This provides much of the functionality of a high level widget set such as GTK or one of the many X toolkits.

Interestingly the entire system is written in Erlang and communicates directly with the X windows server. The system appears to be at least an order of magnitude smaller than equivalent GUI system programmed in sequential imperative languages and in terms of performance it is not noticeably slower than GUI systems programmed using conventional widget programming libraries.

In the remainder of this paper I will describe the X widows programming model, and how this can be mapped onto a collection of parallel processes. I will describe how the X protocol can be programmed in a clear manner in Erlang and how high-level widgets can be constructed as collections of parallel processes. The paper has a number of programming examples which show how complex behavioral patterns can be build from a number of well-chosen primitives. Finally I compare the performance and code size of the system with more conventional systems.

Erlang'04, September 22, 2004, Snowbird, Utah, USA.

2. THE X11 PROGRAMMING MODEL
To start with we observe that the X windows programming model is *incredibly complicated.* Even the simplest "hello world" program is a nightmare of complexity. The first thing that happens when you see *hello world* written directly using Xlib is that you want to roll over and die. The simple act of creating a window and writing a line of text takes 177 lines of code; even the description of hello-world is a heavy 35 pages of description in the Xlib programming manual. If you want to delve deeper into the system all you have to do is read the X Protocol manual (458) pages. The Xlib manuals weigh in at a mighty 1962 pages. Now all of this is pretty complicated stuff, so to simplify things you might like to try reading the *X11 toolkit intrinsics manual* (674 pages) [6] or even the *The X window system in a nutshell* [7] (424 pages.) [1]

In all, the X window system is documented in nine books and a total of 8350 pages of text. At this point most sensible people give up and choose either a high level widget library (such as

[1] Pretty big nutshell!

GTK) or a GUI building program (such as Visual Basic) or Visual C++.

The GUI builder writes the program for you so you don't have to deal with this horrendous complexity. The less sensible person (ie the author) decides to read the manuals and re-implement as much as possible from scratch.

If you want to program a graphics system you have to start somewhere. You can start at a *really* low-level and implement a frame buffer on top of a bit-mapped graphics system or you can start at a somewhat higher level. I chose to start at the X protocol level.

The X protocol itself is a master of simplicity. X windows are created and destroyed by sending and receiving messages to a Unix domain socket or to a TCP socket. All objects within the window are also controlled by sending messages to the control socket.

In the X programming model to do something to a window you send a message to the socket associated with the window. When an event happens within a window the X server writes a message to the socket associated with the window. This is an incredibly simple idea – there are no shared variables between the client and server and the client and server can be physically located on different machines.

The underlying simplicity of the X protocol is entirely hidden from the user by a vast set of libraries which essentially do one thing – the libraries hide concurrency from the user. This the *raison d'être* for the complexity of the set of libraries built on top of the X11 protocol.

The vast majority of conventional programming languages are sequential – if you want concurrency you use some threads package or fork off an operating system process. You can get concurrency if you want it but you have to use operating system processes. Operating system process and threads are appallingly difficult to program and extremely costly. Creating a thread or a process is an extremely costly operation.

Because conventional programming languages are sequential the widget libraries for GUI programming do virtually everything in terms of call-back routines. A call back function is a function that is evaluated when a certain event happens. Unfortunately the scheduling of a callback is undefined, so if a GUI system has set up a number of call backs in a number of different places and then if things start happening within a window, it becomes almost impossible to say what will happen. The exact order in which these callbacks occur is undefined. What is known is that the callback routines will eventually be evaluated, but the exact order in which they are evaluated is undefined. Now usually this doesn't matter – but in the case where the callback routines have to manipulate shared resources, this provides a fertile ground for errors to grow.

The reason why GUI programming is difficult has to do with concurrency. GUI's are by their nature concurrent. Imagine a GUI having two clock faces; one showing the time in Sweden and the other the time in the USA. We expect that whatever is happening within the two clock widgets to happen concurrently, one clock should not stop if the other clock is updated. Emulating this concurrency using a single sequential process and a collection of callbacks is unnatural, leads to ugly code and is highly error prone. The sheer size of the program is daunting since it has to be written in an unnatural manner which does not follow the natural concurrency patterns dictated by the application.

Since Erlang is a concurrent programming language, it is natural to map each concurrent activity in a GUI onto a different Erlang process. Suddenly the programming model becomes simple since there is no impedance mismatch between the concurrent structure of the GUI and the manner in which it is implemented.

It is interesting to note how often problems with concurrency lead to inconsistent behavior in a GUI – for example a clock might stop being updated during the time in which you interact with a drag-down menu. If you are in the middle of filling in a form in a pop-up window you cannot temporarily suspend what you are doing and do something else in the window. These inconsistencies are almost invariably due to the sequential way in which the application is programmed.

2.1 THE EX11 PROGRAMMING MODEL Widows and widgets

1. All objects placed within a top-level window are instances of a widget. All widgets are implemented as one or more concurrent processes. Sometimes a widget is controlled by a single process (called the controlling process of the widget), in other cases the widget is controlled by more than one process; in the latter case one of these processes is designated the controlling process for the widget.

2. All changes to a widget are made by sending messages to the controlling process for the widget. All events occurring in the widget result in messages being sent to the controlling process for the widget. As far as the user is concerned they can think of the widget as being controlled by a single process. If the widget is actually controlled by several processes then the user will only be aware of the top level controlling processes and all the other processes will be hidden from the user. If the widget is destroyed then all processes associated with the widget die automatically. If a software error occurs in the code used to implement the widget, then the widget will be removed from the screen and an appropriate error message written to the error log. Data associated with a widget can be read by sending a message to a widget and waiting for a reply message.

Note that this programming model corresponds exactly to the X windows programming model when viewed at the protocol level. There is no impedance mismatch between the semantics of the interface to X (which is a pure message-passing protocol based interface) and the semantic of how a widget within a window behaves.

The next section shows a few examples of this.

3. EXAMPLES

The first example of EX11 (Figures 1 and 2) shows how to create a window with two labels, two entries and a button.

Widgets are created with the syntax:

```
Widget = swWidgetName:make(Arg1,....ArgN)
```

So for example:

```
Entry1 = swEntry:make(...)
```

Creates an entry.

The arguments **Arg1,..., ArgN** specify the initial state of the widget.

If we have created an entry **E** then we can set the text in the entry to **S** with the following command:

```
E ! {set, S}
```

The syntax **P ! M** means send the message **B** to the process **P**.

To read an entry **E** and assign the value to a variable **Var** we evaluate the expression:

```
Val = E !! read
```

The syntax **P !! R** is used to denote a remote procedure call. A message **R** is sent to **P** and the sending processes waits for a message to come back from **P**. The value of this message is the value of **P !! R.**

```
start() ->
  spawn_link(fun win/0).

  win() ->
  Display = xStart("3.2"),
  Win     = swTopLevel:make(Display,350,145,?bg),
  Label1  = swLabel:make(Win,10,10,220,30,0,
                         ?cornsilk,"First name:"),
  Entry1  = swEntry:make(Win,140,10,120,
                         "Peg leg"),
  Label2  = swLabel:make(Win,10,60,220,30,
                         ?cornsilk, "Last name:"),
  Entry2  = swEntry:make(Win,140,60,120,
                         "Loombucket"),
  Button  = swButton:make(Win,10,100,120,30,
                          ?grey88, "Swap"),
  Button ! {onClick, fun(X) ->
                         Val1 = Entry1 !! read,
                         Val2 = Entry2 !! read,
                         Entry1 ! {set, Val2},
                         Entry2 ! {set, Val1}
                        end},
  loop().

loop() ->
  receive
      Any ->
        loop()
  end.
```

Figure 1: The code to create Figure 2

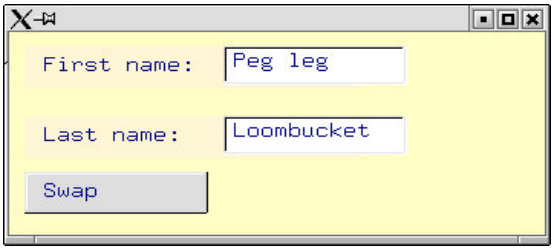

Figure 2: Window created by running the code in Figure 1

Similarly **Label ! {set, Text}** can be used to set the value of a text within a label.

Buttons are created with **swButton:make**. Once a button has been created you can tell it what to do if an event occurs within the button. Like everything else this is done by sending it a message.

```
B ! {onClick, fun(X) -> ... end}
```

Means *if the mouse button is clicked within the button then evaluates the function fun(X)* . Here **X** describes which mouse button was clicked and gives information about where in the widget the pointer was when the button was clicked.

Our little widget is designed so that the contents of the two entries will be swapped when the button is clicked. This is achieved by writing:

```
Button ! {onClick,
          fun(X) ->
             Val1 = Entry1 !! read,
             Val2 = Entry2 !! read,
             Entry1 ! {set, Val2},
             Entry2 ! {set, Val1}
          end},
```

Now evaluating a function must occur within some context, so it is pertinent to ask where this function is evaluated – the answer is within a parallel process created within the *button* process.

Note now that the callback style of programming has re-occurred, but that this time it occurs not at the top-level, as would have been the case were we to code this in the X11Lib style of programming, but within the widget processes that is responsible for controlling the widget itself. It makes sense to allow a button widget to evaluate a function when it is pressed rather than delegating this evaluation to the containing window.

If we imagine a window containing several buttons and we imagine clicking one of these buttons, then, in the normal X window style of programming the code that is evaluated when you press a button is contained in the top-loop of the event dispatching routine which is associated with the top-level window. This fact remains true even when the concurrency is hidden from the user by use of a widget library having an appropriate set of call-back routines.

Imagine further a system having two buttons called *slow* and *fast*. When the *slow* is pressed, a long and involved calculation is started, when *fast* is pressed a small computation is performed. If the user rapidly presses the *slow* button and then the *fast button,* we do not want the computation of the fast button to have to wait until the computation caused by pressing the *slow* button has completed. In a sequential call-back system this is precisely what happens – though, of course there is no need for this to happen if the actions performed by pressing the buttons are unrelated

If the widgets are themselves represented by parallel processes then these kind of problems do not occur. All computations associated with any widget can proceed in parallel. The concurrency can lead to a more subtle problem to – namely that the possibility for live- or dead-lock between processes can occur. In practice I have never seen this happen.

3. HIGHER ORDER WIDGETS

The EX11 system makes extensive use of a number of higher-order widgets which are used to co-ordinate the actions of simpler widgets. I will give one example in this paper. More examples can be found in the EX11 software distribution.

4.1 The drag box

A drag-box is a colored rectangle which can be dragged around the screen. Now that may not sound very exciting, but it can be used for a large number of different purposes.

```
D = swDragBox:make(X,Y,W,H,C)
```

makes a drag box of size **(W,H)** positioned at **(X,Y)** colored **C**.

```
D ! {onMove, fun(X,Y) -> ... end}
```

means if the drag box **D** is dragged to position **(X,Y)** then evaluate the function **fun(X, Y).**

We can use a drag box to make a draggable window, as in Figure 3.

```
DragBar   = swDragBox:make(Win,X,YY,...),
Rectangle = swRectangle:make(Win,XX, ...),
DragBar ! {onMove,
           fun(X, Y) ->
              Rectangle ! raise,
              Rectangle ! {setXY, X, Y+16}
           end}
```

Figure 3. Code to create a draggable window

The code in Figure 3 creates a drag box and a rectangle. Running this code results in Figure 4. If the drag box is moved to (X,Y) the rectangle is raised and moved.

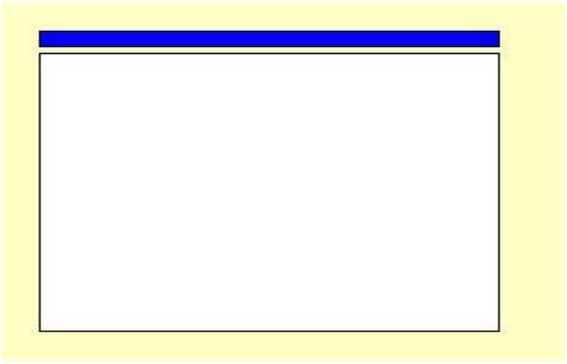

Figure 4 – a drag box and a rectangle

In figure 4 I have deliberately placed the drag box above the rectangle so that you can see the gap between the two. This is to emphasize that fact that a window is in fact constructed from two different widgets. When you move the drag box the rectangle below the drag box will follow. There is no observable delay between moving the blue box and the movement of the underlying rectangle which always follows the the drag box.

By shrinking the gap between the drag box and the underlying rectangle to zero the composite object appears to move as if it were a rigid object. Here we can clearly see that what is perceived at one level of abstraction as an indivisible window can at another level of abstraction be viewed as a composition of more primitive objects.

5. PRIMITIVE WIDGETS

If we can have higher-order things can we not also have lower-order things? The answer is yes. What happens if we extend the abstraction boundaries downwards instead of upwards?

Up to now I have considered a primitive widget (such as a button) as an indivisible object which is accessible only though a protocol. It is, however, instructive to break this abstraction boundary and see how the button itself is programmed.

Figure 5 – almost a button

The display in Figure 5 is created by running the code in figure 6.

```
win2(Pid) ->
    Win  = xCreateSimpleWindow(Pid,10,10,300,100,
                ?XC_arrow, xColor(Pid, ?wheat2)),
    Font = xEnsureFont(Pid, "9x15"),
    Pen  = xCreateGC(Pid, [{function, copy},{font, Font},
                {fill_style, solid},
                {foreground, xColor(Pid, ?DarkBlue)}]),
    Red = xCreateGC(Pid, [{function, copy}, {font, Font},
         {fill_style, solid},
         {foreground, xColor(Pid, ?red)}]),
    xCreateNamedGC(Pid, "black", [{function,copy},
                  {line_width,2},{line_style,solid},
                  {foreground, xColor(Pid, ?black)}]),
    xCreateNamedGC(Pid, "white", [{function,copy},
                  {line_width,2},{line_style,solid},
                  {foreground, xColor(Pid, ?white)}]),
    Cmds  = [ePolyFillRectangle(Win, Red,
               [mkRectangle(10,20,110,22)]),
             ePolyLine(Win, xGC(Pid, "black"), origin,
              [mkPoint(10,43),
               mkPoint(120,43), mkPoint(120,20)]),
             ePolyLine(Win, xGC(Pid, "white"), origin,
                        [mkPoint(10,43),mkPoint(10,20),
                         mkPoint(120,20)]),
             ePolyText8(Win, Pen, 12, 35,
                         "Hello World")],
    xDo(Pid, eMapWindow(Win)),
    xFlush(Pid)
```

Figure 6 – the code to create Figure 5

The details of the code are unimportant. It is its overall shape and how it interacts with the underlying EX11 libraries which are of interest.

At this level of abstraction a button is no longer a button – it is a rectangle containing colored lines and text – *nothing more*.

The above code draws the text "hello world" onto a window and it draws two crooked lines, one white and the other black. Oh, and there's also a red rectangle. By merely re-arranging the coordinates of exactly where we draw the lines and text we can turn this seeming nonsensical collection of lines into a button.

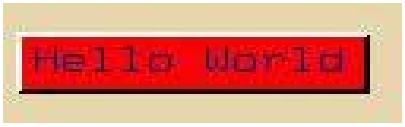

Figure 7- A button formed by rearranging the elements in Figure 5

Figure 7 is composed of exactly the same components as in Figure 5 but now it has mysteriously become a button. Of course there is no button (just as there was no window in Figure 4) just a collection of lines and text. Our eye has turned this collection of lines into a "button".

Now we have to add semantics. A button is a thing that "does something" when we click on it. Recall that the last line of Figure 6 called loop(...)

The "loop" code is written something like this:

```
loop(B, Display, Wargs, Fun) ->
    receive
      {event,_,buttonPress,X} ->
          flash(Display, Wargs),
          Fun(X),
          loop(B, Display, Wargs, Fun);
        ...

flash(Display, Wargs) ->
  S = self(),
  Win=Wargs#win.win,
  spawn(fun() ->
            xDo(Display, xClearArea(Win)),
            xFlush(Display),
            sleep(200),
            S ! {event,Win, expose, void}
        end).
```

What does this code do? - when the user clicks on the button the X11 server writes an event to the controlling socket for the window in question. This event is sent to the EX11 system where it is parsed and then sent to the handler process for the widget concerned. Finally this shows up as an **`{event, Win, buttonPress,X}`** message which is sent to the button controlling processes. This process evaluates **`Fun(X)`** to achieve whatever effect is desired by pressing the button and spawns off a parallel process **`flash`**.

Flash clears the button window and flushes the display (which will cause it to change to the background window color of the button), sleeps for 200 milliseconds and finally sends itself a "window expose" message. The window expose message will cause the window to be repainted.

6. THE PROTOCOL LAYER

The protocol layer of the EX11 system communicates directly with a socket which is controlled by the X windows server. Marshalling protocol packets is done in the Erlang model **`ex11_lib.erl`**.

`ex11_lib.erl` is as far as possible written to be isomorphic to the individual packets in the X protocol.

With a little thought and judicious use of the Erlang bit syntax this is relatively easy to achieve. As an example, the **`ImageText16`** protocol command on page 187 of the X Protocol manual [3] is shown in Figure 9. The encoding of this protocol message is achieved in the Erlang function **`eImageText16`** which is shown in Figure 8.

```
eImageText8(Drawable, GC, X, Y, Str) ->
  Len = length(Str),
  BStr= list_to_binary(Str),
  B = <<BStr/binary>>,
  req(76, Len,
      <<Drawable:32,GC:32,X:16, Y:16,
        B/binary>>).
```

Figure 8 – Code to encode the ImageText16 command corresponding to Figure 9

# bytes	*Value*	*Description*
1	77	Opcode
1	N	Number of CHAR2Bs in string
2	4+(2n+p)/4	Request length
4	DRAWABLE	Drawable
2	GCONTEXT	GC
2	INT16	X
2	INT16	Y
2n	STRING16	String16
P		Unused, p=pad(2n)

Figure 9 – Page 187 of the X protocol manual

Comparing figures 8 and 9 we see that the code is six uncomplicated lines of Erlang, the specification is nine lines long and the correspondence between the two is self-evident.

7. QUANTITIVE PROPERTIES OF THE CODE

EX11 is built from a low-level library which communicates at the X protocol level and a high level widget library.

The low level library is functionally equivalent to a subset of Xlib.

The Erlang code in the low level library has 10 modules and 4241 lines of code. The C code in Xlib has 440 code files and is 125671 lines of code. It is difficult to say what subset of X1ib that EX11 implements. I have implemented about one third of the entire X protocol and this appears to be perfectly adequate for programming a large number of different widgets.

The EX11 widget library has 17 modules and 2363 lines of code. It is unclear how many lines of code it would take to implement similar functionality in an imperative language.

8. RELATION TO OTHER WORK

The idea of using concurrency as the principle axis to structure a windowing system is not new though remarkably little work has been done in this area.

In A Concurrent Window System [8] , Pike observed that "*When implemented in a concurrent language, a window system can be concise."* giving pretty much same reasons as in this work.

Interestingly very few X graphics package interface directly at a protocol level with the X server. Two exceptions to this are SCIX [9] and the eXene system written in CML [10].

Concurrent GUIs programmed in a lazy functional language are also described in [11].

9. BACKGROUND

Before I implemented EX11 I too fell into the trap of "*not wanting to read the X manuals*" so I took the lazy way out. I downloaded just about every X graphics/GUI package that has ever been written compiled them up and tried to run them. What I saw appalled me – a large number of the packages did not work. Or at least I could not easily get them to work and so I gave up in the process. Those that did work were inflexible and difficult to use – none of them used concurrency to structure the widget set, all of them used callbacks in one form or another.

Having wasted several years trying to get other peoples' stuff to work I finally decided to attack the system from below. Fortunately Torbjörn Törnkvist had written a small Erlang program that talked to the X server though a socket. Getting started is just about the most difficult bit but Torbjörn had done this before.

10. ACKNOWLEDGEMENTS

Torbjörn Törnkvist first wrote a program to do graphics using the X protocol, later Tony Rogvall extended the protocol. Vlad Dumitrescu change this program to use Erlang binaries. I then rewrote most of the code (apart from the authentication) Shawn Pearce rewrote the name resolution code used in starting the system and Frej Drejhammar pointed out several mistakes in the design which have now been corrected.

References

[1] Armstrong, J., Virding, R., Wikström, C., and Williams. M. *Concurrent Programming in Erlang*, Prentice-Hall, 1996.

[2] Gettys, J., Karlton, P. L., McGregor, S. *The X Window System, Version 11. S*oftware - Practice and Experience, Volume 20, 1990.

[3] Nye, Adrian. *X window system – Volume 0: X Protocol Reference Manual*, O'Reilly & Associates. 1995.

[4] Nye, A. *X window system – Volume 1: Xlib Programming Manual*. O'Reilly & Associates. 1992

[5] Nye, A. *X window system – Volume 2: Xlib Reference manual.* O'Reilly & Associates. 1992

[6] Nye, A., and O'Reilly, T. *X window system – Volume 4M: X Toolkit Intrinsics Programming Manual*. O'Reilly & Associates. 1992.

[7] Cutler, E., Gilly, D., O'Reilly, T. The X window system in a nutshell. O'Reilly & Associates. 1992.

[8] Pike, R. *A Concurrent Window System* Comp. Sys., Spring 1989, Vol 2 #2, pp. 133-153

[9] Huss, H., Ihrén, J., SCIX – A scheme interface to X windows. Royal Institute of Technology , Sweden, 1990.

[10] Reppy, J. H. *Higher Order Concurrency* – Ph.D. Thesis. Cornell University 1992.

[11] Carlsson, M., Hallgren, T. *FUDGETS: a graphical user interface in a lazy functional language.* In Proceedings of the conference on Functional programming languages and computer architecture, 1993.