Designing GUIs by Sketch Drawing and Visual Programming

Eric Lecolinet

Ecole Nationale Supérieure des Télécommunications (ENST)
46 rue Barrault, 75013 Paris, France
elc@enst.fr http://www.enst.fr/~elc

ABSTRACT

This paper presents a new interactive UIDE that is based on visual programming and constrained sketch drawing. At the early stages of the iterative conception process, GUIs are interactively designed by drawing a "rough sketch" that acts as a first draft of the final description. This drawing is interpreted in real time by the system in order to produce the corresponding widget view (i.e. the actual visible GUI) and a graph of abstract objects that represents the GUI structure. This graph can then be easily modified by mixing visual and textual programming in a fully iterative and incremental way. This system is also based on the use of *generic objects* which are dynamically instantiated into actual widgets according to their structural and functional context. This scheme makes it possible to define very generic GUIs that can then be deeply refined in an efficient way at any stage of the conception process.

KEYWORDS: user interface design, interface builders, visual programming, sketching, visual / textual equivalence

INTRODUCTION

A classical reproach made against interactive GUI builders is that they require designers to make detailed presentation choices too early in the UI conception process. These tools generally force interface designers to deal with specific graphical objects (we will call them *widgets*) and to specify more of the design details than are needed at this early stage[7]. Moreover, GUI builders are quite efficient for designing standardized interfaces that are limited to menus and dialog boxes but generally provide little support for creating application specific objects (for instance an interactive graph editor) or to process data that changes at run time (for instance a dynamic file manager). Writing source code is thus required for specifying such behaviors. Unfortunately, this leads to another major problem: most of these tools do not allow the programmer to freely edit the generated source code, or else the tool can no longer be user for later modifications. This limitation breaks the iterative conception process when designing complex interactive systems.

Several strategies have been proposed to solve these problems. Model based UIMS [6, 7] generate most of the presentation automatically from a high-level specification of the interface. While this idea is appealing, these are essentially text-based approaches that require the knowledge of a special-purpose language. Moreover, the lack of designer control may lead to generated GUIs that are not perfectly suited to the application. Certain systems also include powerful interactive tools [8], but most of them only allow for "cosmetic" final modifications at a later stage. Another approach consists in providing a separate interactive way of designing interfaces at the early stages of their conception. For instance, the SILK system [3] provides an interactive tool that allows designers to quickly "sketch" an interface by using an electronic pad and stylus. The idea is to let designers draw early interface ideas in the same way as they would do on a piece of paper. But this electronic sketch will also produce a fully operational interface.

PROPOSED APPROACH

We propose here an hybrid approach which tries to mix several aspects of interactive GUI builders, model-based systems and interactive sketching. This new system lets designers produce generic interface specifications in an interactive way, either by textual or visual programming or by constrained sketching. The conception process is fully iterative and is consistent from the very early stages of design to the realization of the final application. The full system provides four different views of the GUI that correspond to various stages of the conception process (or to the various level of expertise of the designers involved in this task):

1. The *Sketch View* (Fig. 1) lets designers conceive a "first draft" of the GUI by drawing a "rough sketch" that will become the base of the final program. The system will "interpret" this drawing and will try to infer which actual objects are needed in order to let the designer focus attention on the global layout of the GUI without having to take care of implementation details.
2. The *Graph View* (Fig. 2) provides the designer with a visual iconic representation of the *structure* of the GUI. This representation is fully editable and can be modified interactively by direct manipulation. This graph is not a mere widget tree but a true abstract representation of the GUI that can include (and represent visually) non-widget components such as properties, call-back functions, control statements, interactive behaviors, etc.
3. The *Widget View* shows the actual "concrete" GUI that is normally visible on the screen. This view only shows the *"surface"* of the GUI (its visible part) by opposition with the graph view.
4. The corresponding source code is displayed in the *Text View*. The source code is generated in an incremental way in order to reflect all modifications performed on the other views. Conversely, this view can also be changed interactively by parts: the designer can edit the source code of any subpart of the GUI and change it "on the spot." Syntactic and semantic errors are checked dynamically and the other views are then updated in real-time.

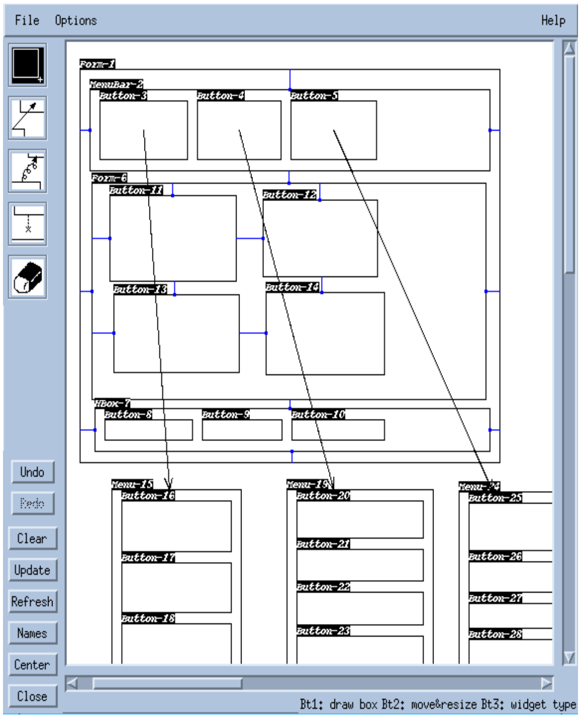

Fig 1: The Sketch View

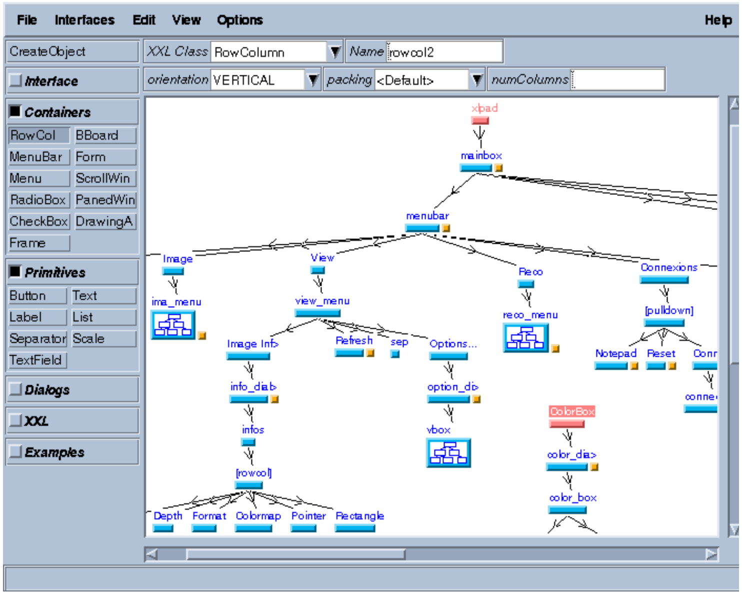

Fig. 2: The XXL Builder and the Graph View

Fig 3a

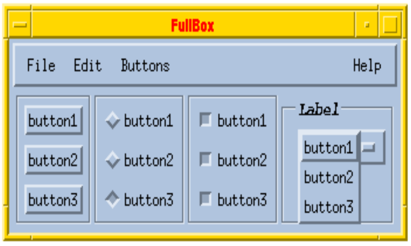

Fig. 3: Generic Objects and Widget Generation. These graphical components have quasi equivalent specifications that can be changed one into another by modifying the container type (Fig 3a: the widget view, Fig 3b: an extract from the corresponding C source code)

Fig 3b

```
(VBox, "menu",
    (Button, "button1", o),
    (Button, "button2", o),
    (Button, "button3", o),
o)

(Menu, "menu",
    (Button, "button1", o),
    (Button, "button2", o),
    (Button, "button3", o),
o)
```

The sketch view is only used at the early conception stages. The resulting widget view is produced in real time so that designers can immediately see the result of their drawing and correct it iteratively (the system providing full undo capabilities). The graph view and the text view are made available to the designer once the sketching stage is over. Visual and/or textual edition can then be performed to refine the GUI, add call-back functions or any other means of interaction. The graph, widget and text views are linked together and are incrementally updated. The graph and the text views provide dual representations of the program that would produce the corresponding widget view (in that sense, this widget view can be seen as the “result” of the specifications contained in these two other views). Furthermore, there is a one to one correspondence between the various representations of a same object. For instance, control-clicking on an object representation in any view will highlight its other available representations in the other views.

Most interactions for building the GUI take place on the graph view, which acts as an “abstract” intermediate representation of the actual widgets. There are several advantages in dealing with an abstract representation rather than with the actual widgets themselves. First, as said before, it is possible to represent non-widget components visually and handle them in a direct manipulation style. Second, this visual representation actually corresponds to a program (which is displayed in the text view). Third, this intermediate level of representation brings abstraction capabilities that are missing in systems based on a pure WYSIWYG paradigm (i.e. systems that let the user manipulate “concrete” widgets directly). An abstract representation makes it possible to specify “immaterial” behaviors and constraints (for instance for controlling the lay out of the GUI dynamically) and to deal with GUI components in a quite generic and uniform way. This last point leads to the concept of *generic objects* which is presented in the next section.

At last, the system provides complete textual and visual equivalence. This means that textual edition can not only take place while using the interactive builder but also at any stage of the conception process. The generated source code (which is in the C language) remains visually editable by using the interactive builder, even if this code is textually modified by the programmer *after* the GUI generation stage. For instance, a generated GUI could be textually included into C or C++ constructs in order to produce a dynamic behavior, and would then remain visually editable by calling the builder again. We believe that this feature can considerably ease the design of applications that deal with application specific objects or data that changes at run-time as it let designers make use of the whole power of a standard programming language while using an interactive GUI builder.

GENERIC OBJECTS AND WIDGET INSTANTIATION

The system presented here is called *XXL* and relies on a former version that was presented in [4]. The present version includes a set of *generic objects* which are dynamically instantiated into different actual widgets according to their structural and functional context. Moreover, object classes can be changed interactively in a direct manipulation style and these changes are recursively propagated to the children they contain. This conception scheme makes it possible to hide many low-level details to the designer and remodel GUIs after their initial creation in quite an efficient way. For instance, a box containing a set of buttons can immediately be changed into a radio box, a dialog box, a menu bar or a menu by only changing the type of the container object, its children being then automatically modified by the system (Fig. 3). Then, linking interactively a generic menu to a menu bar button, to a pull-down menu button or to another object would make it respectively behave as a pull-down, a cascaded or a contextual pop-up menu. Moreover, error checking is performed in real-time in order to detect possible incoherences and to warn the user in an appropriate way.

At last, it can be noted that the underlying textual specification is somewhat similar to the one used by certain model-based UIMS except that the objects can be manipulated interactively in a visual programming style (Fig. 3b). The XXL system is basically based on a hierarchical box / item model that resembles some of the interface specifications that can for instance be found in the ITS system [8].

SKETCH DRAWING

The strategy adopted for drawing is quite different from SILK and other systems [3]: free hand drawing is not allowed and the drawing is electronically constrained to a very small number of possible figures. All objects are specified by drawing rectangles at a certain location in the sketch. Other possible figures include three different kinds of arrows that act as behavioral or topological links. The object types are automatically deduced by considering structural the constraints between objects and their locations in the drawing. This strategy has several advantages:

- It avoids the complex pattern recognition stage that would be needed for recognizing free hand drawing. In addition, recognitions errors or rejections are also avoided.
- The designer can only draw legible figures (rectangles or arrows). The system do not let users think they can draw complex fancy shapes (that would actually be rejected or misrecognized by a pattern recognition system).
- By opposition to free hand drawing, constrained drawing does not require specific hardware (such as an electronic pen and pad). Moreover, people are quite used to this kind of mouse interaction.

Thus, by using a “faked metaphor” (users draw sketches as they would do in reality, by not in the same “material” way) we can ease interaction, avoid errors and ambiguities and avoid having to use specific hardware.

Sketch Interpretation

The first rectangle drawn in the sketching area is implicitly considered as a “main box” (that will not necessarily be the actual main window of the final application but can be included into another object at a later stage). Then, an included horizontal rectangle, located at the top of this main box will automatically be seen as a menu bar by the system (Fig.1) . Drawing enclosed rectangles inside this menu bar will generate menu bar buttons. Such buttons are supposed to open pull-down menus which are created by drawing vertical boxes outside the main box. These menus can contain buttons and refer to cascaded menus that are designed in the same way. Menus (and dialog boxes) are attached to their opening button by drawing a “behavioral link” between these two components. The same principle applies for the other kinds of objects. It is for instance possible to create dialog boxes, to nest container objects within a same window and to lay out primitive objects in an appropriate way. The system proposes default rules for lay out management that can be changed interactively. Objects can be automatically aligned or can be layed out by specifying graphical constraints which are materialized by “topological arrows” that can be set and changed in a direct manipulation style.

This way of designing GUIs favors the use of spatial topological constraints instead of fixing absolute x, y coordinates by moving and resizing widgets directly with the mouse. This leads to a more flexible representation that can evolve dynamically at run-time when the final user resizes the windows or customizes the application (for instance by specifying larger fonts). The use of such constraints is rather easy and natural here because they are either deduced from the drawing, or explicitly drawn in a simple way. Thus, it is interesting to notice that the drawing performed by the user is not a WYSIWYG but a *logical* representation of the GUI. The actual GUI will not exactly look exactly the same as the drawing but will follow the logical constraints specified by the designer. It is up to the graphical system to adjust and lay out the corresponding widgets in an appropriate way.

CURRENT STATUS

The system has been fully implemented and relies on the X Window system and the Motif toolkit. The visual/textual builder has been used for realizing various tools and students’ projects at our institute. The XXL environment has also been used for creating and refining the interactive builder itself. This environment also includes some other features (which are detailed in [4]) such as an interpretable scripting language, the ability to edit GUI at run-time while they are being executed, and a mechanism for dealing with migratory interfaces over a network. It is (partly) available at URL: http://www.enst.fr/~elc.

ACKNOWLEDGMENTS

The author would like to thank F. Tête, P. Agin, L. Ulmer, O. Delahaye and G. Lecourt for their contributions to the implementation of the system.

REFERENCES

1. Avrahami G. et al., A Two-View Approach to Constructing User Interfaces. Computer Graphics, Vol. 23, No. 3, July 1989.
2. Bodart F. et al., Towards a Systematic Building of Software Architecture:The Trident Methodological Guide, Workshop on Design, Specification, Verification of Interactive Systems, pp. 237-253, 1995.
3. Landay J., Myers B., Interactive Sketching for the Early Stages of User Interface Design,Proc. of the CHI Conference, 1995.
4. Lecolinet E., XXL: A Dual Approach for Building User Interfaces,Proc. ACM Symposium on User Interface Software andTechnology (UIST), pages 99-108, Seattle, USA, Nov. 1996.
5. Myers B.A., User Interface Software Tools. ACM Trans. on Computer-Human Interaction, Vol. 2, No. 1, pp. 64-103, 1995.
6. Singh G., Green M., Automating the Lexical and Syntactic Design of Graphical User Interfaces: The UofA* UIMS. ACM Trans. on Graphics, Vol 10, No 3, July 1991.
7. Szekely P., Luo P., Neches R., Beyond Interface Builders: Model-Based Interface Tools, proc. INTERCHI'93, April 1993, pp. 383-390.
8. Wiecha C. et al., ITS: A Tool for Rapidly Developing Interactive Applications. ACM Trans. on Information Systems, Vol 8, No. 3, 1990.