IUP/LED: A Portable User Interface Development Tool

C.H. Levy, L.H. de Figueiredo, M. Gattass, C.J.P. Lucena
Departamento de Informática
Pontifícia Universidade Católica
22453-900 Rio de Janeiro, RJ,
Brazil
{levy,lhf}@icad.puc-rio.br; {gattass,lucena}@inf.puc-rio.br

D.D. Cowan
Computer Science Department & Computer Systems Group
University of Waterloo
Waterloo Ontario,
Canada
N2L 3G1
dcowan@csg.uwaterloo.ca

October 31, 1994

1 Introduction

User Interface Toolkits (UIT) and User Interface Management Systems (UIMS) [HH89] are two major classes of software for constructing graphical user interfaces. A UIT is a library of interface objects which implement different interaction techniques with the user; tools in this class are available as functions that are called by the application in order to create and control the dialog with the user. Some UITs are: XView [Hel90]; OSF/Motif [Fou91]; OLIT [Inc90]; SDK for MS-Windows [Pet90] and the Macintosh User Interface Toolbox [App85]. UITs frequently offer tools to simplify the description and the composition of interface objects. These tools range from simple resource languages to graphic editors that build interfaces through direct manipulation of interface objects. However, management of the interaction with the user must be programmed as part of the application, even when supported by a toolkit.

In contrast, a UIMS is a group of high level interactive programs that are used to design, create prototypes, execute, evaluate and maintain user interfaces, while integrating everything with a simple dialog [HH89]. Two such UIMS are the University of Alberta UIMS [Gre85] and DMS [HHE84]. In some sense, UIMSs encompass UITs since they allow not only the description and composition of the interface objects, but also the specification of the control of the user interaction sequence [MvD91]. UIMSs assume that application development is a joint undertaking of two experts: one expert in the area of the application domain and another in user interfaces. The first expert should solve the computer application problem, while the second expert should work

with the related psychological, cognitive, ergometric and linguistic human factors so as to offer an appropriate interaction between the user and the application [HH89].

Many commercial interface systems, such as Visual Basic [Mic92], only support the construction of dialogs; they do not allow any control over the interaction sequence, which is then programmed as part of the application. Therefore, these systems cannot be classified as UIMS, even though they are integrated systems. Figueiredo et al. [FSGC92] proposed a tool for the automatic generation of interfaces for data entry to programs for engineering simulation and optimization. This tool, even though it is not an integrated application, can be considered as a UIMS because it contains all aspects of user interface design.

Recently, a new generation of UIMS, integrated with knowledge bases has appeared: they are the UIDS (User Interface Development Systems) [Hix90]. UIDSs use knowledge bases on interface design techniques and software design principles which help the interface specification process [LCCC90]. In contrast with UIMSs, which need a software specialist, this integration allows the user to be the interface specialist.

These three types of tools for interfaces support the construction of interactive programs, but they do not attack two important aspects of the problem:

- These tools normally operate within a specific computer environment and do not support multi-platform applications;
- They do not provide a layout definition model which is independent of coordinates and sizes of the interface objects. Thus, the user is usually forced to define coordinates and sizes for each interface object. This difficulty can be minimized by using interactive graphic editors for layout construction through direct manipulation. Nevertheless, these editors do not support the automatic recalculation of the positions and sizes of each interface object when the final user makes an alteration to the dialog size.

Since UIMSs normally use UITs, the construction of a portable UIT supporting an abstract layout model forms a solid base for the construction of user interface toolkits which encompass these important objectives. The goal of this paper is to describe the design and implementation of a portable user interface toolkit named IUP/LED. IUP/LED is portable to many different computer environments, and our experience indicates it is easily used by both programmers and end users. The main characteristics of IUP/LED are:

- dialog composition through an abstract layout description;
- a dialog description language (LED) compiled at the same time as the application execution;
- a look-and-feel appropriate to the operating environment;
- and expandability.

2 Portability of Graphic Applications and Interfaces

In order to maximize use and return on investment, an interactive program should be capable of executing under many different operating environments and graphical user interfaces with appropriate look-and-feel. Such systems as MS-DOS, MS-Windows, OS/2, Macintosh, Motif/X11, Open

Look/X11, IBM VM/CMS, and VAX/VMS may need to be supported. Since these environments are quite different from each other, this goal is difficult to achieve without adequate tools for developing portable programs, especially interactive graphic applications. The main factors affecting portability are:

- hardware differences, such as byte ordering and addressing;
- operating and file system differences, such as multiprocessing capabilities and case sensitivity. Even systems that follow the same basic standards, such as BSD 4.x or System V Unix, have subtle differences that can hamper portability if the developer is not careful [Fre88];
- compiler differences, such as the default size of an integer which may be important for programs that handle binary files. In addition, each compiler offers a proprietary function library which can introduce incompatibilities;
- graphic devices that can support different numbers of colors and may or may not use a graphic processor;
- different application programmer interfaces (APIs) for each type of graphical user interface system.

A detailed discussion of how these factors impact portability is presented in [Bla88, Dur90]. The impact of changes on operating systems and programming languages can by minimized by using de facto standards such as Posix [IEE88] and ANSI C [KR88]. If any platform dependent code still remains, a simple strategy of isolating the dependent code and documenting its functionality can be used so that a future implementation in another operating environment is easier. However, the diversity of interface systems does present some interesting problems that are discussed in the next section.

2.1 Portable user interfaces

Programming graphical user interfaces, such as Microsoft Windows, Presentation Manager, Macintosh Toolbox, Motif, and Open Look, is conceptually the same task for the various platforms, since most graphical user interface systems use the desktop metaphor and corresponding applications usually have a similar look-and-feel. However, the APIs are quite complex, and because of the many differences in the various toolkits, application programmers often must be experts in several systems.

The best solution to this problem would be to use a de facto standard interface system. Since there are no such systems available, an alternative solution would be to use an international standard, but unfortunately efforts in this direction have not yet been successful. The only other way to avoid dependencies on specific computer platforms is to use proprietary tools for building portable interfaces such as CIRL/PIWI [CDEP92a, CDEP92b] or XVT [Roc89]. Even though an application built with these tools becomes device-independent, it now depends on the manufacturer of the tools, because they are proprietary and do not follow an international standard.

There are many strategies for building a portable interface tool. The simplest approach is to develop a tool that provides the functionality which is common to all supported interface systems.

Even though this strategy is simple to follow, applications that use this strategy typically have inadequate support for color and character fonts. Another disadvantage is that the application has to contain interaction mechanisms, such as list selection boxes, which do not appear in all interface systems.

A different strategy involves porting a complete interface system to all environments, without using native interface systems. A characteristic of this solution is that all applications have the same look-and-feel in all computer environments. This may be an advantage when the user uses the same application in different environments, but when the application is used by one user in only one machine, the look-and-feel of the application will probably not be consistent with the look-and-feel of the other applications provided by other suppliers.

The creation of a virtual toolkit, implementing a portable user interface metaphor, is a good strategy for the development of portable tools. A virtual toolkit maps abstract interface elements to the native system elements of the local environment of an application. Thus, the application inherits the look-and-feel of the native system. This solution can increase the productivity of users who execute many applications on the same machine, since users may use techniques already learned in that environment. On the other hand, the user of only one application in different machines will suffer, for this user will have to learn how the program works in many environments. However, we believe this situation is not as common.

The major problem with this strategy is creating a portable metaphor. Such a metaphor is defined by the IUP toolkit, where IUP is an acronym for Portable User Interfaces, in Portuguese. We describe IUP in Section 4. This toolkit supports both fixed look-and-feel and native look-and-feel, because, in addition to drivers for many common interface systems, a complete portable interface system was written.

3 Comparison of De Facto Standards

In order to develop a virtual toolkit, we should be aware of existing interface systems. In this section we examine those that have become de facto standards including: Motif, Open Look/XView, MS-Windows and Macintosh Toolbox.

A comparison among these interface systems makes it clear that Motif is the most complete environment for dialog specification. Through its User Interface Language (UIL), it is possible to separate the programming of interface elements from the program code, allowing rapid prototyping. On the other hand, Motif has so many elements, the UIL is so powerful, and there are so many different forms to compose dialogs that the application programmer may have difficulty making reasonable choices. XView, on the other hand, is a very compact toolkit that offers an easily learned relative positioning model to compose dialogs. Nevertheless, since it does not provide a dialog specification language, it is not easy to separate the user interface from the application, increasing prototyping time. Macintosh and MS-Windows do not offer any abstract model for defining layouts, forcing the programmer to know the size and position of each interface element while drawing the dialogs to scale.

In order to avoid having to design dialogs to scale numerically, Macintosh offers a graphic editor that allows the construction of dialogs through direct manipulation of interface elements. In MS-Windows, this type of editor is provided by programs that permit the specification of certain kinds

of dialogs, such as the Borland C++ compiler, Microsoft C compiler, Visual Basic, and Microsoft Access. These toolkits help in specifying a layout, but they generally do not use an abstract model of a layout. As a consequence, the visually created dialogs do not react automatically to any change in their size.

There are interactive dialog editors for other interface systems, but almost all of them specify the dialog layout by using a concrete rather than an abstract model. Exceptions are the *ibuild* dialog editor InterViews [LVC89] and the one in FormsVBT [ABB89]: both use the boxes-and-glue paradigm as an abstract layout model. In addition, some toolkits, such as Guide for Open Look with XView, create descriptions that need to be translated, compiled, and linked to the application before its execution, thus limiting their utility for rapid prototyping.

The FormsVBT dialog editor provides two views of the dialog specification: a text description of the dialogs in a specific language and a graphical representation that can be directly manipulated by an interactive editor. The user may interact with both views, and changes in one of the views is reflected in the other. This combination has the advantages of both text description and direct manipulation of the WYSIWYG type, without any of their limitations.

As a solution to layout description, IUP/LED proposes an abstract layout model based on the boxes-and-glue paradigm of the TEX text processor [Knu84], and similar to the ones used in InterViews and FormsVBT. As we shall see in the Section 4, this model allows dialogs to be specified without explicitly defining the position of interface elements, enabling automatic repositioning of the interface elements when the size of the dialog changes.

In interactive programs, the communication between the application and the user is bidirectional through the dialogs. The application builds a dialog and makes it visible, and then the application waits for and reacts to user actions. There are two basic models for integrating an application with user actions managed by dialogs: callbacks and events. The callback model associates an application function with each possible action over the interface elements. The toolkit captures the event generated by the user and executes the corresponding application routine. This model is used by XView and by the toolkits based on the X Window Intrinsics Toolkit, such as Motif and OLIT. Other systems, such as Xlib, MS-Windows and the Macintosh, use the event model. This model queues all events generated by the user; the application takes the events from the queue, interprets them and calls the appropriate routines.

The callback and event models are equivalent. The callback model may be converted to an event model by associating all the actions with a single application routine that would handle all events. Conversely, the event model can be converted to a callback model by building a software layer containing an event handler routine and the routines responsible for associating the application with corresponding user events.

The event model may be inefficient because all events are queued, even those that are not handled by the application. For highly interactive applications that need to provide fast response, this might be a limiting factor if the events with no direct meaning for the interface happen too often. The most common examples of these events are those events generated as the user moves the mouse without pressing any button. In this context, when the user moves the mouse, there is no intention of interacting with the dialog over which the mouse passes, but only to position the mouse over an interface element to start an interaction. Nevertheless, when it passes a dialog, many events are created, such as: enter window; many mouse movements; leave window. The X window system uses a client-server architecture, and the application may not be executing on the

same machine as the one with which the user is interacting. This makes the generation of useless events still worse, because it not only overloads the application but also slows down network traffic, affecting all running applications. The solution adopted in Xlib is to allow the application to select the events to be queued. MS-Windows, on the other hand, does not provide a way to avoid the generation of useless events, probably because the underlying operating system (MS-DOS) does not handle networks.

IUP/LED uses the callback model. This model was chosen because it allows a more natural method of programming and avoids the problem just described.

4 The IUP/LED System

IUP/LED is a portable user interface system composed of a dialog specification language (LED) and a small group of functions (IUP). IUP/LED was designed to be:

- portable, allowing interfaces to be built for a variety of platforms, ranging from MS-DOS text mode to Unix/Motif;
- simple and quickly learned;
- extensible;
- and open, in the sense that the programmer should be able to combine calls to IUP with calls to the native toolkit.

The IUP/LED system was designed to be portable as described in [CW84]; in other words, the installation of IUP/LED in a new computer environment requires much less effort than the effort required to rewrite IUP/LED for the new environment. This goal was achieved through the combination of the portability strategies described in Section 2.1. In this way, IUP/LED can be implemented in environments as different as MS-DOS in text mode and MS-Windows.

Simplicity was an important factor in the design of IUP/LED. The interface elements are created and manipulated consistently by the application through a small group of functions; two IUP functions that set and query the associated attributes of the interface elements are of primary importance. The LED language, an expression language with a very simple syntax, is used to create the static description of the dialogs.

In IUP, an application is exclusively formed by a group of potentially concurrent dialogs. A dialog is formed by interface elements that interact with the user, capturing and exhibiting information manipulated by the program. Writing an application consists of specifying its dialogs (possibly by using LED) and implementing the associated application routines.

The main philosophy behind LED is the distinction between abstract and concrete layout. To describe a concrete layout of a dialog is to describe the geometric position of each interface object that composes the dialog. On the other hand, to describe an abstract layout of a dialog is to describe the relative positions of these objects. Frequently, the programmer has a clear idea of the abstract layout, while the calculation of the concrete layout is complicated, tedious and error-prone. Moreover, if a dialog's layout is described abstractly, then it is simple to recalculate the concrete layout when the size of the dialog has been changed by the user or when elements are added to or removed from the dialog by creating prototypes or by executing the application.

IUP/LED associates an attribute (`WID`) with each interface element and the value of this attribute is the handle of the interface element in the native system. Thus, an application may query this value and use it as a parameter in calls to the native interface system.

4.1 LED: a language for dialog specification

LED is an expression language to specify dialogs which are compiled at run time by IUP functions, supporting three important aspects of an interface system:

- independence of dialogs from the application code;
- rapid prototyping;
- and customization for different users and platforms.

Rapid prototyping is possible because a LED program is compiled at run time and no specific function application is needed; it is necessary only that the main program calls the IUP function that compiles an LED program. This single program can be used to create prototypes quickly for the interface of any application.

Customizing the application can be done by the user because dialog definitions are available in text form. Since LED is a simple language that is easily understood, the user can modify the dialog specification in order to create simplified versions of the program or to translate the interface into another language.

4.1.1 Model

The LED language supports the definition of an abstract model of a user interface where the dialogs are defined by their abstract layout and the elements that compose the dialogs are mostly specified by their functions and not by their final appearance. In LED, the programmer needs only to provide some parameters associated with the functionality of each interface element; appearance attributes may be specified, but they aren't strictly necessary.

By using an abstract interface model, the application programmer can create dialogs without having to worry about the interface system in which the program will execute. Moreover, porting the user interface to a new environment should be immediate for it is enough to write a driver for the new native interface system. In this way, the program can run without any changes in systems that are as different from each other as Microsoft Windows, OS/2 Presentation Manager, Motif, OpenLook, and Macintosh. The IUP toolkit offers an API that implements the abstract model supported by LED.

The layout model used in LED is a simple, easily understood model that is able to maintain an abstract layout, independent of size and complexity. The relative position of the interface elements that compose a dialog remain unchanged after the size has been changed by the application user or by the addition or removal of elements. The programmer is freed from having to calculate sizes and positions for the interface elements in each dialog.

4.1.2 Syntax

LED is an expression language designed so that dialogs can be defined mostly by specifying their abstract layout and the functionality of the interface objects that compose the dialog. Appearance attributes, such as color and character fonts, are optionally specified as environment variables, similar to the ones that already exist in Unix and DOS. This distinction between mandatory information (related to functionality) and optional information (related to appearance) is explicit in the syntax of LED.

The syntax of expressions in LED is simply:

$$v = f[a](p),$$

where:

- v is the name that should be used by the application in order to access the interface element that is being defined by the expression $f[a](p)$;
- f is the type of interface element that is being described;
- a is the list of attributes for this interface element in the form $v_1 = a_1$, $v_2 = a_2$, ..., where v_i is the name of the attribute and a_i is its value (a string);
- and p is the list of parameters that define the functionality of elements of type f.

Naming an expression is optional. Nevertheless, an application can only communicate directly with elements that have names. Thus, an application cannot change or query the attributes of anonymous elements, even though these elements may be fully active.

4.1.3 Example

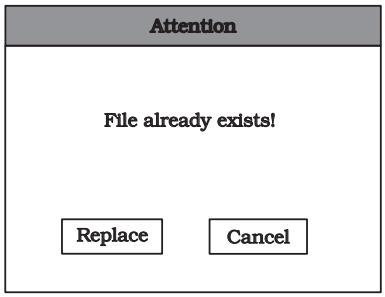

Figure 1: A simple dialog.

As an example of the use of LED, consider the dialog in Figure 1. This dialog is composed of a text string ("File already exists!") and two buttons ("Replace" and "Cancel"). The abstract layout of this dialog can be described in the following form: the buttons are centered at the lower part of the dialog area and the text is centered in the remaining area above the buttons. The specification in LED of the dialog in Figure 1 follows immediately from this description and is shown in Figure 2.

```
confirm = dialog[TITLE="Attention"](body)
body    = vbox(fill(), prompt, fill(), buttons)
prompt  = hbox(fill(), warning, fill())
buttons = hbox(fill(), replace, fill(), cancel, fill())
warning = label("File already exists!")
replace = button("Replace", do_replace)
cancel  = button("Cancel", do_cancel)
```

Figure 2: A simple dialog specification.

In the specification in Figure 1, we used the following interface elements: `dialog`, `vbox`, `hbox`, `label`, `button` and `fill`. We shall describe these and other interface elements in the next section. In the example in Figure 2, all the elements have been given names although this is not necessary particularly for intermediate elements. An alternate specification without the use of intermediate names is given in Figure 3.

```
confirm = dialog[TITLE="Attention"](
          vbox(
               fill(),
               hbox(
                    fill(),
                    label("File already exists!"),
                    fill()),
               fill(),
               hbox(
                    fill(),
                    button("Replace", do_replace),
                    fill(),
                    button("Cancel", do_cancel),
                    fill())))
```

Figure 3: A simple dialog specification without intermediate names.

4.1.4 Elements

The interface elements available in LED can be divided in the following categories:

- group: define a common functionality for a group of elements;
- composition: define a form to exhibit the elements;
- fill: occupy empty spaces dynamically;

- and primitive: interact with the user.

Since the list of parameters that define the functionality of elements may contain other expressions, the elements that compose a dialog are organized in a hierarchical structure. The structure corresponding to the dialog of Figure 2 is shown in Figure 4. It is interesting to notice that the internal nodes of the tree are either composition or group elements while the leaves are either primitive or fill elements. This hierarchical structure permits dialogs to be gradually specified, combining simple, previously tested dialogs into more complex ones.

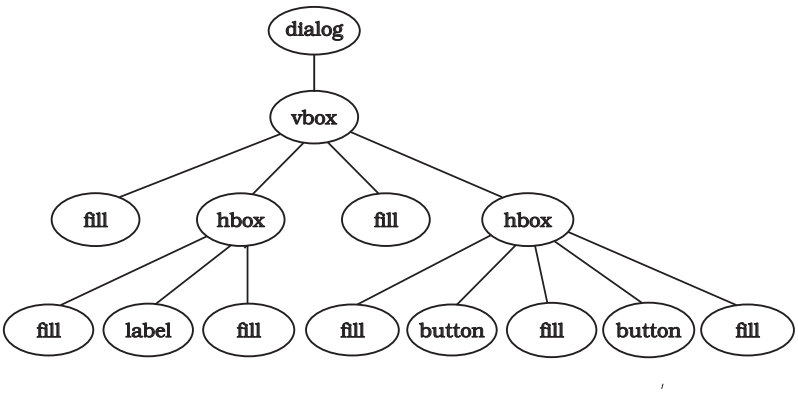

Figure 4: A dialog structure for the simple specification.

Grouping elements The grouping elements define a common functionality for a collection of elements. The types of grouping elements available in LED are:

- `dialog`: compose an interaction dialog with the user;
- `radio`: restrict the *on* state to exactly one of a set of toggles;
- and `menu`: group items and submenus.

Composition elements The composition elements determine whether a collection of interface elements are presented vertically or horizontally. Following the TEX paradigm, we have two composition elements:

- `hbox`: present group of elements horizontally;
- and `vbox`: present group of elements vertically.

With these two composition elements, it is possible to build many dialogs without defining explicitly the coordinates of each element that composes the dialog. The specification in Figure 2 for the example illustrates the use of `vbox` for vertically arranging the two main items (`prompt` and `buttons`), and the use of `hbox` for a horizontal arrangement of the two buttons.

Fill elements `fill` is the only filling element; it occupies the empty spaces in a dialog proportionally and dynamically. `fill` is responsible both for maintaining the abstract layout when the dialog is resized and for relative positioning of the interface elements in the composition elements (`hbox` and `vbox`). The specification in LED for the example in Figure 2 illustrates the use of `fill` to center the label (`prompt`) horizontally and vertically in the dialog area. If the specification of `body` were:

```
body = vbox(prompt, fill(), buttons)
```

then the text would appear at the top of the dialog area.

Primitive elements The primitive elements available in LED are:

- `button`;
- `canvas`: working area;
- `frame`: creates a border around an interface element;
- `hotkeys`: function keys;
- `image`: static image;
- `item`: menu item;
- `label`: static text;
- `list`: string list with scrollbar;
- `submenu`: menu within a menu;
- `text`: captures a text fragment of two or more lines;
- `toggle`: two-state button (*on*/*off*);
- and `valuator`: captures a numeric value.

With the exception of `canvas`, all other primitive elements are well defined and have the same behavior in all interface systems. The interface element `canvas` is a very peculiar element for it is the main link between the graphical part of an application and the interface system. It is through `canvas`es that the application objects are exhibited and manipulated by the user. This intimate connection to the application makes it hard to give an abstract definition for the behavior of `canvas`. InterViews gives a formal definition for this behavior, removing from the application the treatment of some events such as repaint and resize. (An alternative abstraction was given by Neelamkavil-Mullamey [NM90].) In IUP/LED, the behavior of `canvas` is simple: all events that happen on the `canvas` are passed on to the application, which is responsible for them. It is important to note that, among primitive elements, `canvas` is the only one which competes with `fill` for empty spaces.

4.1.5 Attributes

Attributes for interface elements are implemented as environment variables represented by the expression $[v = a]$, where v is the name of a variable and a is its value (a string). These environment variables provide an inheritance mechanism for the attributes: the variables defined for an element are automatically exported to its children. For example, a variable defined for an `hbox` is also defined, with the same value, for all elements that are in this `hbox`. If one of these elements defines a variable with the same name, the associated value of the element has priority over the value defined for the `hbox`. This mechanism allows global attribute assignment, with the possibility of local changes. For example, to change the character font globally for the `confirm` dialog, but locally in the `replace` button, we could write:

```
confirm = dialog[FONT="Helvetica"](...)
replace = button[FONT="HelveticaBold"](...)
```

Some variable names are recognized by the system and represent the attributes of interface elements. The majority of these attributes control appearance such as color, character fonts, and cursor style. Some attributes, such as the `HOTKEYS` attribute, define functionality; in this case, function keys are associated with dialogs.

Names not recognized by the system can be used by the application for any purpose. This feature provides a general-purpose, extensible attribute table which may be used by the application; interface objects can maintain their own "state". This mechanism also allows platform dependent attributes to be specified and interpreted by the corresponding driver, without any consequences for other platforms, thus making it possible to fine-tune the interface for each platform and yet maintain a single LED specification.

4.2 IUP: a toolkit based on LED

IUP is a toolkit with approximately 40 functions for building and manipulating dialogs for applications. This toolkit is basically an API for implementing LED and contains functions for:

- converting LED specifications to native interface system objects;
- creating interface elements without using LED;
- registering the application functions corresponding to the actions used in LED;
- associating names with interface elements;
- exhibiting and hiding dialogs;
- and querying and setting attributes for interface elements.

IUP is written in ANSI C and can be ported to many different environments such as Microsoft Windows, OpenLook via XView, Motif, and DOS. For DOS, we have written a complete and portable interface system having an appearance similar to Motif. The control flow in an application that uses IUP with LED is analogous to the ones that use other toolkits and can be summarized with the following statements:

- initialize IUP, by calling the function `IupOpen`;
- create dialogs by compiling an LED specification with the function `IupLoad`, or by calling IUP functions to create each interface element;
- register the functions corresponding to LED actions with the function `IupSetFunction` (in the example in Figure 2, `do_replace` and `do_cancel` are actions and not application functions);
- exhibit the initial dialogs with the function `IupShow`;
- and yield control to IUP with the function `IupMainLoop`, which waits for user's actions and calls the corresponding application functions.

4.2.1 Integrating IUP with LED

In LED, we can associate names with interface elements, but these names cannot be directly used in the IUP functions that create and manipulate the interface elements; IUP functions expect a handle. Therefore, to refer to an interface element created in LED, an application has to call the function `IupGetHandle`. As an example, we present the IUP code in Figure 5 that exhibits the dialog in Figure 1 specified in LED in Figure 2. Even though we are not checking for errors, all IUP functions return a code that indicates success or failure in the execution of the function.

```
/* load and compile a LED specification */
       IupLoad("attention.led");

/* register application functions */
       IupSetFunction("do_ok", (Icallback) f_ok);
       IupSetFunction("do_cancel", (Icallback) f_cancel);

/* exhibit dialog */
       IupShow(IupGetHandle("confirm"));
```

Figure 5: Code to exhibit dialog.

4.2.2 Creating interface elements in IUP

Interface elements can be created dynamically by writing programs in IUP. However, the creation of interface elements in IUP differs from the creation of an element with LED in two specific ways. The first difference is that in IUP names are not associated with interface elements at creation time, as they are in LED. When we create an element with IUP, the corresponding function returns a handle that must then be passed to the function `IupSetHandle` to associate a name with the interface element. The second difference is that in LED the definition of the attributes happens at creation time, while in IUP the element has to be created before its attributes can be defined. As an example, we present the IUP description in Figure 6 of the dialog defined in LED in Figure 2.

```
/* interface element identifier declaration */
       Ihandle *cancel, *replace, *warning, *buttons, *prompt, *body, *confirm;

/* create interface elements */
       cancel  = IupButton("Cancel", "do_cancel");
       replace = IupButton("Replace", "do_replace");
       warning = IupLabel("File already exists!");
       buttons = IupHbox(IupFill(), replace, IupFill(), cancel, IupFill(), NULL);
       prompt  = IupHbox(IupFill(), warning, IupFill(), NULL);
       body    = IupVbox(IupFill(), prompt, IupFill(), buttons, NULL);
       confirm = IupDialog(body);

/* define a title for the dialog */
       IupSetAttribute(confirm, "TITLE", "Attention");

/* register application functions */
       IupSetFunction("do_ok", (Icallback) f_ok);
       IupSetFunction("do_cancel", (Icallback) f_cancel);

/* exhibit dialog */
       IupShow(confirm);
```

Figure 6: Code to exhibit dialog using IUP.

In the code in Figure 6, names for the interface elements are not defined, since we refer to the elements by the handles returned by the functions that create elements. In order to associate names with interface elements, we call the function `IupSetHandle` after creating each element. This procedure is shown in the code in Figure 7.

```
/* create interface element and associate the name "cancel" */
       cancel  = IupButton("Cancel", "do_cancel");
       IupSetHandle("cancel", cancel);
```

Figure 7: Associating names with interface elements.

As in LED, it is not necessary to keep the handles of intermediate elements; the dialog in Figure 1 can be specified as shown in Figure 8.

4.3 Implementation

The main routine in the implementation of IUP/LED is the one that converts the abstract layout model to a concrete model. The algorithm has three phases. The first phase computes the smallest

```
confirm = IupDialog(
              IupVbox(
                 IupFill(),
                 IupHbox(
                    IupFill(),
                    IupLabel("File already exists!"),
                    IupFill(),
                    NULL),
                 IupFill(),
                 IupHbox(
                    IupFill(),
                    IupButton("Replace", "do_replace"),
                    IupFill(),
                    IupButton("Cancel","do_cancel"),
                    IupFill(),
                    NULL),
                 NULL));
```

Figure 8: Specifying interface elements in IUP without intermediate names.

size that holds the interface element. This size is called the natural size of each element and is defined in Figure 9. The second phase computes the element's present size or the size with which the element will actually be exhibited to the user. The third and last phase computes the position of each interface element. Implementing this algorithm requires the answer to the following questions related to the appearance of the user interface.

- What is the policy for distribution of empty spaces among `fill`s and `canvas`es?
- Does a `fill`/`canvas` which is deep in the hierarchy receive less space than a shallow `fill`/`canvas`?
- When a user explicitly defines the size of a element (specifying the `SIZE` attribute), what size is being referenced, the natural or the concrete one?
- In what unit does the user define the size of the interface element?

Prototypes were extensively studied to obtain answers to these policy questions and their combinations. For the first question, the best answer is to give priority to `canvas` over `fill` in empty space distribution. This decision was justified based on the functionality of the two elements: `fill` is for justifying elements, while `canvas` is the space used by the application and the user for communication using the application objects. Therefore, it seems reasonable that increasing the size of a dialog should increase the size of the `canvas`.

The distribution of empty space for elements in the same hierarchical level is proportional: they all receive the same amount of empty space. The key problem is the distribution of empty space for elements at different levels. If all elements receive the same amount of space, irrespective of

Element	Natural Size
`dialog`	the size of the element it contains
`radio`	it has no size for it only defines functionality
`menu`	minimum to hold all its elements
`hbox`	height equal to the height of its highest element; width equal to the sum its elements' width
`vbox`	height equal to the sum its elements height; width equal to the width of its widest element
`button`	somewhat larger than its text or image
`canvas`	the size of a character
`frame`	somewhat larger than the size of the element it contains
`hotkeys`	it has no size for it only defines functionality
`image`	the size of its image
`item`	somewhat larger than its text or image
`label`	the size of its text or image
`list`	dependent on the native system
`submenu`	somewhat larger than its text
`text`	somewhat larger than its initial text
`toggle`	minimum to hold its text or image and a feedback box
`valuator`	dependent on the native system

Figure 9: Natural size of elements.

their level, it would be impossible to divide a dialog in regions as illustrated in Figure 10, because all elements would receive the same amount of space and the dialog illustrated in Figure 10 would appear as illustrated in Figure 11 (a or b, depending on how the dialog was defined: a vertical box containing two horizontal boxes; or a horizontal box containing two vertical boxes). The algorithm we implemented divides a dialog into regions, equally distributing empty spaces of a box among its extensible elements, which then equally divide their empty space among their own elements, and so on. Thus, the outer elements get more space then the inner ones, in an exponential proportion.

In order to answer the third question, we decided that, when a user chooses a size, the user wants the interface element to be shown in that specified size and not in one calculated by the IUP. Therefore, it becomes clear that the size specified by the user refers to the current size. Thus, the interface element's size must not be recomputed after the dialog is resized. The only exception is the dialog element that has its size recalculated as part of the application.

About the last question, it is known that raster units or pixels should not be used as a unit of size because of an obvious dependency on device resolution in interface element specification. Therefore, we have accepted that the values related with sizes should be proportionally specified to a fraction of the average size of a character from the character fonts used by the element in question. The width unit represents 1/4 of the medium width of a character and the height represents 1/8 of the height of a character (these values are also used by MS-Windows). This kind of unit guarantees that the interface elements will show the users the same information, irrespective of the output

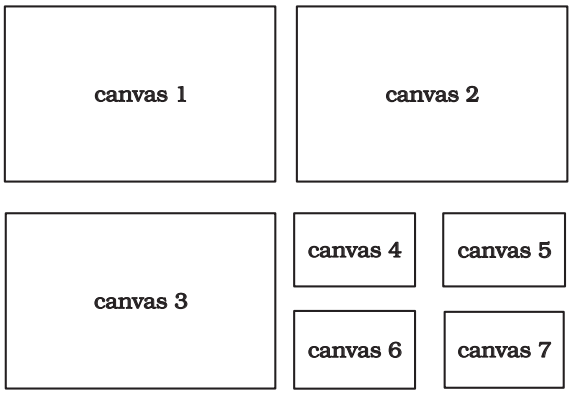

Figure 10: Seven canvases dividing a dialog in symmetrical areas.

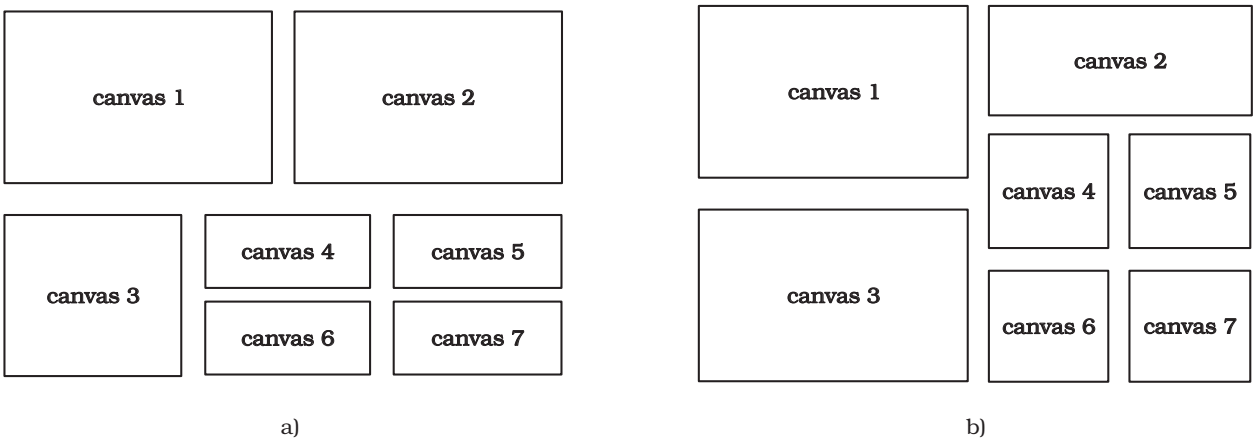

Figure 11: Seven canvases dividing a dialog in asymmetrical areas.

device.

We now present three recursive algorithms that have been implemented to convert abstract layout to concrete layout. The first algorithm calculates an element's natural size (Figure 12), the second its current size (Figures 13 and 14), and the last each interface element's size and position (Figure 15).

4.3.1 Algorithm for computing concrete layout

The algorithm for computing the natural size of interface elements is shown in Figure 12. The algorithm only computes the sizes of the composition elements (`hbox` and `vbox`) and group elements (`dialog`); the sizes of the primitive elements, such as `label`, `button` and `text`, are obtained by querying the driver for the native interface system.

The algorithm in Figure 12 receives as input a node of the hierarchical structure of a dialog, representing an interface element and computes its natural size. The return value indicates the directions (vertical or horizontal) in which the element can grow and with which priorities (high or

low). For example, an `hbox` that contains a `fill` can grow horizontally with a low priority, and an `hbox` that contains a `canvas` can grow in both directions with high priority. This information will be useful in order to calculate the current size and to determine the empty space distribution.

The algorithm in Figure 12 prepares the interface elements for the computation of their current size. In Figures 13 and 14, we present an algorithm that distributes the empty spaces and calculates the current sizes of the interface elements. This algorithm, which has no return value, receives as input a node of the hierarchical structure of a dialog, representing a grouping element, and provides the width and height of a rectangle where the interface elements will be composed.

To complete the conversion from abstract layout to concrete layout, the algorithm in Figure 15 positions the interface elements. The algorithm doesn't have any return value and has as parameters: a hierarchical structure knot; and a pair of coordinates (x, y) to indicate the upper left corner position where the element should be placed.

The complexity of the complete conversion from abstract layout to concrete one is computed by adding the complexities of the three algorithms presented in Figures 12 through 15. The three algorithms traverse the tree structure of a dialog exactly once; thus, they are linear in the number of interface elements contained in a dialog. Therefore, the conversion to a concrete layout is also linear. This complexity is adequate for real time recalculation of a concrete layout. Nevertheless, the recalculation only occurs when the user has finished resizing the dialog, in order to avoid the "blinking" effect that could occur if all elements had to be erased and redrawn for each mouse movement.

5 Experience with IUP/LED

TeCGraf is a research and development laboratory at the Pontifical Catholic University in Rio de Janeiro (PUC-Rio) with many industrial partners. Several substantial projects at TeCGraf have used IUP/LED. In addition, graduate students in Computer Graphics and in user interface courses at PUC-Rio have used this system. In this section, we will examine the use of IUP/LED in TeCGraf's PETROX project; this project requires the definition of almost 50 dialogs, so we believe it provides a good evaluation of the abstract layout model. We will also examine the use of IUP/LED in the Computer Graphics course where students have had to create an interactive graphic editor, a program that requires a substantial amount of interaction between IUP/LED and the graphics system.

5.1 PETROX project

The PETROX project required the creation of a multi-platform interactive program for editing chemical process diagrams that provide entry information for a simulator. The fifty dialogs that capture the numeric information associated with the elements of the diagram were specified using LED by a chemical engineer who had no knowledge of user interface concepts at the beginning of the project. After one month of training, she was able to understand user interface concepts and learn both the IUP toolkit and the LED language. Since the dialogs have common interface elements, the engineer separated these files and created a library. Thus, every time a dialog needed a common element, it was copied from the library and reused in the dialog's specification. As the

dialogs were built, requirements for two new interface elements that were not in IUP/LED were identified. These elements are:

- the drop down list, that contains the primitive element list, but with a different presentation. Instead of exhibiting a complete list, only one element in the list is shown beside a button with an arrow pointing downwards. When selected, this button gives the option list, allowing a new option to be selected;
- the vector, that permits values to be attributed to a vector without having to define a text type element for each vector position. The number of fills used by the interface element alignment in dialogs of the PETROX project is quite large. Reducing this number involved creating the `ALIGNMENT` attribute for the `hbox` and `vbox` composition elements. In `hbox`, the possible values are: `IUP_TOP`, `IUP_CENTER` and `IUP_BOTTOM`; while in `vbox`, they are: `IUP_LEFT`, `IUP_CENTER` and `IUP_RIGHT`.

5.2 Graphic editor

The Computer Graphics students had to create an interactive graphic editor by using IUP/LED as an interface system and a local implementation of GKS as a graphic system. They started using IUP/LED after only one lesson where an abstract layout model was described and functions were provided. A quick reference to IUP/LED functions and interface element attributes was made available as support for this activity. The students were easily able to specify the dialogs and use the toolkit. Nevertheless, some students found difficulties in using an asynchronous rather than a sequential programming model. This problem is not attributable to IUP/LED but instead to a lack of experience with the user interaction model.

The graphics system was used passively and all input was through IUP. The only detected problem while using this model is that IUP coordinates are in raster units at the upper right corner of the canvas and the positions needed by GKS are in world coordinates defined by the programmer (relative to the lower left corner of the workstation). A solution was provided by a transformation routine between both systems and its incorporation in the IUP interface for GKS.

6 Comparative Analysis

There are other interface systems, such as IntGraf [TeC91], CIRL/PIWI [CDEP92a, CDEP92b], XVT [Roc89] and OI Toolkit [Dat91], that offer solutions to user interface portability problems. In this section, we make a comparison of IUP/LED with CIRL/PIWI, since in earlier papers [CDEP92a, CDEP92b] CIRL/PIWI was compared with these other toolkits.

6.1 CIRL/PIWI

CIRL/PIWI [CDEP92a, CDEP92b] is a portable user interface toolkit developed by the University of Waterloo. This toolkit uses a user interface abstraction to support a portability strategy. This abstraction has two components: CIRL, a language that uses tags to specify interface elements; and PIWI, a virtual toolkit similar to IUP. The structure and appearance of interface elements are specified separately, and a knowledge base contains information about the look-and-feel of a

specific native system. The two descriptions and the knowledge base are provided as input to a compiler which produces a description of interface elements in the tagging language for the native system. PIWI is a virtual toolkit similar to IUP. PIWI is available for the Macintosh, X11/Motif, Microsoft Windows, and Presentation Manager in OS/2. Besides the user interface functions, graphics functions for drawings are also available. The interface elements communicate with the application through events, where each dialog has an accompanying event routine.

CIRL/PIWI and IUP/LED share the following common features:

- the implementation of a tagging language;
- the layouts are defined without having to define their coordinates;
- the implementation of a virtual toolkit;
- and their implementation inherits the native look-and-feel.

Nevertheless, there are several differences that separate the two solutions including:

- the CIRL compiler uses a knowledge base containing information about the look-and-feel of different interface systems. This knowledge base allows the compiler to make decisions related to the layout that weren't defined by the tagging or appearance model. On the other hand, LED defines a very simple model of layout definitions based on the boxes-and-glue paradigm of the TEX processor;
- the CIRL compiler produces a specification in the tagging language of the native interface system that then needs to be connected to the application. The LED specification can be compiled at the same time as the application, potentially decreasing the time required to produce a prototype;
- the interface elements can only be created through CIRL, while in IUP/LED the interface elements can be defined by using LED or by using IUP services;
- IUP/LED provides an attribute mechanism that allows the appearance of interface elements to be defined and fine-tuned for specific interface systems. In CIRL, the appearance of interface elements are defined in an optional file, the fine-tuning is done based on the tagging created by the CIRL compiler, and it isn't possible to associate information with the interface elements;
- in CIRL/PIWI, the communication between interface elements and the application is based on the event model, while in IUP/LED, it is based on the callback model;
- IUP/LED supports a fixed look-and-feel in addition to a native look-and-feel;
- CIRL/PIWI supports better fine-tuning of the user interface than IUP/LED. Since CIRL/PIWI creates a description of interface elements in the tagging language of the native interface system, a final adjustment can be made at that level. IUP/LED only allows fine-tuning with the characteristics implanted in the IUP/LED driver for the native interface system. IUP allows a native identifier of interface elements to be checked through the `WID` attribute, allowing dynamic fine-tuning through the functions of the native interface system.

7 Contributions

IUP/LED has made a number of contributions to technology supporting the development of portable user interface toolkits:

- IUP/LED defines an abstract layout model that allows dialogs to be created in a natural form, without having to compute the position of the interface elements;
- the model used by IUP/LED can be produced for many different interface systems;
- LED describes a dialog using its functions; the appearance attributes are optional;
- LED allows applications to by customized by the users themselves for different users and platforms;
- LED offers an attribute mechanism that allows specific adaptations to an interface system, and makes it possible to add application information directly to the interface elements;
- LED is an expression language with a simple syntax that can be learned quickly;
- IUP is a virtual toolkit that allows portable interactive programs to be built without forcing the programmer to be knowledgeable about the native interface system;
- IUP allows programs to have both the system's native look-and-feel, which helps the user of only one environment, and the system's fixed look-and-feel, which helps the user of only one application who needs to run the program on different machines;
- IUP has only forty functions, a very small group when compared to the hundreds of functions, of MS-Windows, the Macintosh Toolbox, and Motif. This feature makes it easy to learn the IUP functions quickly.

8 Conclusions

In this paper, we have described a portable user interface toolkit called IUP/LED. LED is a tagging language for specifying dialogs, and IUP is set of functions for creating dialogs and to connect dialogs specified in LED to the native system. This toolkit allows interactive applications to be moved easily to different computer environments with minimal effort.

We have described the layout problem for interface dialogs and have indicated that it is difficult to define the layouts by using explicit geometric positions in a concrete layout model. As a solution to this problem, IUP/LED defines an abstract layout model based on the boxes-and-glue paradigm of the TEX text processor. The main advantages of this model are:

- it frees the programmer from calculating interface element sizes and positions;
- and it maintains an abstract layout after any change in the size made by the application user or by the addition or removal of elements.

This model does not work with dialogs with geometrically irregular layouts; however, this type of dialog is rarely used. In addition, the interface designer can always find a geometrically regular layout that is able to expose the required information adequately.

The dialog tagging language, LED, implements an abstract layout model by an expression language with a simple syntax that allows it to be learned quickly even by domain experts with limited computer experience. LED is a powerful language that allows:

- interface elements to be defined without defining appearance attributes;
- dialog definition to be separated from the application;
- and customization for different users and platforms.

The LED language served as the basis for developing the IUP virtual toolkit that allows the application to inherit or ignore the look-and-feel of the native interface system. It is through the IUP toolkit that the application controls the dynamics of the interface elements defined in LED. The basic services IUP offers are:

- convert the tagging in LED to native interface system objects;
- create interface elements dynamically without using LED;
- bind functions from the application to the actions used in LED;
- associate names with elements;
- exhibit and hide the dialogs;
- and set and query attributes for the interface elements.

IUP is a very small toolkit with only forty functions that are easily learned, when compared to the hundreds of functions defined in MS-Windows, Macintosh Toolbox, and Motif. This small number of functions was motivated by the model used by XView. Many interface element controls are not functions but can be specified as element attributes. In this approach, all element manipulation is through two functions: one to query and the other to set a value for each attribute. The advantage of this is that it is easy to extend IUP; the disadvantage is that the programmer who only needs to learn the API of the IUP toolkit will know very little about the interface elements, since it is also necessary to know the attributes of each element.

The main difficulty in developing IUP/LED was in building the algorithm to transform the abstract layout into a concrete one.

The IUP/LED environment can be improved by:

- developing an interactive dialog editor for IUP/LED [Pra94];
- allowing reference to interface elements by name rather than through their handle;
- implementing other interface elements such as the vector that was motivated by the PETROX project;

- and implementing help and clipboard mechanisms.

An important and interesting theme refers to the Multiple Document Interface (MDI) concept introduced in MS-Windows. This concept standardizes the use and programming of the applications that allow users to work with many documents (files) simultaneously. There are similar concepts in OS/2 Presentation Manager and the Macintosh. An analysis of the real benefit of the MDI and of how this concept may be implemented in other interface systems is a question that should be considered.

Another concept largely used in interface systems is the dynamic exchange of data between applications using approaches such as Dynamic Data Exchange (DDE) in MS-Windows. Even though this concept doesn't directly deal with the user interface, it enables the integration of data between applications.

References

[ABB89] G. Avrahami, K. P. Brooks, and M. H. Brown. A Two-view Approach to Constructing User Interfaces. *Computer Graphics*, 23:137–146, 1989.

[App85] Apple Computer, Inc. *Inside Macintosh*, volume 1. Addison-Wesley, Massachusetts, USA, 1985.

[Bla88] G Blackham. Building Software for Portability. *Dr. Dobb's Journal*, 146:18–27, December 1988.

[CDEP92a] D. D. Cowan, C. M. Durance, Giguere E., and G. M. Pianosi. CIRL/PIWI: A GUI Toolkit Supporting Retargetability. Technical Report CS-92-28, University of Waterloo, Dept. of Computer Science, Waterloo, Ontario, 1992.

[CDEP92b] D. D. Cowan, C. M. Durance, Giguere E., and G. M. Pianosi. CIRL/PIWI: A GUI Toolkit Supporting Retargetability. *Software Practice & Experience*, 23(5):511–527, 1992.

[CW84] D. D. Cowan and T. A. Wilkinson. Portable Software: An Overview. In *Proceedings of the1984 Canadian Conference on Industrial Computer Systems*, pages 68–1–68–7, Ottawa, May 1984.

[Dat91] Neuron Data. Open Interface Toolbox. 156 University Avenue, Palo Alto, CA 94301, 1991.

[Dur90] C. M. Durance. An Approach to Application Software Mobility Across User Interface Toolkits. Master's thesis, Faculty of Mathematics, University of Waterloo, Waterloo, Ontario, Canada, 1990.

[Fou91] Open Software Foundation. *OSF/MOTIF Programmer's Guide, Revision 1.1*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Fre88] Donnalyn Frey. Unix vs. Unix. *Dr. Dobb's Journal*, 146:28–35, December 1988.

[FSGC92] L. H. Figueiredo, C. S. Souza, M. Gattass, and L. C. G. Coelho. Geração de Interfaces para Captura de Dados sobre Desenhos. *Anais do V SIBGRAPI*, pages 169–175, 1992.

[Gre85] M. Green. The University of Alberta User Interface Management System. In *Proceedings of SIGGRAPH'85,12th Annual Conference*, pages 205–213, New York, July 1985. ACM. San Francisco, California, July 22-26.

[Hel90] D. Heller. *XView Programming Manual*, volume 7. O'Reilly & Associates, Inc., Sebastopol, California, 2 edition, July 1990.

[HH89] H. R. Hartson and D. Hix. Human-Computer Interface Development: Concepts and Systems for its Management. *ACM Computing Surveys*, 21(1):9–93, March 1989.

[HHE84] H. R. Hartson, D. Hix, and R. W. Ehrich. A Human-computer Dialogue Management System. In *Proceedings of INTERACT'84, First IFIP Conference on Human-Computer Interaction*, pages 57–61, London, September 1984. International Federation for Information Processing.

[Hix90] D. Hix. Generation of User Interface Management Systems. *IEEE Software*, pages 77–87, September 1990.

[IEE88] IEEE. *Standard 1003.1-1988 (Posix)*. IEEE, 1988.

[Inc90] Sun Microsystems Inc. *Open Look Intrinsics Toolkit Widget Set Programmers's Guide, Revision A*, June 11 1990.

[Knu84] D. E. Knuth. *The TEXbook*. Addison-Wesley, 1984.

[KR88] B. Kernighan and D. Ritchie. *The C Programming Language*. Addison-Wesley, Reading, Massachusetts, USA, 2 edition, 1988.

[LCCC90] C. J. P. Lucena, D. D. Cowan, I. M. Campos, and R. H. B. Cabral. Interface as Specifications in the MIDAS User Interface Development System. *ACM SIGSOFT*, 15(2):55–72, April 1990.

[LVC89] M. A. Linton, J. M. Vlissides, and P. R. Calder. Composing User Interfaces with InterViews. *IEEE Computer*, pages 8–22, 1989.

[Mic92] Microsoft. *Visual Basic Programming System for Windows version 2.0, Programmer's Guide*. Microsoft Corporation, 1992.

[MvD91] A. Marcus and A. van Dam. User Interface Developments for the Nineties. *IEEE Computer*, 24(9):49–57, September 1991.

[NM90] F. Neelamkavil and O. Mullarney. Separating Graphics from Applications in the Design of User Interfaces. *The Computer Journal*, pages 437–443, 1990.

[Pet90] C. Petzold. *Programming Windows: the Microsoft Guide to Writing Applications for Windows 3*. Microsoft Press, Redmond, Washington, 1990.

[Pra94] R. O. Prates. Visual LED: uma ferramenta interativa para geração de interfaces gráficas. Master's thesis, Departamento de Informática, Pontifícia Universidade Católica, Rio de Janeiro, 1994.

[Roc89] M. J. Rochkind. XVT: A Virtual Toolkit for Portability between Window Systems. *USENIX*, pages 151–163, Winter 1989.

[TeC91] TeCGraf. *Manual de Referência do IntGraf: Sistema de Interface Gráfica com Usuário.* Pontifícia Universidade Católica, Rio de Janeiro, 1991.

```
function Nsize(n): integer
    case type(n)
        dialog:
            compute Nsize(child(n))
            expansion(n) ← both directions
            Nwidth(n) ← Nwidth(child(c))
            Nheight(n) ← Nheight(child(c))
        hbox:
            expansion(n) ← no direction
            Nwidth(n) ← 0
            Nheight(n) ← 0
            for each child c of n do
                expansion(n) ← expansion(n) | Nsize(c)
                Nwidth(n) ← Nwidth+Nwidth(c)
                Nheight(n) ← max(Nheight(n),Nheight(c))
        vbox:
            expansion(n) ← no direction
            Nwidth(n) ← 0
            Nheight(n) ← 0
            for each child c of n
                do expansion(n) ← expansion(n) | Nsize(c)
                Nwidth(n) ← max(Nwidth(n),Nwidth(c))
            Nheight(n) ← Nheight+Nheight(c);
        button, label, text:
            get natural size from driver
            expansion(n) ← no direction
        canvas:
            get natural size from driver
            expansion(n) ← both directions with high priority
        fill:
            Nwidth(n) ← 0
            Nheight(n) ← 0
            if n is inside an hbox then
                expansion(n) ← horizontal with low priority
            else
                expansion(n) ← vertical with low priority
    return expansion(n)
```

Figure 12: Algorithm for calculating an element's natural size.

```
function Csize(n,w,h)
    if n can expand horizontally then
      Cwidth(n) ← max(Nwidth(n),w)
    else
      Cwidth(n) ← Nwidth(n)
    if n can expand vertically then
      Cheight(n) ← max(Nheight(n),h)
    else
      Cheight(n) ← Nheight(n)
```

Figure 13: Algorithm for calculating an element's current size.

```
function Type(n,w,h)
    case type(n)
        dialog:
            Csize(child(n),Cwidth(n),Cheight(n))
        hbox:
            if n expands horizontally with high priority then
              m ← number of children on n that expand horizontally with high priority
              spaces ← Cwidth(n)-Nwidth(n)/m
              priority ← high
            else
              m ← number of children on n that expand horizontally
              spaces ← Cwidth(n)-Nwidth(n)/m
              priority ← low
            for each child c of n do
              if c expands horizontally with high priority and priority is high then
                w ← spaces
              else
                if c expands horizontally with low priority and priority is low then
                  w ← spaces
                else
                  w ← 0
              Csize(c,Cwidth(c)+w,Cheight(n))
        vbox:
            if n expands vertically with high priority then
              m ← number of children on n that expand vertically with high priority
              spaces ← Cheight(n)-Nheight(n)/m
              priority ← high
            else
              m ← number of children on n that expand vertically
              spaces ← Cheight(n)-Nheight(n)/m
              priority ← low
            for each child c of n do
              if c expands vertically with high priority and priority is high then
                h ← spaces
              else
                if c expands vertically with low priority and priority is low then
                  h ← spaces
                else
                  h ← 0
              Csize(c,Cwidth(c),Cheight(n)+h)
```

Figure 14: The function to determine type.

```
function position(n,x,y)
    x(n) ← x
    y(n) ← y
    case type(n)
        dialog:
            position(child(n),x,y)
        hbox:
            foreach child c of n do
                position(c,x,y)
                x ← x + Cwidth(c)
        vbox:
            for each child c of n do
                position(c,x,y)
                y ← y + Cheight(c)
```

Figure 15: Algorithm to compute each element's size and position.