XINTERFACE: A dynamically configurable process level interface to X

John D. Lewis and Bruce A. MacDonald[1]

ABSTRACT

This paper describes XINTERFACE, a process level interface to the X window system. It allows programmers with a minimum knowledge of X to create an interface for new or existing applications, as well as allowing more freedom in choosing a language in which to implement the functional portion of the application. XINTERFACE is a skeleton client that is invoked interactively by a user or by an application, to form a shell process dedicated to maintaining the interface. The initial appearance and behaviour of the interface are specified by a compiled User Interface Language (UIL) file, augmented by command line arguments and default resource files. A separate process is responsible for implementing the functionality of the application. Inter–process communication allows the interface to communicate user interactions and the functional process to reconfigure the interface as needed. This approach extends the advantages of UIL as well as exploiting the superior language support for process communications, over direct window system interaction.

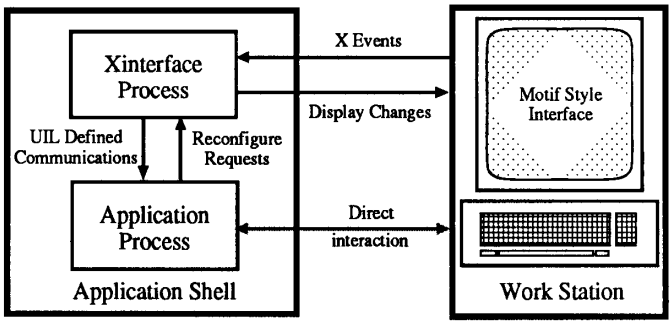

Figure 1: XINTERFACE in operation

1 INTRODUCTION

In this section we discuss the goals of the project, outline a solution provided by XINTERFACE, and present two examples of completed applications. The next section describes XINTERFACE in more detail, how it can be used, and where it fits among the existing X products. XINTERFACE is then exemplified by a number of useful applications. Finally the advantages and disadvantages are discussed.

1.1 Goals

X is a large complex system, with many levels of abstraction, facilities, options, and styles. It requires a significant initial effort for programmers. The difficulty of producing even simple interfaces in X motivated this project. The goals were two–fold: firstly, we wished to speed up the process of producing graphical interfaces; and secondly, we wanted to expand the number of tools and languages from which X is available.

[1] Computer Science Department, The University of Calgary. This research is financially supported by the Natural Sciences and Engineering Research Council of Canada.

XINTERFACE achieved these goals. The first was attained by using the MOTIF Resource Manager's (MRM) support for UIL [5] thereby reducing the knowledge necessary to use X, and so, speeding up interface development. The latter was accomplished by making the interface wholly process oriented and exploiting the more common support for processes than for window systems.

1.2 Overview of XINTERFACE

XINTERFACE is a skeleton client that is dedicated to maintaining the "look and feel" portion of an application as defined by a UIL specification. The functional aspect of an application must be provided by a separate process. These two processes exchange information via Unix pipes. The UIL specification denotes which user interactions are to be communicated by XINTERFACE to the functional process over the pipe. Conversely the functional process can communicate reconfiguration requests to XINTERFACE which will then install the requested changes. Figure 1 depicts XINTERFACE in operation.

1.3 Two Examples

Two examples follow which show how the separation of form and function can be used to advantage.

A simple example is given by a Chez SCHEME [1] based calculator which is shown to the right and discussed later in detail. Here XINTERFACE provides access to X from a language which does not supply direct support.

The second example shows that XINTERFACE provides a means for attaching an interface to an existing application. With the excellent GNU EMACS [2] support for sub-processes an XINTERFACE process can provide a graphical interface without modifying the editor. Figure 3 depicts the GNU EMACS interface in use with several of its popup widgets active. Notice that the current buffer contains the GNU EMACS lisp code which implements two way communication with XINTERFACE. Other interfaces can be provided simply by specifying an alternate UID (compiled UIL) file.

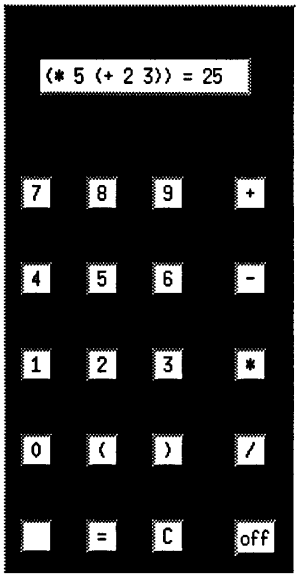

Figure 2:calculator

2 XINTERFACE

The relationship between XINTERFACE and other components of the X window system is described. Then a detailed description of XINTERFACE is given followed by an illustrative example.

2.1 Xlib and the Toolkits

X provides programming abstraction levels ranging from the painful Xlib level, through Xt, and MOTIF to the more acceptable combination of UIL and MRM. UIL is a specification language for describing the initial state of a MOTIF application's user interface. Using UIL avoids most of the intricacies involved in creating an interface. In a UIL file a programmer specifies the hierarchy of objects that comprise the interface and MRM handles the creation and initialization of the MOTIF widgets. This provides a quick and easy means of coding an interface as well as providing an error detection mechanism when the UIL code is compiled. However, after the interface has been created the application is responsible for maintaining it with additional X calls. Here the programmer is not "protected" from the lower levels, as a considerable knowledge of X is still required; often one must resort to lower level calls to accomplish tasks such as graphics.

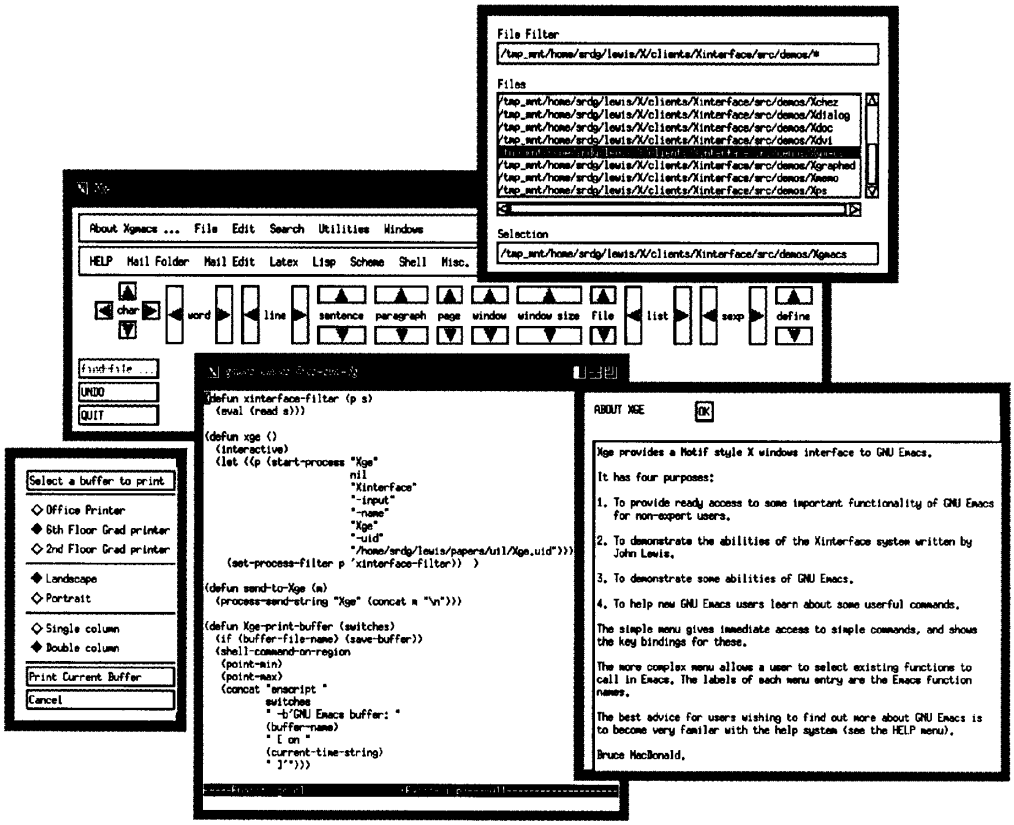

Figure 3: XGMACS in operation. The main XINTERFACE window is shown with several popup windows activated. User configurability allows unwanted functions to be removed from the menu if the user finds it cluttered.

2.2 XINTERFACE, UIL, and MRM

XINTERFACE provides this missing protection. It is built on top of MRM's support for UIL and gives a high level tool for X that completely hides lower levels in the abstraction hierarchy. The interface is described completely in terms of UIL and a set of predefined callback functions. XINTERFACE assumes the responsibility of managing the interface; it uses MRM to read in the UIL specification and create the initial interface, and then maintains it by translating requests from the application process into X calls. But XINTERFACE does more than just support the UIL specification; it extends UIL in the following ways.

Separation of Form and Function is complete in XINTERFACE. Since it runs as an independent process the interface may be completely modified — maintaining only the communication to its functional counterpart — without affecting the functionality of the overall application. This approach has already proven useful: several GNU EMACS interface specifications have been developed to provide graphical interface capabilities for beginner, intermediate, and expert users.

Predefined Callback Functions are the key to this complete separation of form and function. They allow an interface to be completely defined — not just appearance but also behaviour — in the UIL specification. The predefined functions enable:

- creation and destruction of objects
- management of existing objects
- access to any object's resources, variables, window, and callbacks
- "virtual" addressing of objects
- inter–object communications
- communication to the calling process
- system command invocation.

These callback functions provide a great deal of power. One system demonstrates this by defining a template for each MOTIF widget and enabling an application (or a user if the interface is invoked directly from a shell) to dynamically create and modify an interface. As well as an excellent prototyping tool, this demonstration could be the basis of a simple, interactive, graphical interface builder.

An XINTERFACE Object is created simply by registering a UIL object in the XINTERFACE object table. In addition to the resource set provided by the UIL object from which it was built the XINTERFACE object allows: (1) name aliasing; (2) an extensible resources set; and (3) Graphics support. These items will be discussed below. Once created, an object is accessible to other objects who may read the value of its resources, and variables; and both other objects and the functional process may send it messages. Message communications can completely reconfigure an object — or even destroy it — but the sender must know the registered name of the object that it wishes to send to.

Name Aliasing adds a level of indirection to this system. Exchanging the registered names of objects allows a type of "virtual" addressing to be applied to XINTERFACE objects. This allows the single name to be used for different objects at different times. For example, a global *previous-window* object is maintained in the interactive self–documentation for XINTERFACE: as the user moves through the window hierarchy each object registers the node it should return to; when the button labelled *previous* is pressed it simply calls upon the object registered as "previous".

Object Templates are supported by this naming methodology as well. A single UIL definition can be reused to create multiple instances each of which is registered under a different name. Since UIL objects have a predefined set of arguments which parameterize their behavior — such as the **labelString** argument of a button — each instance of a template will appear and behave according to the resources provided for it under its registered name. This has been implemented such that the resources defaults file can refer to instantiated objects by their registered names. The calculator example shown in figure 2 uses this technique to supply most of the buttons.

Extensible Resources are provided by XINTERFACE to exploit the potential of this approach. XINTERFACE provides *variables* (as well as the standard UIL arguments) which can be used to define the behaviour of the interface. An object's behavior is defined by the actions it takes when its callback methods are activated. If an object refers to arguments or variables in its callback functions then it can behave differently from other objects created from the same UIL template. Additionally, by changing the values during execution, the behavior of an XINTERFACE object can be redefined dynamically.

Graphics Support is provided by each XINTERFACE object. A POSTSCRIPT interpreter[2] converts POSTSCRIPT commands into low–level X calls. If POSTSCRIPT commands are sent to an XINTERFACE

[2] A slightly modified version of **ralpage** originally written by Crispin Goswell and later adapted to X by John G. Myers has been used

object then it invokes a POSTSCRIPT interpreter — a different one for each object — which runs as a separate process and draws directly on the objects' window. This allows complex graphics commands to be performed without interfering with the handling of other interface events. Since a different interpreter is invoked for each object a separate drawing environment exists for each window.

Also, each object is capable of communicating events back to the functional process. So, as well as allowing POSTSCRIPT commands to be sent to a drawing object from the functional process, user interactions can be communicated back to enable interactive editing. This approach was used to provide a graph editing program which automatically lays out[3] the nodes and edges of a specified graph, displays the result, and awaits interactions from a user to edit the display.

2.3 XINTERFACE Communication

Communication is event driven. XINTERFACE events are generated by messages from the functional portion of the application, user actions on interface objects, or messages from other objects. Messages passed between the interface and its functional counterpart are simply strings printed over the standard input and output channels of XINTERFACE. Arbitrary output message strings are generated by callback methods — which are invoked as a result of user interaction — to be interpreted by the application. (e.g. the SCHEME and GNU EMACS messages are lisp code). The application reads the incoming messages, performs the necessary functions, and then communicates any required changes back to the interface.

Both external input and inter–object communication is implemented by a single message handler which expects messages to take the form:

message ≡ {POSTSCRIPT | SETARG | SETVAR | CALLBACK } object *a*

where *a* may be

- postscript code for display
- an argument–value pair to modify
- a variable–value pair to modify (or automatically create)
- a callback method to invoke.

Messages can be used to completely reconfigure the interface including: creating and destroying objects, modifying arguments and variables, and drawing POSTSCRIPT on an object's main window.

2.3.1 An example from SCHEME To illustrate how the communication between XINTERFACE and the application process is conducted we give details of the calculator example. Figure 4 shows the UIL specification, the resource defaults, and the SCHEME code.

The UIL specification consists of three main parts: a text display, a button object which is replicated for each numeral and function, and three special purpose buttons ("off", "clear" and "="). The button instantiations are performed by a callback when the calculator is created, demonstrating the convenience of object templates. Each of the three special buttons performs particular functions when activated, as shown by their "activate callback" definitions. The "off" button cleanly exits XINTERFACE. The "clear" button sets the "value" variable of the text object — which is used to remember the calculator display contents — to null (this demonstrates inter–object communication). The "=" button sends the current value to SCHEME (demonstrating output from XINTERFACE). The replicated buttons each append their label (specified in the defaults file) to the current display string, and activate the text callback (more inter–object communication). This in turn sets the *argument* value to the new string so that it will be displayed. The text object registers itself on creation, and

[3] Steve Easterbrook, Brian Gaines, and Angus Davis implemented the graphing algorithm, based on [6]

UIL code for calculator

```
module xinterface
      names = case_sensitive
include file
        'Xinterface.uil';
object
    xinterface_main : XmBulletinBoard {
        controls {
            XmText calc_text; XmPushButton calc_off_button;
            XmPushButton calc_eq_button; XmPushButton calc_cl_button;};
        callbacks {
            MrmNcreateCallback = procedures {
                xinterface_create_widget('"calc_button calc_0_button"');
                xinterface_create_widget('"calc_button calc_1_button"');
                        .
                        .
            };};};};
object
    calc_off_button : XmPushButton {
        callbacks {
            XmNactivateCallback = procedure xinterface_exit('"0"');};};};
object
    calc_eq_button : XmPushButton {
        callbacks {
            XmNactivateCallback = procedure \
                xinterface_fprintf('stdout, "%s\n", Registry(calc_text).value');};};};
object
    calc_cl_button : XmPushButton {
        callbacks {
            XmNactivateCallback = procedures {
                xinterface_iprintf('"SETVAR calc_text value "');
                xinterface_iprintf('"CALLBACK calc_text activateCallback"');};};};};
object
    calc_button : XmPushButton {
        callbacks {
            XmNactivateCallback = procedures {
                xinterface_iprintf('"SETVAR calc_text value %s%s", \
                                Registry(calc_text).value, Resources().labelString');
                xinterface_iprintf('"CALLBACK calc_text activateCallback"');};};};};
object
    calc_text : XmText {
        callbacks {
            MrmNcreateCallback = procedures {
                xinterface_register_widget('"calc_text"');
                xinterface_iprintf('"SETVAR calc_text value "');
            };
            XmNactivateCallback = procedure \
                xinterface_iprintf('"SETARG calc_text value %s", \
                                    Registry(calc_text).value');};};};
end module;
```

Resources for calculator

```
Xcalc*.xinterface_main.background:  black
Xcalc*.XmPushButton.background:     white
Xcalc*.calc_text.editable:          false
Xcalc*.calc_eq_button.labelString:  =
Xcalc*.calc_0_button.labelString:   0
Xcalc*.calc_1_button.labelString:   1
                        ... (rest omitted)
```

Chez Scheme code for calculator

```
(define comm-proc (process "Xinterface -uid Xcalc.uid -name Xcalc -input"))
(define *X-IN*  (car comm-proc))
(define *X-OUT* (cadr comm-proc))

(define *error-handler*
  (lambda x
    (display (format "SETARG calc_text value ~a~%" (cadr x)) *X-OUT*)
    (flush-output-port *X-OUT*)
    (read-loop)))

(define read-loop
  (lambda ()
    (let ([s (read *X-IN*)])
      (if (eof-object? s)
          (exit)
          (begin
            (display (format "SETARG calc_text value ~s = ~s~%" s (eval s)) *X-OUT*)
            (display (format "SETVAR calc_text value ~%") *X-OUT*)
            (flush-output-port *X-OUT*)))
      (read-loop))))
(read-loop)
```

Figure 4: Calculator code. UIL definition, resource defaults, and SCHEME code

when activated copies the variable named **value** into the standard text object argument named **value**, for display. The defaults file gives the label, size and position of each button, as well as setting the background colors. Note that in this simple example, the button labels are equivalent to the SCHEME functions or numbers they invoke. The argument "userData" can be used for specifying the function, when the label is not the same.

The SCHEME code for the calculator is shown at the bottom of figure 4. The first three lines invoke XINTERFACE as a subprocess for SCHEME and establish bi-directional communication through ports. The error handler redirects SCHEME errors to be displayed in the calculator text panel. The read loop checks for an end of file then sends the result of evaluating the expression.

2.4 Existing Systems Using Xinterface

By the time XINTERFACE's development was complete, nine applications already existed and some were used regularly. Customizable interfaces have been developed for GNU EMACS, Postscript display, a **dvi** previewer, graph layout, a calculator, a memo editor, a dialog box tool, a mail alarm, and interactive XINTERFACE documentation. These examples are discussed below, and show that XINTERFACE can greatly reduce the complexity of providing such interfaces. With the exception of the GNU EMACS interfaces and the online XINTERFACE documentation, each UIL file is less than one page of straightforward UIL specification.[4]

GNU EMACS: The GNU EMACS examples show that user interfaces can be added to existing applications without modification, when a process interface is available. GNU EMACS supports process filters, allowing an XINTERFACE process to be started during EMACS initialization, by command line argument, or by the user during execution. By specifying a simple read/eval loop as the process filter in EMACS (see Figure 3), and having the XINTERFACE callbacks send lisp code, XINTERFACE can invoke EMACS functions. We have constructed such a sample interface with various push button controls, pulldown menus, and popup dialog boxes, to make GNU EMACS more accessible to beginners. It is reasonably easy for more experienced EMACS users to tailor the few pages of UIL code to their own use.

POSTSCRIPT Interpreter: XINTERFACE utilizes a slightly modified version of `ralpage` to provide postscript graphics capabilities for all interface windows. An XINTERFACE POSTSCRIPT previewer was created using a simple Drawing Area object. The application simply reads postscript commands and forwards them to the interface with the object name and postscript identifier prepended. An interactive debugger for an implementation of Lozano–Perez's [3] Cspace robot motion planner also uses the Drawing Area UIL specification.

Dvi Previewer: The POSTSCRIPT interpreter also provided a simple mechanism to preview **dvi** files which contain POSTSCRIPT graphics. The application simply converts each page of the **dvi** file to POSTSCRIPT and forwards it to the POSTSCRIPT interpreter.

Interactive Graph Editor: The Drawing Area UIL specification also served as a display canvas for an interactive graph editor. After the graph is displayed the application waits for communication (from XINTERFACE) resulting from user actions on the displayed window. The type and location of button presses, mouse movements with a button down, and button releases are returned to the application, and are used to move the graph nodes and redisplay. So a graph is initially laid out automatically by the grapher, then interactively tuned by the user.

[4] XINTERFACE and all demonstration systems are available by anonymous ftp at: cs-sun-fsa.cpsc.UCalgary.CA, 136.159.2.1.

Memo Editor and Dialog Box: The standard process interface to XINTERFACE allows the production of interfaces which are "standalone" processes; effectively using a simple shell script as the application. The memo editor and dialog box programs were written in this way. Xdialog provides a general mechanism for requesting command line arguments from a user, thereby enabling clients to be invoked in an application which lacks the mechanism itself, (e.g. invoking a shell script from a menu). Xdialog is defined with 40 simple lines of UIL code and invoked by a one line script. The memo editor is a simple interface to the UNIX **at** and **mail** commands. It provides a mechanism to edit notes and send them to yourself at a given time.

Interactive Mail Monitor: A mail alarm comprises a single button in a window. The button is at first insensitive and upon it is displayed a bitmap to indicate that no mail is present. When new mail is received the functional process changes the bitmap and sensitizes the button. When the button is activated the new mail will be processed in a user configurable way.

Interactive Documentation Reader for XINTERFACE: The online version of the XINTERFACE Reference Manual uses XINTERFACE. The manual is displayed in a text window and a hierarchical menu allows the reader to index areas of interest and be automatically repositioned in the manual.

3 DISCUSSION

This section will discuss the advantages and disadvantages of the approach used in XINTERFACE.

3.1 Disadvantages

Interfaces using this approach are practically limited to what can reasonably be specified with the extended UIL language. This will not include some complex, intricate interfaces which interact at more than one level of the application. Top-level interfaces are more suitable, as is any problem where there is some natural separation between the application and the interface (thus allowing process communication to be effective). Note however, that XINTERFACE has been used successfully for many diverse applications.

A more critical problem with XINTERFACE is that communication with the functional portion of the process must incur the overhead of UNIX pipe communication. This may be unacceptable for simple callbacks, but will be inconsequential for callbacks that take longer to execute. Callbacks which require a long time to complete should fork off a process to perform the computation so that the parent can return to the event loop. For applications which do not wish to incur the overhead of communication over pipes, XINTERFACE can still be useful. Applications which will eventually use MRM directly can initially use XINTERFACE to develop the UIL description.

3.2 Advantages

There are two primary advantages to the XINTERFACE approach: (1) XINTERFACE provides a quick and easy means of producing Motif style interfaces; and (2) XINTERFACE can be used when the functional process does not support X directly. The first is demonstrated by each of the systems which have been implemented using XINTERFACE. In particular the calculator example gives a basis for comparison: the XINTERFACE version is implemented in less than 100 lines of UIL and less than 20 lines of scheme; the Xt version — which is slightly more complex — is about 300 lines of C code for the interface and about 800 more lines of C for the functional portion. The X11R3 version which was written at the Xlib level was significantly larger again.

The adoption of a process level interface provides a simple mechanism to support user interfaces for a wide variety of applications. While relatively few languages provide direct X support a considerably

larger number provide support for processes. This allows the functional portion of an application to be written in the programmer's language of choice.

A variety of important but lesser advantages are listed below:

- Interfaces can be developed independently from, and without adding complexity to, applications.
- Considerably less expertise is required of the XINTERFACE programmer than is of a MRM and UIL programmer.
- XINTERFACE can be easily incorporated into an existing product if that product supports processes.
- A completely different interface module can be attached to the functional portion of an application — or vice versa — as long as the communication between them is compatible.
- Interface specifications are reusable leading to standardization of the "look and feel" among a group of applications.

4 CONCLUSION

XINTERFACE is a powerful prototyping tool and a reasonable way to provide interfaces for languages and tools that do not possess window systems. It makes the X window system more accessible to both users and software systems.

REFERENCES

[1] Dybvig, Kent R. (1987) *The Scheme Programming Language.* Prentice–Hall.

[2] Stallman, Richard (1986) *GNU Emacs manual.* Fifth Edition.

[3] Lozano-Perez, T. (1986) Motion planning for simple robot manipulators. Third International Symposium on Robotics Research, edited by O. Faugeras and G. Giralt, MIT Press, 133–140.

[4] *OSF/Motif programmer's reference guide.* Open Software Foundation, Cambridge, MA.

[5] *UIL programmers's guide.*

[6] Watanabe, H. (1989) Heuristic Graph Displayer for G-BASE. Ricoh Software Research Center, Tokyo. International Journal of Man–Machine Studies. Vol. 30-3, March, 287–302.