XXL: A Dual Approach for Building User Interfaces

Eric Lecolinet

Ecole Nationale Supérieure des Télécommunications (ENST)
46 rue Barrault
75013 Paris - France
Eric.Lecolinet@enst.fr

ABSTRACT

This paper presents XXL, a new interactive development system for building user interfaces which is based on the concept of textual and visual equivalence. XXL includes an interactive builder and a "small" C compatible special-purpose language that is both interpretable and compilable. The visual builder is able to establish the reverse correspondence between the dynamic objects that it manipulates and their textual descriptions in the original source code. Interactive modifications performed by using the builder result in incremental modifications of the original text. Lastly, XXL not only allows users to specify the widget part of the interface but can also be used to manage various behaviors and to create distributed interfaces.

KEYWORDS: User interface software, interface builders, scripting languages, textual and visual equivalence, iterative development, distributed interfaces.

INTRODUCTION

It is a well-known fact that user interfaces constitute an essential part of software applications but still remain difficult to conceive and to implement [4,5]. Besides the difficulties inherent to the task of designing human-computer interfaces, we still believe that a non negligible part of this complexity also results from the design of available UI software tools and that there is still room for improvement in this field. So, we propose a new system called XXL that includes an interactive builder and a C compatible language and that is based on the concept of textual and visual equivalence.

RELATED WORK

Interface Builders

As noted in [9], interface builders are intuitive and easy to use but not expressive enough in the general case. These systems are quite convenient to create standard objects such as menus or dialog boxes but provide little support for creating application specific objects and specifying their behavior (for instance a music editor, an image visualizer, graph representations...). Besides, they are generally unable to process variable amounts of data or data that changes at run time (for instance a file manager, visualization of complex data coming from a remote database server...).

Such limitations can in fact be considered as the intrinsic consequence of "direct manipulation". Moving and manipulating widgets directly by using the mouse pointer is especially intuitive to novice programmers because of the "concreteness" of such a process. But this concreteness implies a corollary drawback: the inability to specify "immaterial" behaviors or parametrization in a simple and coherent way (like for instance: the dynamic creation and positioning of new objects, the creation and parametrization of variable sets of objects, or the processing of complex events and the related behaviors). Thus, these systems require writing source code to specify complex or dynamic behaviors (and in fact, certain systems even require writing callback functions to program quite basic interactions).

Scripting and Special Purpose Languages

This category includes very different kinds of systems but their common point is to facilitate and rationalize the textual specification of user interfaces. A primitive example is UIL, a special purpose language that provides a simpler interface to the Motif toolkit than the usual C interface. However, UIL is only able to represent the widget part of the user interface (i.e. the graphical objects and their properties). In contrast, Tcl/Tk [7] provides a scripting language that includes constructs that can also deal with the state and behavior of the user interface at run time. Furthermore, Tcl/Tk code is rather compact and (relatively) easy to understand. This last point explains the success of this system and shows that visual systems do not have the exclusivity of simplicity: textual languages can also be pretty convenient to use if well designed and well suited to the domain.

Amulet [6] is based on an object oriented language which is written on the top of C++. Once again, this specialized language provides simpler, better adapted and more powerful specifications to program the Amulet toolkit than a classical C or C++ interface would do. Furthermore, a major advantage of this system lies in the fact that it is C++ compatible and does not impose developers to deal with "exotic" languages that are difficult to integrate with the functional core of the application.

Model-Based Systems

Model-Based Systems [8,9,10] can generate most of the presentation automatically from a high-level description of the interface. These are rather text-based approaches (the interface must generally be specified textually by means of a special-purpose language). Some systems also include powerful interactive tools [9], but most of them do not allow interactive specification (or just allow for "cosmetic" little modifications in a separate stage). These systems rely on rather complex tools and strategies that are beyond the goal of the proposed approach which is rather oriented towards interactive development. However, XXL provides abstraction capabilities that are connected to certain concepts of this type of approaches.

PROPOSED APPROACH

It seems that most existing systems tend to be based either on a visual *or* on a textual approach but provide poor integration capabilities between these two forms of representation. This is especially true for interface builders. They allow intuitive and simple visual specifications, but at the same time, they often require writing source code without providing much help to simplify this code. Moreover, this separation is increased by the fact that these systems are usually based on *univocal source code production*: the produced code consists in a very large number of quite illegible lines of code and is not supposed to be modified - nor even to be read - by the developer. So, any further textual modification performed by the programmer ***will definitively be lost*** if the interface builder is used again. In other words, these systems are not able to take into account *textual modifications performed by an external source* (i.e. by another tool or a programmer) to the code that they initially produced. This is quite an important limitation and we believe it is the cause of many difficulties in programming complex interfaces because:

- On the one hand these tools impose textual programming to specify complex (and sometimes even simple) behaviors, but on the other hand they strongly limit how textual specifications can be done or modified,
- This results in a strong separation between the part of the interface that can be specified visually (the widgets and their appearance) and the part that must be specified textually (widget interaction and dynamic (time-varying) behaviors).

The code that defines the interface components is thus located in different files that can not be modified or reorganized easily. For instance, such simple things as creating a variable number of widgets may become complicated: should the developer define this widget by means of the interface builder, then try to find the corresponding code and modify it (in which case he will not be able to reedit and modify this widget visually by using the builder).... or had he better write this code directly by using basic toolkit objects and functions?

Textual and Visual Equivalence

The XXL system proposes a *dual approach* that attempts to integrate visual and textual programming into the same framework. The key idea is to benefit from the advantages of both textual and visual specifications and make them cooperate in a powerful way. To achieve this goal XXL is based on the following principles:

1. All interface components have a *dual representation*: a declarative *textual* specification and an iconic *visual* representation. Interface components are not limited to *widgets* (the interactive objects of a graphical toolkit) but also include objects for specifying widget appearance or behavior and control statements.
2. All UI components can be *mixed together*, either by textual or visual programming, independently of the way they were initially produced (i.e. textually or visually).
3. Programmers can always *choose the most appropriate means* to represent what they need. For instance, graphical objects can either be specified and configurated visually or textually, depending on programmers' preferences and their level of expertise.
4. The XXL visual builder is able to reedit and modify interactively any legal XXL description. The builder is thus able to establish the ***reverse correspondence*** between the dynamical objects that it manipulates and their textual descriptions in the original **C** or **C++** source code (with no restriction on the location of these descriptions in that source code).

The builder can thus deal with *pre-existent C code* and let the user manipulate interactively the corresponding XXL descriptions. This also means that C code resulting from previous specification with the builder can be modified and reorganized textually (by a programmer or another program) and then be reedited and modified again by using the builder. Furthermore, the original source code is modified in an incremental way by the builder and these modifications can even be made ***while the application is running***. A very important consequence of this approach is that it guarantees a *truly iterative development scheme* whereas classical interface builders often break this development loop.

Three View Edition

The XXL Builder provides three views of the interfaces that are being developed: the *text view*, the *graph view* and the *widget view*. The text view shows the corresponding descriptions in the source code. The graph view is an iconic representation that is equivalent to the text view. These two views constitute the dual (textual and visual) "abstract" specification of the UI while the widget view can be seen as the "result" of this specification. These views are linked together and are incrementally updated whenever the UI is modified interactively by using the builder. Interface components can be selected in any view (when applicable) and are then automatically highlighted in the other views.

This three-view model makes it possible not only to show and control the resulting code but also to represent the "hidden part" of the UI. As said before, XXL components are not limited to graphical widgets but can potentially specify any kind of functionality or data specification (the set of XXL components being extensible). This is a major difference with classical point-and-click direct manipulation interface builders: XXL does not only provide a WYSIWYG representation but is also able to represent non-widget components. The graph view is not a mere widget tree (a feature that is available in some commercial systems) but can also represent objects that do not have a visible representation in the widget view. So, this system can be considered as an attempt to let the programmer see "what is behind the curtain" and provide the kind of "indirect manipulation" defined in [2] as the ability to "directly manipulate an abstraction that controls the behavior or appearance of the actual objects".

The XXL Specification Language

The XXL system is based on an underlying specification language which is designed to be compact and reasonably easy to understand. The main interest of this language is that it is not "yet another programming language" (like for instance Tcl or UIL): in spite of its very specific form, it is actually a subset of the **ANSI C** language. This means that XXL descriptions can be freely included into C or C++ functions or other constructs and can be **compiled** as any other C statement. XXL interfaces can thus make use of the whole power of a standard programming language. This point is especially important when designing complex interfaces that deal with variable amounts of data or that evolve dynamically at run time. To paraphrase a quotation from [9], this means that we can both benefit from the intuitiveness - but (relatively) low expressivity - of interface builders and the low intuitiveness - but high level of expressivity - of standard programming languages.

Other systems (for instance the Amulet [6] system) also provide a special purpose language written on the top of a standard language. But, as said before, XXL is not only a text-based solution but also provides true equivalence between textual and visual descriptions. Conversely, FormsVBT [1], which is one of the few systems that are also based on a mixed textual/visual paradigm, does not provide a C compilable language. Moreover, these two systems rely on specific implementations of the widgets and do not provide the same kind of interactive specification capabilities.

Run-Time Interpretation and Distributed Interfaces

In addition to the previous characteristics, XXL descriptions can also be *interpreted* at run-time. The system ensures that the resulting graphical interfaces will be the same in both cases (compilation or interpretation) and will be manageable in the same way by the Visual Builder. This feature leads to the following possibilities:

Script Files. XXL descriptions can be located in *Script Files* that can be run as usual Unix scripts. These files are executed by means of the **XXL Shell**, a special-purpose interpreter. Moreover, XXL provides simple mechanisms for interacting with the standard Unix shell. This feature (which is somewhat similar to the Tcl/Tk shell) can be used not only to add simple UI to Unix commands (or other non-graphical programs) but also to test new interfaces at the beginning of the prototyping phase. Here, the major difference with Tcl/Tk and other similar systems is that these descriptions can then be ***directly included*** into C or C++ code *without any modification or translation* and can then still be modified visually by using the Builder.

Such a scheme provides the advantages of both interpretation and compilation: interpretation speeds up the development stage while compilation offers better performances at run-time. Moreover, compilation avoids having to provide the final user with source code (this can be useful for commercial applications) and simplifies the installation of executable programs (the binary files are self-sufficient and do not require the installation of specific description files or proprietary shell programs).

Run-Time Interpretation in C Code. Scripts files can also be dynamically loaded from C programs, in which case the included descriptions are interpreted at run-time. These descriptions can access the call-back functions of the C program and share static variables with this program.

Distributed Interfaces. XXL Interfaces can be exchanged dynamically between separate (possibly remote) programs. The corresponding descriptions are sent through the network by using sockets and are interpreted at run-time by the receiving program. This mechanism can also be used for modifying widgets, updating variables or calling functions that reside in remote programs.

THE XXL SYSTEM

XXL consists of an object oriented software layer which relies on a pre-existent toolkit and windowing system. So, XXL brings a better interface and new functionalities to standard commercial widgets but *does not implement truly new widgets* (all graphical objects provided by XXL are mere encapsulations or combinations of the pre-existent widgets of the underlying toolkit). Thus, programmers will find the usual properties of well-known commercial widgets again. The present version of the system has been written on the top of the **Motif** toolkit and the **X Window** system. However, its layered conception could make it possible to implement it on the top of other toolkits.

XXL includes a set of predefined objects that constitute the basis of the interface descriptions and a set of functions which aim at facilitating the writing of callback (and other) functions. There are four different types of objects:

1. *Widget Objects* that "encapsulate" the graphical widgets of the underlying toolkit (these objects are instantiated into Motif graphical widgets),
2. Objects for specifying *control statements* such as repetitions or conditional evaluation,
3. *Structuring objects* whose aim is to decompose the interfaces into smaller components (called *subinterfaces*),
4. *Property objects* that specify the appearance and behavior of the actual widgets.

New subclasses can be derived from these predefined classes in order to integrate into the XXL systems widgets other than those primarily provided by the Motif toolkit. XXL also includes the concept of *pseudo-classes*. Pseudo-classes can encapsulate a subinterface of XXL objects and let the developer manage it as if it was a single object. This encapsulation mechanism helps to structure the interface into homogeneous and reusable subparts. It allows one to construct composite objects in a simple way without having to deal with the inner functionalities of the underlying toolkit (for instance, creating a new X Window widget is quite a complex task that requires an extended knowledge of the implementation details of this system).

To each XXL object corresponds a textual and a visual representation that depends on the *metaclass* of this object. As user interfaces generally consist of trees (or sometimes of graphs) of various objects, we have decided to adopt a list formalism to represent the textual descriptions of XXL objects. So, each XXL object is represented by means of a list whose first member indicates the class of this object. The following members of the list depend on the metaclass of the considered object. For instance, in the case of widget objects, the class name must be followed by the name of the newly created instance and by the possible children of this object (if any). The parenthood between XXL objects is thus implicitly defined. Figures 1 and 2 show an example of an XXL description and the resulting graphical interface. It must be noticed once again that, despite its unusual form, this textual description is strictly legal (i.e. compilable) **ANSI C** code (there is no specific pre-processing stage before the C or C++ compilation and these descriptions do not require run-time interpretation). Fig. 1 illustrates several features of XXL descriptions that are described in the following paragraphs:

❑ The embedded list that describes the whole interface can be decomposed into several sub-blocks that are referenced by intermediate C variables (for instance, variable open_dialog points to the description of a dialog box which is then referenced in the description of the file_menu). This decomposition has no functional effect and is only provided as a way to make descriptions easier to read. The whole description is encapsulated by a special object called **Interface** that is purely *virtual* (it does not necessarily correspond to a graphical widget) and that defines a new *subinterface*. It is possible to define as many subinterfaces as needed which can be embedded. So, all interfaces (or subinterfaces) can themselves be made of several (and possibly reusable) subinterfaces.

❑ The XXL specification of an interface does not produce by itself the creation of any graphical widget. These are actually created by *instantiating this formal description* by means of the **XlBuild** function. This function produces an instance tree of Motif graphical widgets from the specification tree (or graph) of XXL objects. XXL descriptions can thus be considered as *models* that are instantiated into physical realizations. Moreover, these descriptions can be parameterized and can be multi-instantiated into actual widgets (XXL taking care of all the necessary duplication of data and other constructs to avoid unwanted interactions between these independent instances).

❑ This example also shows how widget resources (i.e the presentation properties, constraints and callback functions which are attached to the widgets) can be specified. The **Args** object specifies a list of attribute names and values that are given to the corresponding widget at creation time. The **Set** object shares the same syntax but is used to modify these attributes dynamically (that is to say once the widget has been created). Finally the **Callback** object specifies the C callback functions that will be called when events occur on widgets. These resource descriptions can possibly be shared by several graphical objects. Furthermore, the "value" fields of these descriptions can either be textual strings (that are automatically converted into an adequate internal representation), numerical constants or refer to C variables.

```
void foo(Widget parent_widget)
{
/* This Dialog Box calls the OpenProc function when a file is selected */
XlObj open_dialog =
  (FileSelectionDialog, "open_dialog",
    (Args,XmNdialogTitle,"Open Image",o),
    (Callback,XmNokCallback,OpenProc,NULL,o),
  o);

/* File Menu: each button opens a previously defined dialog box*/
XlObj file_menu =
  (PulldownMenu, "file_menu",
    (Button, "New",  new_dialog,o),
    (Button, "Open", open_dialog,o),
    (Button, "Save", save_dialog,o),
    (Separator,"",o),
    (Button, "Exit", exit_dialog,o),
  o);
   ........etc .....

XlObj menubar =
  (MenuBar, "menubar",
    /* each button will open the corresponding pulldown menu */
    (Button, "File", file_menu,o),
    (Button, "View", view_menu,o),
    (Button, "Reco", reco_menu,o),
    ....etc .....
    (Button, "Help", help_menu,o),
    (Set, XmNmenuHelpWidget, ".Help", o),
  o);

XlObj browser_obj =
  (Interface, "browser",
    (VBox, "browser",
      menubar,
      (Label, "message", o),
      (ScrolledWindow, "scrollwin",
        iconbox =
          (HBox, "iconbox"
            /* this box will have a white background */
            (Args,XmNbackground, "white", o),
  o),o),o),o);

/* This function will create the actual Motif widgets */
XlBuild(parent_widget, browser_obj);
}
```

Fig. 1: C code specification of the UI shown in Fig. 2. This UI contains a menu bar (that opens pulldown menus), a message zone and an (empty) icon box located in a scrollable window. The description is instantiated into widgets by calling the **XlBuild** function.

VISUAL BUILDER

The Visual Builder can manage the three views of all the interfaces or subinterfaces that are contained in an application (Fig. 2). The icons of the graph representation are all active and can be handled directly in order to modify the corresponding interfaces: icons can be moved, added or removed, their parenthood (which is materialized by arrows) can be changed interactively, etc. In any case, such modifications have an immediate effect on the widget view (the "resulting interface") and on the corresponding textual description. All forms of representations are thus modified at the same time in an incremental way. XXL objects can be selected by clicking in an appropriate way on any of their visible representations (widgets have three representations while other objects only have text and graph representations). Such selection immediately highlights the other corresponding representation(s). Objects can be edited and modified by means of specialized editors that are accessible via their corresponding icon (this icon and its associated behavior being dependent on the metaclass of the corresponding object). For instance the builder provides a complete Resource Editor (Fig. 3) that can show and modify interactively almost all the properties of Motif widgets.

Unlike most interface builders, most interactions do not take place on the "physical envelope" of the UI (the widgets) but on its *abstract* iconic representation (the graph view). This point implies the following properties:

❑ "Immaterial" objects such as conditions, repetitions, callback functions... can be represented in a homogeneous way in the graph view. The effect of the corresponding actions can be modified interactively by changing the links in the graph view or by calling special-purpose editors.

❑ This principle also *conceptualizes* the way of designing user interfaces as one generally deals with an abstract representation that *produces* the resulting widget view rather than with the widgets themselves. In other words, one can *apply* icons representing various properties (for instance, geometry constraints) to icons that represent widgets and let the system compute and show the physical result of this specification (for instance, lay-out can be computed automatically from such constraints). This difference between this paradigm and the usual one is somewhat similar to the difference between a sophisticated text processor (such as Latex) and a simple typewriter: in the first case one defines generic operators and lets the system apply them, while in the second case all settings have to be performed manually.

❑ Graph views are also quite convenient for *re-designing* interfaces. It is for instance quite easy to select a subtree of objects (selecting an object also selects all its children in the graph view) and move this subtree to another part of the interface or to another separate interface. The corresponding widgets will first disappear from their initial location and then reappear in their final location, all changes in geometry being incrementally and automatically performed. Properties can also be applied to widgets in a direct manipulation style (by editing the links in the graph view). Moreover, the same property object can be applied to several widget objects, thus providing an abstract representation that can control the presentation or behavior of a group of widgets. This partly solves a classical drawback that interface builders are usually blamed for: they require taking decisions that fix the presentation too early in the conception process [10]. The XXL builder allows for "second thoughts" by letting programmers modify their initial decisions and deeply change

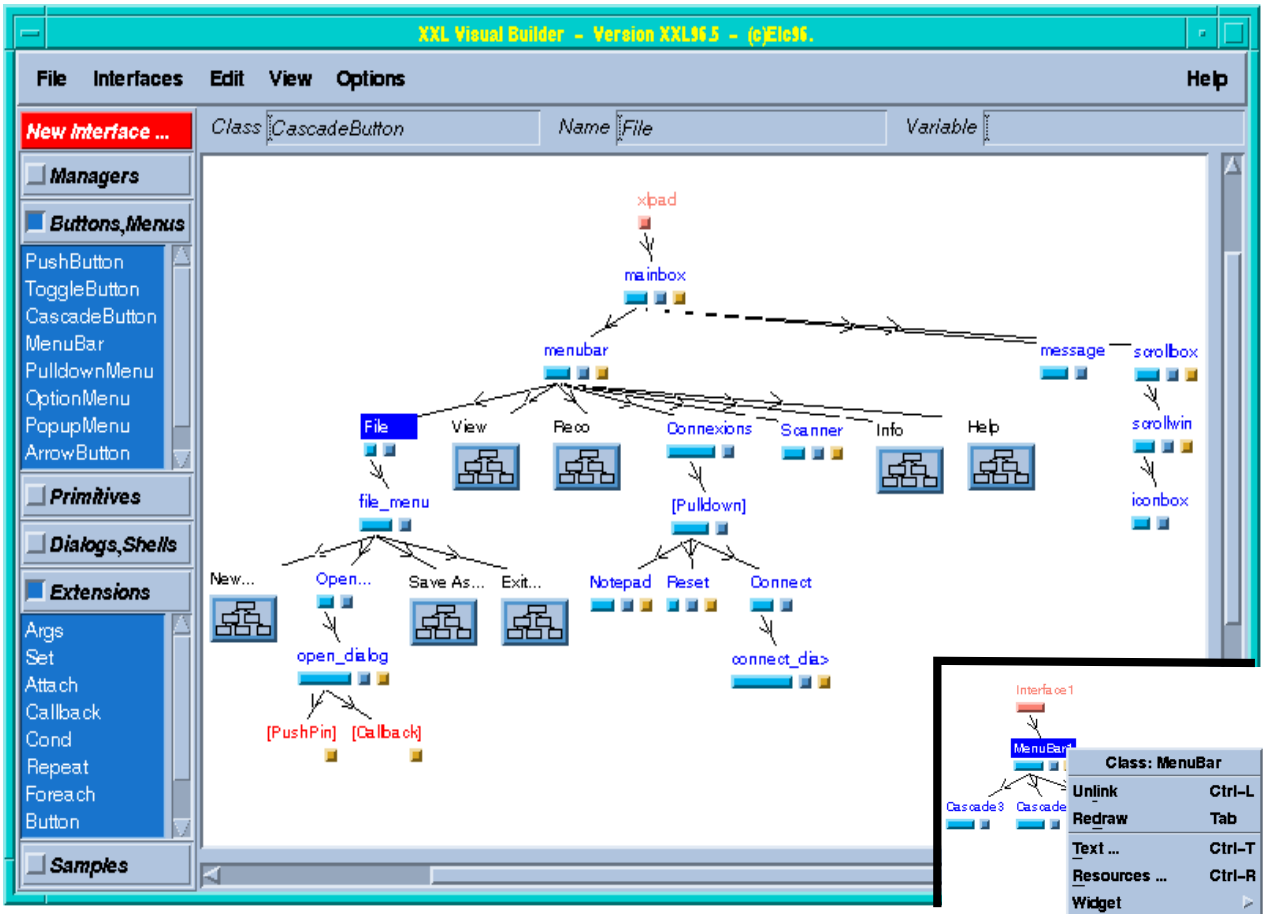

Fig. 2a: The Graph View and the XXL Builder

Fig. 2d: An active icon and its associated menu

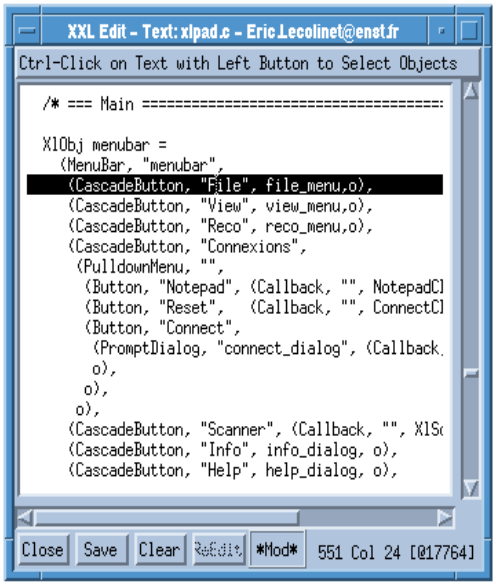

Fig. 2b: The Text View

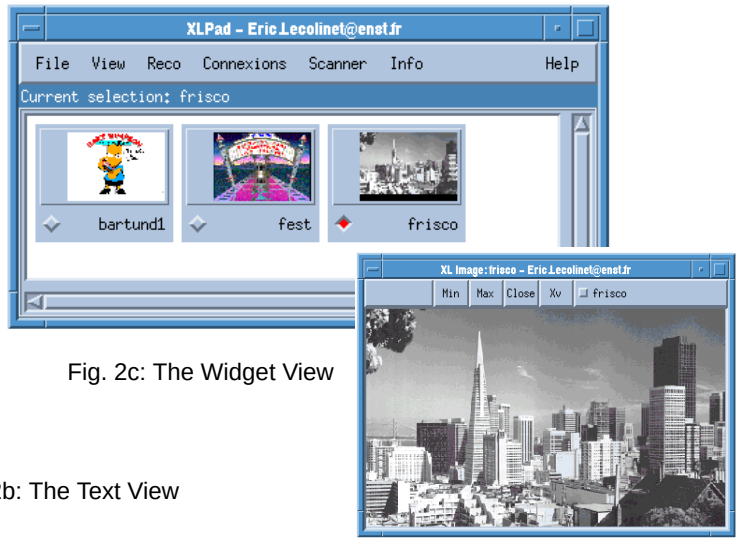

Fig. 2c: The Widget View

Fig. 2: The three views of the main window of an image browser. Fig. 2a shows the graph view and the XXL Builder. Fig. 2b shows the text view (same as Fig. 1) and Fig. 2c the resulting widget view. This application can display several images that can be opened successively at run-time (it can thus process variable amount of data). Each image (right-hand side of Fig. 2c) is displayed by means of a pre-existent XXL Object. An associated icon that shows a reduced image is inserted into the "iconbox" of the browse (A C function for building such icons is shown at Fig. 6). Fig. 2d shows the associated menu of a Widget Object in the graph view. All Icons are active and the graph can be modified interactively.

the structure, behavior or presentation of the UI at any stage of the iterative design process.

❑ These descriptions actually constitute *interface models* that can be instantiated several times. For instance, the three icons that are included in the interface shown in Fig. 2 result from three successive realizations of the same XXL subinterface (whose code can be seen in Fig. 6). The XXL description is actually *mapped* with its *last* widget realization (so, clicking on the icon of a widget object in the graph view will highlight *the* corresponding widget resulting from the *last* instantiation of this representation).

The builder can simultaneously edit all the interfaces (and subinterfaces) that are contained in a C application and in as many XXL Scripts as wanted. Subparts of these descriptions can be moved interactively from one interface to another (and can possibly be moved from a Script File to the C application and vice-versa). The builder also provides a set of predefined "skeletons" (that are actually pre-existent Scripts) that can be modified and integrated into any other script or C application.

The builder also provides an effective way to visualize the graph view. Any tree (or any part of them) can be *minimized* (i.e. hidden) in order to economize on graphical space and *let the user focus* on the part he is currently working on. All icons have a "minimize" button. Clicking once on this button makes all child icons disappear and changes the visual aspect of the selected icon. A second click will revert to the previous state. This mechanism provides hierarchical encapsulation and allows users to visualize interfaces at various levels of granularity (one can have a "global view" of the main components or more specific views of the details of some of these components).

Run-Time Editing

Finally, a rather uncommon property is that modifications can be performed on the interface descriptions *while the corresponding script or C program is running*. Unlike classical interface builders, there are no differentiated "creation" and "testing" phases: the visual builder is started at the same time as the application and can interactively modify it at run-time. The visual edition of the (sub-) interfaces contained in the application is totally dynamical. A notification mechanism makes interfaces available to the builder when created by a C function (and this creation process can take place at any time). These interfaces can then be edited by clicking on their name in a special-purpose menu (which is automatically updated when a new interface is created) or by clicking in an appropriate way on any of their widgets. The builder will then create and show the visual representation of the selected interface (the graph view) and establish the correspondence between the graph of XXL objects and the *original source code* that produced this interface. In other words, the builder is able to find *where* the objects have been defined and to establish the *reverse connection* between these dynamical objects (i.e. created at run-time) and their original textual description. The corresponding source files can then be displayed, and, as said before, any modification performed on the icon trees will then immediately take place on these texts. Besides, texts are incrementally modified in *facsimile way*: their content is actually changed *on the spot* so that their original structure can be preserved (including comments). This mechanism does not require any special declarations by the programmer and does not depend on the location of XXL objects (that can be defined at any place in any of the files that constitute the C program). At the end of this editing phase, it is just necessary to save the modified files and then recompile the application. This application is then ready to use, but can also be modified textually or visually again.

BEHAVIORS

XXL does not aim to provide very complex behaviors that would require a highly specific and complicated language. We believe that standard languages such as C or C++ are perfectly well suited for that purpose and that defining callback functions for specifying complex actions is quite reasonable (although it may be interesting to benefit from higher-level functions than those of the original toolkit, as it is the case here). On the other hand, real interfaces often contain quite a large number of callback procedures that only perform trivial tasks and that make the code look like a "Spaghetti of call-backs" [3]. The main purpose here is to provide simple ways to eliminate most of these unnecessary callbacks. This can be done in the three following ways:

- implicitly
- by conditional evaluation
- by active values.

Implicit behaviors

Certain behaviors are automatically added by XXL when combining certain objects. For instance, any dialog box or pulldown menu that is specified as the child of a button will be automatically popped up when this button is clicked. Such behaviors are dynamically added (or removed) when creating (or destroying) objects (for instance when (un)linking objects in the visual builder).

Conditional Evaluation

Conditional evaluation offers a powerful and simple way to specify many kinds of behaviors. Conditional evaluation means that a subpart of an XXL description will be reevaluated each time a certain event occurs on certain widgets. As XXL descriptions can typically describe how to create, modify and destroy widgets, conditional evaluation provides a rather simple way to do the same thing dynamically.

Fig. 4 illustrates a basic example of this mechanism (see code below). This interface changes the color and the label of the two top widgets by reading a string that is entered by the user in a text field. When the user clicks on the update button, the sub-expression that is included in the **Cond** statement is *reevaluated*. This sub-expression specifies an assignment that gets the string value that was entered in the entry widget, *converts* it to the appropriate types and changes the labelString resource (the displayed label) of widget newname and the background color of widget newcol.

```
(VBox,"vbox",
  (HBox,"",
    (Label,"newcol",o),
    (Label,"newname",o),
  o),
  (HBox,"",
    (TextField,"entry",o),
    (Button,"update",
      (Cond,"",
        (Set,
          "~*newname.labelString", "{~*entry.value}",
          "~*newcol.background",   "{~*entry.value}",
        o),
      o),
     ......
```

Active Values

XXL also provides an *active value* data base. Active values consist of internal variables that provoke some associated actions when their value is changed. They can either be accessed from the interface descriptions or from any function of the application, thus providing an efficient way to share

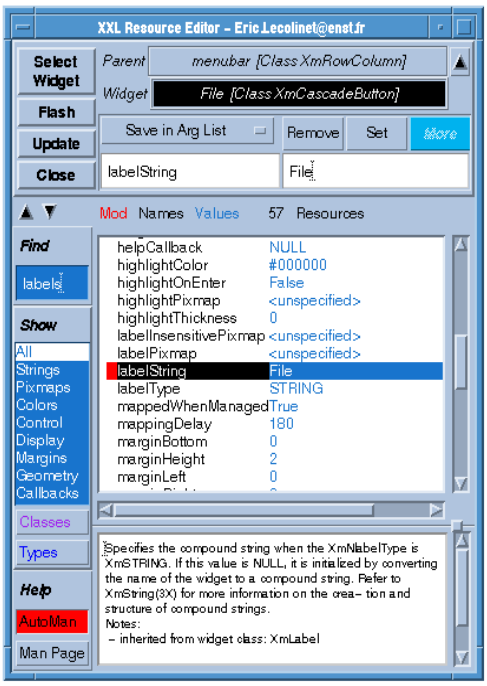

Fig. 3: The Resource Editor of the XXL Builder

Fig. 4a

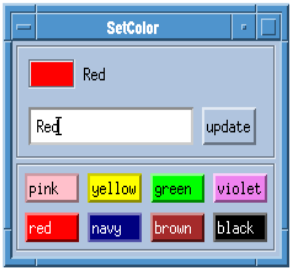

Fig. 4b

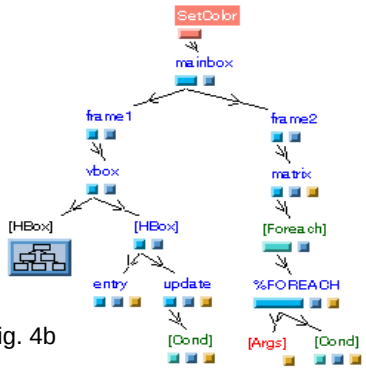

Fig. 4d

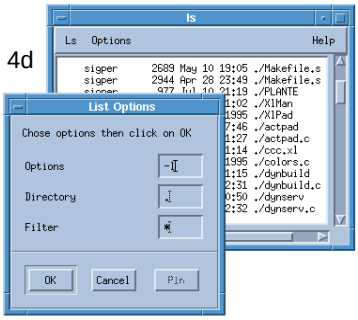

Fig. 4c

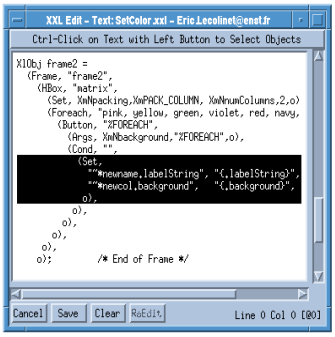

Fig. 4: Conditional Evaluation (4a: widget view, 4b: graph view, 4c: text view of a repetition statement) and Interaction with the Unix Shell (4d).

data and associated behaviors between the interface and the rest of the application. They are also particularly well-suited for dealing with multiple views and synchronizing the presentation with the functional part of the application. Possible associated actions can be:

- the execution of a (generalized) call-back function,
- or the reevaluation of a subpart of an XXL description.

This mechanism can be used in a symmetrical way in the interface and in the functional part (active values can for instance be used to update widgets when data changes in the functional part, or, conversely, a procedure defined in the functional part can be called when a widget is activated). So, active values simulate *non-graphical events* and generalize the notion of call-back functions (see an example in Fig. 6).

Textual References

The previous example is made possible by the use of *textual references*. Textual references can identify any widget (or any widget resource) contained in the application. For instance, the string: `"~*newcol.background"` identifies the background resource of widget newcol. Prefix **~*** specifies that this widget must be found in the *current instance* of the interface. More precisely, the **~** symbol represents the top widget of the interface instance and the ***** symbol indicates the valid "path" to access widget newcol in the widget tree from **~**. Reference paths can also contain the names of intermediate widgets in order to restrict the scope and avoid confusion with possible synonyms.

Widgets and widget resources can thus be seen as a *hierarchical data base*. It is important to notice that search scope can be specified in a more or less precise way (i.e. by either indicating the full pathname to reach a widget or wildcard symbols such as *****). Moreover, this mechanism is also able to deal with *multiple instances* of a same interface: the **~** symbol ensures that a widget will actually be found in the *right instance*, (that is to say the instance on which the event actually occurred). In other words, if the interface shown in Fig. 4 was instantiated several times, clicking on the update button of one instance would modify the fields of the *same instance*. This property is essential when decomposing an interface into several subinterfaces because they can possibly be reused by other components of the same application.

Implicit Type Conversion and Expressions

The previous example also illustrates the *type conversion mechanism*. Widget resources have internal representations which are all different (for instance a "size" will be represented by an integer, a "color" will be a numerical index in a color map, a "string" may even be encoded by means of a special purpose representation in order to take into account various kinds of languages). XXL provides a *double conversion mechanism* that is able to transform most resource values into standard ASCII strings and vice-versa. This single feature can considerably simplify interaction between widgets as it avoids having to write callback functions (whose only role would be to convert resource values into different types). For instance, in the following line:

`"~*newcol.background", "{~*entry.value}"`

resource value contains an ASCII (entered by the user in the entry widget) that is implicitly converted into an appropriate internal type in order to change the background color of the newcol widget. It must be noticed that these conversions may involve complex allocation processes (for instance the allocation of a new color in a color map). In case of failure (for instance if the color name that was entered was incorrect), an error dialog would be popped up and the target resource would not be changed. Lastly, the **{ }** symbols mean that the enclosed string must be evaluated (i.e. all references must be replaced by their actual value). These symbols can take place either on the left or the right hand side. They can also be nested, thus allowing *double (or multiple) evaluation*. The enclosed string can also contain more complex expressions (as shown in the next example). The combination of XXL expressions with the conditional evaluation mechanism makes it possible to define *formulas* that can change dynamically widget resources and other variables.

OTHER FEATURES

Interaction with the Unix Shell

Unix commands can be called from XXL descriptions and their output can be used to set resource values. For instance, Fig. 4d shows a simple interface that executes the "ls" ("list directory") Unix command. Several parameters can be set by the user (the directory to list, a filtering pattern and some command specific options). The appropriate Unix command is generated (by concatenating the entered arguments) when clicking on the "Ls" button and the resulting output is displayed in a text widget. The principle followed here is the same as in the previous example except for the backquote characters whose effect is to send the generated string to the Unix shell and replace this string with the output of the command:

```
(Text,"result",o),
(TextField,"options",o),
(TextField,"dir",o),
(TextField,"filter",o),
/* button that generates and activates the Unix command */
(Button,"Ls",
  (Cond,"",
    (Set,
       "~result.value",
       "`{ls ~options.value ~dir.value/~filter.value}`",
o),o),o),
```

Application Specific Objects

XXL provides several means to help the programmer write application specific objects. First, XXL includes an optional library that includes functions and objects to manage images, icons, graphs and low-level graphical primitives (i.e. lines, rectangles, circles, etc.). XXL provides objects that let the user deal with simple graphical primitives as if they were actual widgets. This library also includes functions that can read and display images (Fig. 2c) and manage the color tables in an appropriate way. Furthermore, it is also possible to make widgets movable under certain conditions (for instance, the active icons of the XXL visual builder can be directly moved by using the mouse pointer). The following example shows how to specify a graph of movable icons. These icons are linked by arrows that follow the icons when they are moved interactively by the user (Fig. 5).

```
XlObj icon =
  (Interface, "icon",
   (HBox, "%INSTANCE",o),
    (ArrowButton, "b", (Args, "background","Red",o),o),
    (Button, "%INSTANCE",
       (MoveHandle, o),
   o),o),o),

XlObj canvas =
  (Canvas, "canvas",
    (Instance, "ico1", icon, o),
    (Instance, "ico2", icon, o),
    (Instance, "ico3", icon, o),
    .....
    (Set,
       "*ico1.x", 200, "*ico1.y", 50,
       "*ico2.x", 100, "*ico2.y", 100,
    .......
     o),
    (DLink, "ico1", "ico2", o),
    (DLink, "ico1", "ico4", o),
    (DLink, "ico4", "ico2", o),
     .......
```

Icons are generically specified by a single subinterface which is instantiated as many times as needed by means of the **Instance** objects (the `%INSTANCE` variable contains the name of the current instance). The **MoveHandle** object means that its parent widget will provide a "handle" to move the whole subinterface (in other words, the icon subinterface can be moved interactively by pushing on this handle widget and moving it). Finally, widgets are linked together by means of **DLink** objects. DLinks are materialized by arrows that are automatically redrawn when the attached widgets are moved by the user.

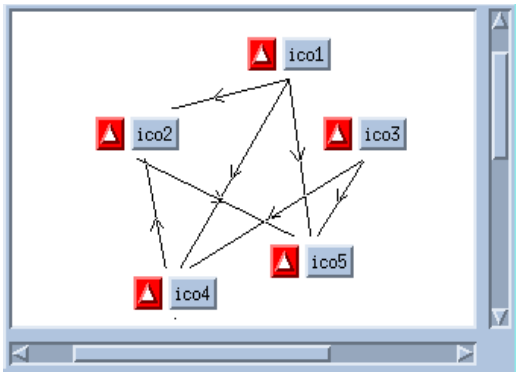

Fig. 5: Movable icons with dynamic links.

Processing Variable Amounts of Data

Mixing XXL descriptions and C constructs can be of great help to process variable amounts of data, data that changes at run time or to create specific objects. For instance, the image browser shown in Fig. 2 can display several images simultaneously (each of them in a different window, which is actually a pre-existent XXL library object). Each time an image is opened, a corresponding icon (that shows a reduction of the image) is also created and inserted into the iconbox widget of the browser. This icon actually results from the multiple instantiation on an XXL subinterface that could be coded in the following way:

```
void AddIcon(char* ima_name, XImage **xima)
{
  static XlObj icon_obj=nil;
  if (!icon_obj)
  icon_obj =
    (Interface,"%iname",
      (VBox,"%iname",
        (Button,"iconpict",
          (Callback,"buildCallback", ShowPict, xima,o)
          (Cond,"", (Set,"%selected","%iname",o),o),
         o),
        (Button,"iconname",
          (Args,"labelString","%iname",o),
          (Cond,"", (Set,"%selected","%iname",o),o),
        o),
      o),
    o);
  XlBuildInstance(xlw(iconbox), icon_obj,
                  "iname", ima_name, nil);
}
```

Fig. 6: The C function that builds the icons of Fig 2c.

This function is called each time a new image is opened. The returned icon is a combination of several widgets: the iconpict button that shows a reduced image, the iconname button that shows the name of the image and a box to contain them. These widgets are produced by instantiating the XXL description which is itself created at the first call of the function. This description is parameterized by `%iname`, an instance local variable that is used to store the name of the corresponding image. The ShowPict function is called each time a new icon is built in order to compute the reduced image that will be displayed on the iconpict widget (C functions can thus be encapsulated into XXL interface to perform complex or specific functions). Clicking on the two buttons would change the value of `%selected`, a global XXL variable which can act as an active value if a behavior is associated to it. For instance, the string displayed by the message widget located in the main window of the browser (Fig. 2) could be updated when clicking on an image icon by modifying its description (bottom of Fig. 1) as follows:

```
  (Label, "message",
      (Cond, "%selected",
         (Set, XmNlabelString, "{Image: %selected}", o),
       o),
  .......
```

The `(Cond,"%selected",...` statement means that the enclosed sub-expression will be automatically reevaluated each time the %selected active value is modified (that is to say each time the iconname or the iconpict buttons are clicked). The same mechanism could also be used to create, open or delete widgets, to execute call-back functions, etc. This formalism is especially convenient because it makes it possible to specify simultaneously the objects and their associated behaviors (the message widget could also be directly modified by the AddIcon function, but nothing would indicate in the description of this object that it is dynamically modified at run-time). This technique is thus very useful when displaying several separate views of the same data as behaviors can be specified separately for each view, thus allowing a modular architecture.

Geometry Management

XXL simplifies the way to deal with the geometry management capabilities of the underlying toolkit. In Motif interfaces, geometry is ensured by specialized widgets, called *Geometry Managers* that can control dynamically the layout of their children widgets in three different ways:

- *Explicit geometry:* the manager does not control the geometry of its children. The positions of these widgets are set by *direct manipulation* (i.e. by moving them with the mouse when using the XXL Builder).
- *Implicit geometry:* the children widgets are automatically aligned in a row, a column, or a matrix. Their size is computed dynamically and their physical position depends on their rank in the widget tree. The XXL Builder ensures this correspondence dynamically (this means that widget geometry is changed when the corresponding icon is moved in the graph view).
- *Constrained geometry*: the children can be attached one to another, or to a virtual and elastic grid. XXL offers a special-purpose object to specify such constraints and a specialized editor to configure them interactively in the Builder. Such constraints are specified as follows:

```
(Attach,
 "menubar",   "top:FORM,     left:FORM, right:FORM",
 "message",   "top:menubar,  left:FORM, right:FORM",
 "scrollbox","top:message, left:FORM, right:FORM, bottom:FORM",
o)
```

Each line specifies top, left, right and bottom attachments to another widget, to a position or to the enclosing frame.

Distributed Interfaces

Textual references can not only identify widgets which are contained in the current application but also widgets of (possibly remote) other applications. This is done by specifying a symbolic name in the pathname of the widget reference that identifies a connection with another application (this connection being opened by means of an appropriate function using the socket mechanism). It is thus possible to specify and modify the resources of the widgets of other applications in the same way as if they were local. This mechanism is made possible by means of implicit conversions between the ASCII and the internal values of the widget resources: values are automatically converted to ASCII, sent through the net and reconverted to the appropriate type. As the same mechanism applies to active value, it is thus also possible to execute callback functions located in remote programs, to perform conditional evaluation of interface sub-expressions or to update variables in a coherent way. Finally, interfaces can be exchanged dynamically between separate applications and interpreted at run-time (for instance a “server” could answer a request from a “client” by sending it an appropriate UI). Such interfaces can of course be defined dynamically by C programs (by producing and sending the adequate textual description). They can also be generated by XXL descriptions by using the conditional evaluation mechanism.

IMPLEMENTATION

XXL object instances control their own representation in the text, graph and widget views. The first two views are of course optional and are only displayed when creating or modifying a graphical interface with the Builder. XXL objects consist in a (hidden) internal representation that is controlled by a set of input and output methods. Input methods control user interaction on the different views and the resulting modifications on the internal representation. These modifications in turn trigger the appropriate output methods in order to ensure coherency between the different views. All views are thus incrementally updated when the internal representation is modified.

Script Interpretation and Textual Representation

The description parser follows the object oriented architecture of the system. XXL classes include a specific method that is devoted to the textual parsing of the corresponding object instances. These methods are actually defined at the metaclass level and are inherited by derived classes in order to ensure style coherency between similar objects. This modularized architecture makes it possible to define classes with new textual styles without having to modify the code for parsing the pre-existent classes.

Another noticeable feature is that the parser does not only interpret the input text but also *memorizes the locations* of object descriptions. The positions of the textual fields that define a given object are stored in the internal representation of this object instance. The textual representation can thus be dynamically updated when object instances are modified (for instance when an object is interactively modified by the user in the graph view). Textual code is thus represented in two complementary ways:

- A viewable text buffer (the text view) that is updated incrementally when object instances are modified,
- An internal representation that is controlled by the corresponding object instances.

This internal representation is hierarchical and only deals with the relative locations of the object descriptions in the text buffer. Each object instance is considered as a local reference frame that controls the lengths of the textual fields that compose it (textual fields being the sequences of characters separated by commas in object descriptions). Absolute positions are computed by scanning recursively the parent objects that effectively embed a given field. This representation is quite efficient for on-line manipulation, and especially when a group (usually a subtree) of objects is moved interactively from one part to another part of an interface or when it is exchanged between separate interfaces.

C Code Compilation and Reverse-Interpretation

It is first important to notice that XXL descriptions that reside in C or C++ code are effectively *compiled*. There is no specific pre-processing stage before the C or C++ compilation and these descriptions *do not require* run-time interpretation ***except*** when using the Builder to modify them. More precisely, the system works as follows:

- XXL descriptions figuring in C code are always compiled and executed as ordinary C statements in any case.
- But these descriptions are *post-interpreted dynamically* by the Builder *when needed* (that is to say when modified interactively by the user).

This means that the system is able to post-interpret the C source code from the internal representation of the objects that have been previously created. We call this "*reverse-interpretation*" as the parsing of the source code is guided by the run-time process (which actually results from the compilation of the same source code). In other words, XXL can be seen as a (C compatible) compilable language that is able to post-interpret its own source code dynamically at run-time in order to establish the reverse correspondence between the internal (i.e. binary) and external (i.e. textual) representations of the produced objects.

As a consequence, XXL objects can dynamically modify at run-time their own external representation (i.e. their textual description inside the C source code). This point explains why XXL interfaces can be modified dynamically while the application is running. The Visual Builder can be seen as a tool interface that gives access to the XXL objects contained in an application, these objects being able to control their own representations when created, deleted or modified.

The following paragraphs detail the compilation, execution and post-interpretation stages:

Compilation. XXL descriptions are transformed into nested variadic functions by the C pre-processor. `ClassName` tokens expand to: " `XlNew(XlcClassName` " where `XlNew` is the C function that creates a new object instance and `XlcClassName` is a pointer to the hidden representation of the class. The " `o` " token expands to: " `NULL)` " where `NULL` acts as a list terminator by indicating to the `XlNew` function that no more arguments are to be read. For instance, the following description:

```
(HBox, "mybox",
      (Label, "hello", o),
      (Button, "clickme", o),
 o),
```

would expand to:

```
(XlNew(XlcHBox, "mybox",
      (XlNew(XlcLabel, "hello", NULL)),
      (XlNew(XlcButton, "clickme", NULL)),
 NULL)),
```

Omitting the " `o` " terminator leads to unbalanced expressions that can not be compiled. This avoids careless mistakes when writing C code directly. In the same way, it must be noted that the homogeneity of XXL descriptions (almost all arguments share the same C type) makes them fairly easy to write while avoiding mistakes.

Execution. Executing a description just creates the internal representations of the objects it contains and initializes the parenthood between these objects (parenthood being implicitly defined by the level of embedding in the description). No widget is created at this time: widgets are produced by applying the XlBuild (or similar) function to the internal representation of the whole interface once it is complete. This scheme has several interesting properties:

- There is a clear separation between XXL objects (the "abstract representation") and actual widgets (the physical implementation).
- Textual descriptions can be read inside-out or outside-in (the first case corresponds to C execution and the second case to script interpretation) as the resulting widgets are globally created at the end.

Finally, it must be noted that type checking between XXL objects is performed at this stage or when modifying descriptions dynamically with the Builder. Objects classes contain a property that defines the list of the legible child classes (or child superclasses) of this object.

Reverse-Interpretation. XXL applications can be compiled in two different ways: an optimized mode is available for final use, while the "development mode" should only be used to edit interfaces with the Builder. In this second mode, descriptions are expanded in a slightly different way than what is shown above in order to provide object instances with the file name and the line number where they are located in. Objects are then post-interpreted by parsing the source code at the locations (file name and line number) stored at execution time. Parsing is performed by means of the same interpreter as for reading scripts except that no object is created (the aim of reverse-interpretation being just to find the exact location of the textual items that correspond to the dynamic objects). XXL objects and descriptions can then be modified dynamically by the builder once the correspondence between the source text and the resulting objects is obtained.

CURRENT STATUS

The XXL system has been implemented under the SunOS4 and SunOS5 operating systems. All Motif 1.2 widgets can be manipulated through the XXL classes and it is thus possible to create the same kind of interfaces as would be done by using this toolkit directly. Furthermore, the system also includes classes and library functions for dealing with images and low-level graphical primitives and provides direct manipulation capabilities. A set of core objects is also provided for dealing with repetitions, conditional evaluation, widget presentation and behavior and for decomposing complex UIs into smaller subinterface components. Finally, various functions are available for dealing with distributed interfaces and for accessing and modifying widgets dynamically with the same capabilities (textual references, implicit conversions, active values...) as in interface descriptions.

The XXL system has been tested and used for realizing various applications such as: several image processing and pattern recognition tools, a visual interface with a sensitive map for accessing a subway data base, a tool for editing hypertextual links in book manuscript images and various students' projects. However, the best example of XXL application may the one that is the object of this paper, that is to say the XXL Builder.

ACKNOWLEDGMENTS

The author would like to thank François Tête for his contribution to the implementation of the system.

REFERENCES

1. Avrahami G., Brooks K., Brown M., A Two-View Approach to Constructing User Interfaces. Computer Graphics, Vol. 23, No. 3, July 1989.

2. Morse A., Reynolds G., Overcoming Current Growth Limits in UI Development. Communications of the ACM, Vol.3 6, No. 4, April 1993.

3. Myers B. Separating Application Code From Toolkits: Eliminating The Spaghetti Of Call-Backs. ACM Symposium on User Interface Software and Technology, pp. 211-220, Nov. 1991.

4. Myers B.A., Challenges of HCI Design and Implementation.. ACM Interactions, Vol. 1, No. 1, pp. 73-83, Jan. 1994.

5. Myers B.A., User Interface Software Tools. ACM Trans. on Computer-Human Interaction, Vol. 2, No. 1, pp. 64-103.

6. Myers B, McDaniel R., Mickish A., Klimovitski A., The Design for the Amulet User Interface Toolkit.. Proc. Human-Computer Interaction Consortium meeting, Feb. 1995.

7. Ousterhout J.K., Tcl and the Tk Toolkit. Addison Wesley, 1994.

8. Singh G., Green M., Automating the Lexical and Syntactic Design of Graphical User Interfaces: The UofA* UIMS. ACM Transactions on Graphics, Vol 10, No 3, July 1991.

9. Szekely P., Luo P., Neches R., Beyond Interface Builders: Model-Based Interface Tools, proc. INTERCHI'93, April 1993, pp. 383-390.

10. Wiecha C., Bennett W., Boies S., Gould J., Greene S., ITS: A Tool for Rapidly Developing Interactive Applications. ACM Transactions on Information Systems, Vol 8, No. 3, July 1990.