SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 0(0), 1–27 (? 1995)

IUP/LED: A Portable User Interface Development Tool

C. H. Levy, L. H. de Figueiredo, M. Gattass, C. J. P. Lucena
Departamento de Informática, PUC-Rio
Rua Marquês de São Vicente 225, 22453-900 Rio de Janeiro, RJ, Brazil
levy,lhf,gattass,lucena@icad.puc-rio.br

AND

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

SUMMARY

Minimizing the amount of code that must be written and maintained is particularly critical in the development of the user interface for a highly interactive system, since the code for the user interface represents a substantial part of the application. This is especially important where the interactive system is available on a number of distinct platforms. Providing a single user interface abstraction requiring only one set of source code that can be mapped automatically into specific interface systems appears to be the preferred approach; but the underlying model must be designed carefully in order to keep the system relatively simple, easy to use and maintain, and allow ease of experimentation as user interfaces are produced. We describe the design and implementation of IUP/LED, a portable user interface toolkit that we believe has these properties. The toolkit is designed for rapid prototyping and modification, to provide a look-and-feel appropriate to a specific computing environment, is easily expanded to support new interface developments, and supports an abstract layout model. We also present a summary of the experiences in using the toolkit to indicate that it does support the original design objectives.

KEY WORDS Software User Interfaces User Interface Toolkits User Interface Management Systems Retargetability User interface resources

INTRODUCTION

User Interface Toolkits (UITs) and User Interface Management Systems (UIMSs) are two major classes of software for constructing graphical user interfaces.[1] A UIT is a library of interface objects that implement different interaction techniques with the user; tools in this class are available as functions that are called by the application to create and control the dialog with the user. Some UITs are: XView,[2] OSF/Motif,[3] OLIT,[4] SDK for MS-Windows,[5] and the Macintosh Toolbox.[6] UITs frequently provide 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

CCC 0038–0644/95/000001–27

Received ?
Revised 25 July 1995

direct manipulation of interface objects. However, management of the interaction with the user must be programmed as part of the application, even when user interface development is supported by a toolkit.

In contrast, a UIMS is a group of integrated high level interactive programs used to design, create prototypes, execute, evaluate and maintain user interfaces.[1] Two such UIMSs are the University of Alberta UIMS[7] and DMS.[8] In a 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.[9] UIMSs assume that application development is a joint undertaking of two experts: one expert in the application domain and another in user interfaces. The first expert solves the computer application problem, while the second expert works with the related psychological, cognitive, ergometric and linguistic human factors to design an appropriate interaction between the user and the application.[1]

Many commercial interface systems, such as Visual Basic,[10] 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.[12] proposed a tool for the automatic generation of interfaces for data entry to programs for engineering simulation and optimization. Although this tool is not an integrated application, it can be considered as a UIMS because it contains all aspects of user interface design.

Recently, a new generation of UIMSs has appeared: User Interface Development Systems (UIDSs).[13] UIDSs use knowledge bases on interface design techniques and software design principles to help the interface specification process.[14] In contrast with UIMSs, which need interface experts, the integration provided by UIDSs allows users themselves to be the interface experts.

Other approaches to user interface development are model-based UIMSs and Programming-by-Demonstration (PBD, also know as Programming-by-Example, PBE). One model-based UIMS is the User Interface Design Environment (UIDE).[15] In this system, user interfaces are specified at a high level of abstraction that includes an abstract data model, with typed attributes and operations on them, and pre- and post-conditions for these operations, specifying when the operations are valid and how they change attributes. The high level specifications in UIDE are written in an extended form of Pascal that are then compiled into running code. Programming-by-Demonstration was pioneered in Peridot.[16] In such systems, user interfaces are built by drawing the layout and performing actions as the final user would; The overall behavior of the interfaces created by demonstration is obtained by inference from such examples.

According to Myers,[17] "the challenges for future tool creators seem to be to provide tools which are easier to learn and which significantly increase the efficiency of the user interface designers." The three types of tools already described support the construction of programs with user interfaces, but they do not attack many important aspects of the problem which would contribute to ease of use and designer efficiency. These two important characteristics can be affected by several factors including the capability to:[18]

* In this paper, *dialog* is mainly used in the following technical sense: a dialog is "any interactive exchange of information that takes place in a limited spatial context."[11] In window systems, dialogs are top level windows containing primitive interface elements, such as buttons or lists.

- develop applications on multiple platforms from the same specification;
- specify the user interfaces in a platform-independent fashion yet with the look-and-feel of a specific platform (native look-and-feel);
- use the same development system on multiple computer platforms;
- rapidly prototype the user interface before the application code is written;
- rapidly prototype the user interface without impacting other parts of the application;
- easily incorporate modifications including those discovered through demonstration and user testing;
- produce multiple user interfaces for the same application;
- minimize the amount the user of the tools must learn before becoming productive;
- minimize the number of different types of expertise required to design and prototype an interface;
- and be expandable to take advantage of any new user interaction modes.

Certainly the construction of a portable UIT supporting an abstract layout model and allowing some degree of run-time binding can form a solid base for the construction of more complex user interface tools that encompass these factors. The CIRL/PIWI[19] toolkit provided a partial solution by supporting the development of applications for more than one platform and using an abstract layout model for platform-independent implementation. The goal of this paper is to describe the design and implementation of a portable user interface toolkit developed by the Pontifical Catholic University in Rio de Janeiro (PUC-Rio) and named IUP/LED which considers all these previously mentioned factors in its design.

SOFTWARE PORTABILITY

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 the various flavors of Unix, have subtle differences that can hamper portability if developers are not careful;[20]
- compiler differences, such as the default size of an integer which may be important for programs that handle binary files. In addition, each compiler provides proprietary function libraries that can introduce further incompatibilities;
- graphic devices that can support different resolutions and numbers of colors, and may or may not use a graphic processor;
- and 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[21, 22]. The impact of changes on operating systems and programming languages can by minimized by using de facto standards such as Posix[23] and ANSI C.[24] If any platform dependent code still remains, the simple strategy of isolating the dependent code and documenting its functionality can be used so that a future implementation in another environment is easier. However, the diversity of user interface systems does present some interesting problems, which are discussed next.

Programming 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 in these systems are quite complex, with hundred of functions. Moreover, application programmers often must be experts in several systems, because of the many differences in the various toolkits.

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 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[19] or XVT.[25] Even though an application built with these tools does become 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 common to all supported interface systems (this approach is called "the least common denominator solution"). 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 applications may have to implement interaction mechanisms that do not appear in all interface systems such as list selection boxes or file selection dialogs.

A different strategy is to port a complete interface system to all environments, without using native interface systems, but instead relying on native graphics functions. A characteristic of this solution is that all applications have the same look-and-feel in all computer environments. This may be an advantage for the users of 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. This is the strategy followed in SUIT[26] and in IntGraf.[27]

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 interface 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 the users of many different applications on the same machine, since those 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 for UITs. Such a metaphor is defined by the IUP toolkit. 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 also written.

COMPARISON OF DE FACTO STANDARDS

In order to develop a virtual toolkit, existing interface systems must be considered first. In this section, we examine the systems 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 main program code, allowing rapid prototyping. On the other hand, Motif has so many elements, its UIL is so powerful, and there are so many different forms to compose dialogs that application programmers may have difficulty making reasonable choices. XView, on the other hand, is a very compact toolkit that provides 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, thus increasing prototyping time. The Macintosh Toolbox and MS-Windows do not support any abstract model for defining layouts, forcing programmers to know the size and position of each interface element when drawing the dialogs to scale.

In order to avoid having to design dialogs to scale numerically, Macintosh systems include a graphic editor for the construction of dialogs through direct manipulation of interface elements. In MS-Windows, this type of editor is provided by programming tools such as Borland C++ compiler, Microsoft C compiler, Visual Basic, and Microsoft Access. These tools are useful in creating layouts, but they do not support an abstract layout model. As a consequence, dialogs created visually cannot react automatically to size changes.

There are interactive dialog editors for other interface systems, but almost all of them specify dialog layout by using a concrete rather than an abstract model. In addition, other tools, 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. Exceptions are the ibuild dialog editor in InterViews[28] and FormsEdit for FormsVBT:[29] both use the boxes-and-glue paradigm of TEX[30] to model abstract layout.

The FormsVBT dialog editor provides two views of the dialog specification: a text description of the dialogs in a Lisp dialect and a graphical representation that can be directly manipulated by an interactive editor. The user can interact with both views; changes in one of the views are 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.

Like InterViews and FormsVBT, IUP/LED uses an abstract layout model based on the boxes-and-glue paradigm of TEX. This model allows dialogs to be specified without explicitly defining the position of interface elements, thus enabling automatic repositioning of interface elements when dialogs are resized. This is especially important for interactive graphics applications, when resizing usually means the intention to see more working area; thus, dialog layout must be recomputed after resizing.

In interactive programs, the communication between the application and the user is by nature bidirectional, through dialogs. The application builds dialogs, makes them visible, and then 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 events generated by the user, provides feedback, and then executes the corresponding application routines. 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 Toolbox, use the event model. In this model, the system queues all events generated by the user; the application takes the events from the queue, interprets them and calls the appropriate routines.

The callback model and the event model are equivalent. The callback model may be converted to an event model by associating all 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 complete event queueing may be a limiting factor if the events with no direct meaning for the interface happen too often. For instance, the events generated as the user moves the mouse without pressing any button are not usually meaningful to applications. 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 the mouse passes over a dialog, many events are generated, such as: enter window; many mouse movements; leave window. The X window system uses a client-server architecture; the application may not be executing on the same machine as the one with which the user is interacting. In this setup, the generation of useless events is even more serious, 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 applications to select which events are 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. Moreover, the callback model allows IUP to abstract the events that can occur and also handle any necessary prolog and epilog that may be necessary around application responses to events.

THE IUP/LED SYSTEM

IUP/LED is a user interface system composed of a virtual toolkit (IUP) and a dialog specification language (LED). The IUP/LED system is designed to have the following main characteristics:

- a dialog description language (LED) that can be learned quickly;
- a simple user interface specification model using abstract layout descriptions;
- both native and fixed look-and-feel provided by the IUP toolkit;
- run-time interpretation for LED with minimal overhead, for rapid prototyping;
- portable, in that interfaces can be built for a variety of platforms, ranging from MS-DOS text mode to Unix/Motif;
- available on multiple platforms;
- and expandable.

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 set of functions; the two IUP functions that set and query attributes associated with 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.

LED supports the distinction between abstract and concrete layout. To describe a concrete layout for a dialog is to describe the exact geometric position of each interface object that composes the dialog. On the other hand, to describe an abstract layout for a dialog is to describe the relative positions of these objects. Frequently, programmers have a clear idea of the abstract layout, while the computation of the concrete layout is complicated, tedious and error-prone. Moreover, if a dialog layout is described abstractly, then it is simple to recompute the concrete layout when the the dialog is resized by the user or when elements are added to or removed from the dialog by creating prototypes or by executing the application. The abstract layout description is based on the boxes-and-glue paradigm of the TEX text processor.[30]

IUP/LED associates an attribute, named `WID`, with each interface element. The value of this attribute is the information necessary to access the corresponding interface element in the native system; it is typically a *handle* or pointer to opaque data structures, an integer or a string. Thus, an application may query this value and use it as an argument in calls to the native interface system. In this sense, IUP/LED is also an open system.

The inclusion of a small LED interpreter with an application provides support for rapid prototyping and ease of modification of user interfaces. Only the LED script needs to be altered; it is not necessary to compile and link the application every time the interface changes. In fact, using this approach, the prototype for a user interface can be built in advance of the application. In addition, this type of facility easily supports multiple interfaces for the same application code.

Both the IUP/LED development system and the supporting environment were designed to be portable[31] in the sense that the installation of both these components of IUP/LED on a new computer platform requires much less effort than the effort required to rewrite IUP/LED for that platform. This goal was achieved through the combination of the portability strategies described previously. In this way, IUP/LED can be (and has been) implemented in environments as different as MS-DOS in text mode and MS-Windows. Thus, IUP/LED interface descriptions can be prepared and run on a wide variety of computing platforms.

In IUP/LED, 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; they correspond to top-level windows. Writing an application consists of specifying its dialogs (possibly by using LED) and implementing the associated application routines.

LED: a language for dialog specification

LED is an expression language for specifying dialogs; it supports three important aspects of an interface system:

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

Independence of the user interface code from the application code is achieved by having LED specifications reside in external text files. This also allows rapid prototyping because LED files are interpreted at run time and no actual application functions are needed; it is only necessary that the main program calls IUP to load and interpret LED specification. This single program can be used to create and test 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, users can themselves modify dialog specifications in order to create simplified versions of the program, to translate the interface into another language, or even re-arrange the dialogs completely.

Layout model

The LED language supports an abstract user interface model, where dialogs are defined by their abstract layout and the elements that compose the dialogs are mostly specified by their function and not by their final appearance. In LED, programmers need only to provide some parameters associated with the functionality of each interface element; appearance attributes may be specified, but they are not mandatory.

By using an abstract interface model, application programmers 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 have a IUP driver for the new native interface system; this driver is written once only, by expert programmers. In this way, programs 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 provides an API that implements the abstract model supported by LED.

The layout model used in LED is based on the boxes-and-glue paradigm of the TEX text processor.[30] This model is simple, easily understood, and is able to maintain abstract layouts, independent of size and complexity. The relative position of the interface elements that compose a dialog remain unchanged after the dialog has been resized by the application user, or by the addition or removal of elements. Programmers are freed from having to compute sizes and positions for the interface elements in each dialog, a tedious, error-prone task that must be done several times during the software cycle, and also at run time.

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 (mostly related to appearance) is explicit in the syntax of LED.

The syntax of expressions in LED is simply $n = f[a](p)$, where:

- n 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 (currently, `button`, `canvas`, `dialog`, `fill`, `frame`, `hbox`, `image`, `item`, `label`, `list`, `matrix`, `menu`, `radio`, `submenu`, `text`, `toggle`, `valuator`, `vbox`, `zbox`);
- a is a list of attribute-value pairs, in the form $a_1 = v_1$, $a_2 = v_2$, ..., where a_i is the name of an attribute and v_i is its value (a string);
- 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 created with LED, even though these elements may be fully active.

Example

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 (labeled "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. A specification in LED for this layout follows immediately from this verbal description:

```
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)
```

This example specification uses the interface elements `dialog`, `vbox`, `hbox`, `fill`, `label` and `button`. These and other interface elements are described below. In the example, all elements have been named, but this is not necessary, specially for intermediate elements. An equivalent specification without intermediate names is given below. Note that the dialog still needs a name (`confirm`) so that it the application can show it to the user.

```
confirm = dialog[TITLE="Attention"](
          vbox(
```

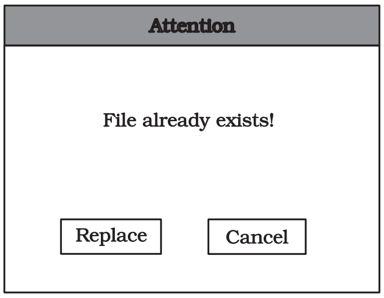

Figure 1. Example dialog

```
fill(),
hbox(
     fill(),
     label("File already exists!"),
     fill()),
fill(),
hbox(
     fill(),
     button("Replace", do_replace),
     fill(),
     button("Cancel", do_cancel),
     fill())))
```

Interface elements

The interface elements available in LED are divided in the following categories:

- *grouping*: define a common functionality for a group of elements;
- *composition*: define a form to exhibit the elements;
- *filling*: 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 tree structure. The structure corresponding to the example dialog in Figure 1 is shown in Figure 2. This hierarchical structure allows dialogs to be gradually specified, combining simple, previously tested dialogs into more complex ones.

Grouping elements. 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.

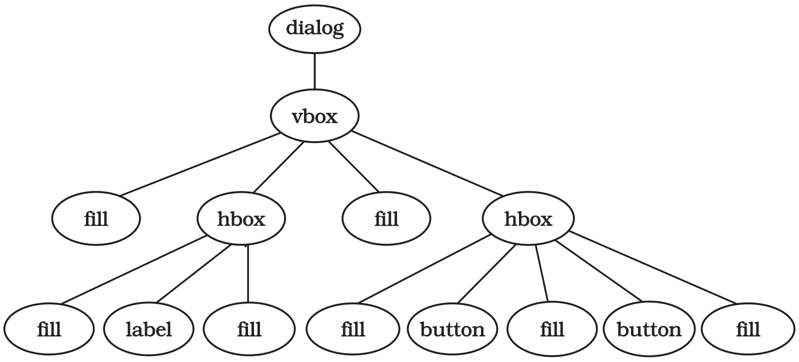

Figure 2. Dialog structure for the example specification

Composition elements. 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 example LED code for the `confirm` dialog 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.

There is a third composition element, not present in TeX, called `zbox`. This element models a stack of interface objects; only the "top" object is visible at any time.

Fill elements. There is only one filling element: `fill`; 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`). In the example, `fill`s are used 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, not in the center.

Primitive elements. The primitive elements currently available in LED are:

- `button`;
- `canvas`: working area;
- `frame`: creates a border around an interface element;
- `image`: static image;
- `item`: menu item;
- `label`: static text;

- `list`: string list with scrollbar;
- `matrix`: a matrix of `text` cells, like a spreadsheet;
- `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 understood and have the same behavior in all interface systems. The interface element `canvas` is a different element because it is the main link between the graphical part of an application and the interface system. It is through `canvas`es that application objects are exhibited and manipulated by the user. This intimate connection with 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 and Mullamey.[32]

In IUP/LED, the behavior of `canvas` is simple: all events that happen on a `canvas` are passed on to the application, which is responsible for handling them. However, before passing events to the application, the IUP driver does handle any necessary prolog and epilog; this typically happens for optimizing `canvas` redraw by setting the clipping area to the exposed area.

It is important to note that, among primitive elements, `canvas` is the only one that competes with `fill` for empty spaces. Thus, when a dialog is resized, its working area is also resized accordingly.

Attributes

Attributes for interface elements are implemented as environment variables represented by the expression $[a = v]$, where a is the name of a variable and v is its value (a string). IUP/LED implements an inheritance mechanism for attributes: the variables defined for an element are automatically exported down 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 value defined for 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, one could write:

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

Some variable names are recognized by IUP/LED and represent attributes of native interface elements. The majority of these attributes control appearance such as color, character fonts, and cursor style. Some attributes define functionality; for instance, the `HOTKEYS` attribute associates function keys with dialogs.

Names not recognized by the system can be used by the application for any purpose; IUP/LED stores these attributes but does not try to interpret them. This feature provides a general-purpose, extensible attribute table, which may be used by the application; in particular, interface objects can maintain their own "state." This mechanism also allows platform dependent attributes to be specified and interpreted only by the corresponding driver, with no consequences for other platforms. Thus, it is possible to fine-tune the interface for each platform and still maintain a single LED specification.

IUP: a virtual toolkit for supporting LED

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

- converting LED specifications to native interface system objects;
- creating interface elements directly, 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 has two main modules: a driver that maps virtual interface objects to native interface objects, and a geometry manager that converts abstract layouts to concrete layouts, by computing sizes and locations for native interface objects. The geometry manager acquires size information from the driver. Of course, the driver depends on the native system, but not the geometry manager. Thus, although there is a different library implementing IUP/LED for each platform, this is transparent to interface programmers.

IUP is written in ANSI C and has been ported to many different environments, such as Microsoft Windows, OpenLook via XView, Motif, and DOS. For DOS, which does not contain a native interface system, we have written a complete and portable interface system having an appearance similar to Motif.

The control flow in an application that uses IUP is analogous to the ones that use other toolkits and can be summarized as follows:

1. initialize IUP, by calling `IupOpen`;
2. create dialogs by loading and interpreting LED specifications with `IupLoad`, or by calling IUP functions to create each interface element;
3. register the functions corresponding to actions with `IupSetFunction` (in the example, `do_replace` and `do_cancel` are actions and not application functions);
4. yield control to IUP by calling `IupMainLoop`, which waits for user actions and calls the corresponding application functions.

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 because IUP functions expect a handle. Therefore, to refer to an interface element created in LED, an application has to call `IupGetHandle`. The following code uses IUP to specify the dialog in Figure 1.

```
IupLoad("attention.led");
IupSetFunction("do_replace", (Icallback) f_replace);
```

```
IupSetFunction("do_cancel",  (Icallback) f_cancel);
IupShow(IupGetHandle("confirm"));
```

Even though this code does not check for errors, all IUP functions return a code that indicates success or failure in the execution of the function.

Creating interface elements in IUP

Interface elements can be created dynamically by calling IUP. However, the creation of interface elements with IUP and with LED differs 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 an element is created with IUP, the corresponding function returns a handle, not a string name. The function `IupSetHandle` may be used to associate names with interface elements, after they have been created, but this is not necessary. The second difference is that, in LED, the definition of the attributes happens at creation time, whereas in IUP the element has to be created before its attributes can be defined. The following code creates the example dialog using IUP.

```
Ihandle *cancel,*replace,*warning,*buttons,*prompt,*body,*confirm;
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);
IupSetAttribute(confirm, "TITLE", "Attention");
IupSetFunction("do_replace",(Icallback) f_replace);
IupSetFunction("do_cancel", (Icallback) f_cancel);
IupShow(confirm);
```

As in LED, it is not necessary to keep handles to intermediate elements; however, no operation is possible on elements for which no handle is stored. The same dialog can be created with the following code:

```
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));
```

Implementation

The main module in the implementation of IUP/LED is the geometry manager, which converts the abstract layout model to a concrete model. The creation of a driver for a specific system is simply an arduous exercise in translating abstract IUP calls into calls to the native API.

The conversion algorithm implemented in the geometry manager has three phases (see Appendix). The first phase computes the smallest size that holds the interface element. This size is called the natural size of each element and is defined in Table I. The second phase computes current sizes, that is, the size with which elements will actually be exhibited to the user. The third and last phase computes the final position of each interface element. Before implementing this algorithm, the following questions related to the appearance of the user interface must be answered:

1. What is the policy for distribution of empty space among `fills` and `canvases`?
2. Does a `fill` or `canvas` that is deep in the hierarchy receive less space than a shallow `fill` or `canvas`? If so, how much less?
3. When a user explicitly defines the size of an element (by specifying the `SIZE` attribute), what size is meant, the natural or the concrete one?
4. In what units 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 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 application objects. Therefore, it seems reasonable that increasing the size of a dialog should increase the size of the `canvas`. In other words, resizing is interpreted as meaning intention to see more work area, not to spread interface elements further apart.

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 received the same amount of space, irrespective of their level, then it would be impossible to divide a dialog in regions as shown in Figure 3, because all elements would receive the same amount of space; the dialog shown in Figure 3 would appear as illustrated in Figure 4 (a or b, depending on how the layout was defined: a vertical box containing two horizontal boxes; or a horizontal box containing two vertical boxes). The algorithm implemented in IUP/LED divides a dialog into regions, distributing the empty space of a box equally among its extensible elements, which then divide their empty space equally among their own elements, and so on. Thus, the outer elements get more space then the inner ones, in exponential proportion.

Our answer to the third question is that when a user chooses a size, the user wants the interface element to be shown in that specified size and not in one computed by the IUP. Therefore, it becomes clear that the size specified by the user refers to the current size. Thus, the size of such an interface element must not be recomputed after the dialog is resized. The only exception is resizable `dialogs`, whose sizes are always recomputed.

Regarding the last question, it is clear that raster units or pixels should not be used as a unit of size because of the obvious dependency on device resolution. In

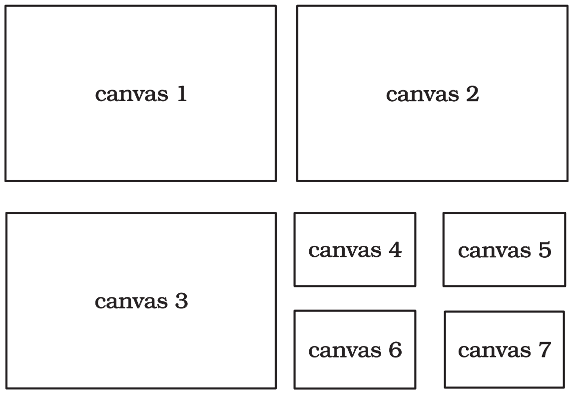

Figure 3. Seven canvases dividing a dialog in symmetrical areas

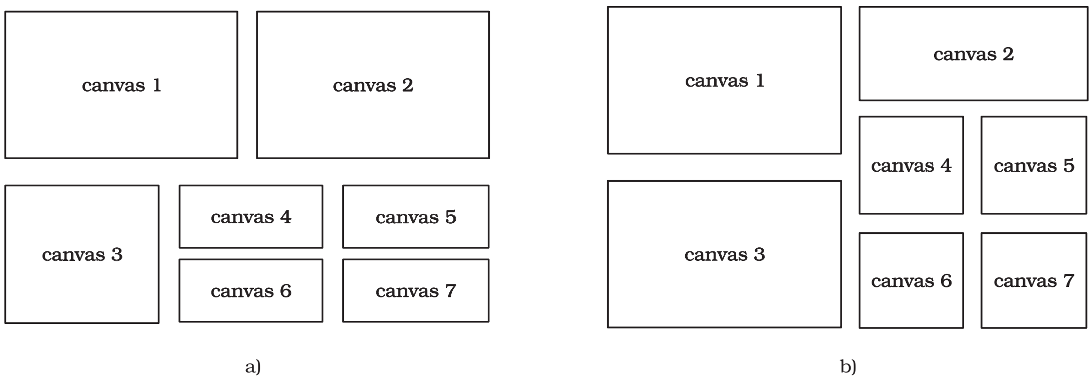

Figure 4. Seven canvases dividing a dialog in asymmetrical areas

IUP/LED, the values related to sizes are proportional 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 not magical or sacred; they are also used by MS-Windows). The use of this kind of unit guarantees that the interface elements will show the users the same information, irrespective of the output device.

Algorithm for computing concrete layout

The Appendix contains pseudo-code for three recursive algorithms that have been implemented to convert abstract layout to concrete layout. Algorithm A computes natural sizes; Algorithm B computes current sizes and distributes empty space; and Algorithm C computes final sizes and positions.

Algorithm A 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.

Algorithm A receives as input a node of the hierarchical structure of a dialog, representing an interface element and computes its natural size. The return value

Table I. Natural size of elements

Element	Natural Size
`button`	somewhat larger than its text or image
`canvas`	the size of a character
`dialog`	the size of the element it contains
`fill`	zero
`frame`	somewhat larger than the size of the element it contains
`hbox`	height equal to the height of its highest element; width equal to the sum its elements width
`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
`matrix`	dependent on the native system
`menu`	minimum to hold all its elements
`radio`	it has no size for it only defines functionality
`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
`vbox`	height equal to the sum its elements height; width equal to the width of its widest element
`zbox`	height equal to the height of its highest element; width equal to the width of its widest element

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, but an `hbox` that contains a `canvas` can grow in both directions with high priority. This information is useful to compute the current size and to determine the distribution of empty space.

All three algorithms start at the top of the hierarchical structure of a dialog, and recursively explore the hierarchy. Algorithm A prepares interface elements for the computation of their current size. Algorithm B computes the current sizes of the interface elements and distributes empty space. To complete the conversion from abstract layout to concrete layout, Algorithm C positions interface elements.

Since the three algorithms traverse the tree structure of a dialog exactly once, the conversion from abstract to concrete layout is linear in the number of interface elements contained in a dialog. This complexity is adequate for real time recalculation of 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. Some native toolkits are specially slow at erasing and redrawing interface elements.

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. Some forty programmers at TeCGraf have used IUP/LED over the past three years to develop several substantial products. The layout model and the toolkit were found to be simple to learn and use by programmers with all levels of expertise. Moreover, IUP/LED is being successfully used in courses on Computer Graphics and User Interfaces at PUC-Rio.

In this section, we report on two early experiences in the use of IUP/LED soon after its implementation was completed. that served to validate the design. We first examine the use of IUP/LED in TeCGraf's PETROX project. Because this project required the definition of almost fifty dialogs, we believe it provides a good evaluation of the abstract layout model. We then examine the use of IUP/LED in a Computer Graphics course for Engineering students, where they had to create an interactive graphics editor, a program that requires a substantial amount of interaction between IUP/LED and the graphics system.

PETROX project

The PETROX project required the creation of a multi-platform interactive program for editing chemical process diagrams that provide input data 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 had many interface elements in common, the engineer created a library of common LED specifications; every time a dialog needed a common element, it was copied from the library and reused in the dialog specification. As the dialogs were built, a need was found for two new interface elements that were not originally in IUP/LED:

- the *drop down list*, which provides a different presentation for the primitive element `list`. 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. IUP/LED now implements drop down lists as an attribute for `list`;
- the *vector*, that allows values to be attributed to a vector without having to define a text 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. The possible values of `ALIGNMENT` are `TOP`, `CENTER` and `BOTTOM`, for `hbox`, and `LEFT`, `CENTER` and `RIGHT`, for `vbox`. IUP/LED now contains the more powerful `matrix` primitive.

Graphic editor

In the Computer Graphics course, Engineering students had to create an interactive graphics editor using IUP/LED as the interface system and a local implementation of GKS as the graphics system. They started using IUP/LED after a single lecture describing the abstract layout model LED and the IUP toolkit. A brief reference manual describing the functions and the attributes for each interface element was provided to support this activity. Although they were not sophisticated programmers, the students were easily able to specify dialogs and use the toolkit to build good user interfaces for the graphic editor. 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 in programming user interactions using callbacks.

The graphics system was used passively and all input was handled by IUP. A minor problem was detected: IUP coordinates are in raster units with origin at the upper right corner of the `canvas`, whereas GKS needs world coordinates defined by the programmer with origin at the lower left corner. To solve this problem, a transformation routine between both systems was added to the IUP interface for GKS.

COMPARATIVE ANALYSIS

There are other interface systems, such as IntGraf,[27] CIRL/PIWI,[19] XVT,[25] OIT,[33] and SUIT,[26] that provide solutions to user interface portability problems. In this section, we make a comparison of IUP/LED with CIRL/PIWI and SUIT, since in an earlier paper[19] CIRL/PIWI was compared with the other toolkits.

CIRL/PIWI

CIRL/PIWI[19] 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 that 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 handling routine.

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

- the implementation of an interface layout language;
- layouts are defined without having to define element coordinates;
- the implementation of a virtual toolkit;
- and dialogs inherit the native look-and-feel.

Although the goals are similar, CIRL/PIWI and IUP/LED were developed independently. 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 were not 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 compiled and linked to the application. LED specifications are interpreted at run time, potentially decreasing the time required to produce a prototype;
- interface elements can only be created through CIRL, while in IUP/LED the interface elements can be defined by using either LED or IUP;
- IUP/LED provides a general-purpose 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; fine-tuning is done based on the tagging created by the CIRL compiler, and it is not possible to attach application specific information to interface elements;
- in CIRL/PIWI, the communication between interface elements and the application is based on the event model; in IUP/LED, this communication 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 known to the IUP driver for the native interface system. Nevertheless, IUP allows native interface elements to be accessed through the `WID` attribute of the corresponding virtual elements, allowing dynamic fine-tuning through the functions of the native interface system.

SUIT

SUIT[26] (Simple User Interface Toolkit) is a portable user interface toolkit developed by the University of Virginia. The library implementing SUIT contains all standard interface objects, and also an interactive layout editor that can be invoked at run-time, qualifying SUIT as a UIMS. The portability of SUIT is achieved by using a fixed look-and-feel implemented with SRGP, a portable graphics library.[34] Although SUIT was designed primarily as an educational tool, it has been successfully used in many professional research laboratories[35].

The main similarities and differences between SUIT and IUP/LED can be summarized as follows:

- SUIT implements a "true" toolkit on top of a portable graphics library, thus providing only fixed look-and-feel. IUP/LED uses a virtual toolkit that provides both fixed and native look-and-feel.
- the API in both systems is small and easy to learn.
- both SUIT and IUP/LED support attributes for interface objects and attribute inheritance. Attributes are called "properties" in SUIT and can be edited at run-time using a built-in graphical properties editor. Like IUP/LED, SUIT allows user defined properties. Unlike IUP/LED, however, SUIT properties are typed.
- some interface objects in SUIT can be presented in more than one way, which may be selected at run-time by the user or via properties. In IUP/LED, the presentation of an interface object depends on the native interface systems, and would be selected via attributes as in SUIT, but the standard systems do not support multiple presentation styles. However, multiple presentation styles can be implemented in IUP/LED with `zboxes`, a more powerful construct.
- although the SUIT toolkit describes layouts concretely, by giving explicit geometrical coordinates for interface objects, layouts can also be defined interactively, using the built-in layout editor. This editor can be used by the programmer and also invoked by the user at run-time. Layouts in IUP/LED are described abstractly and created either as a set of procedure calls using IUP or a set of declarations using LED. SUIT has composition objects that can implement horizontal and vertical arrangements, but not fill objects, and thus cannot support abstract layouts.
- the current layout of a SUIT interface can be saved at run time in an external text file that can be later interpreted to recover the layout in a subsequent run of the program. However, SUIT actually writes a C source file that must be carefully edited to avoid parsing mistakes. Moreover, this file does not contain information about the actions associated with the interface objects, because it does not know the names of the corresponding C functions. IUP/LED has "actions" that are dynamically bound to C functions. This allows LED files to contain information about interface actions without knowing names of C functions.
- both SUIT and IUP/LED were designed to support rapid prototyping. SUIT supplies built-in layout and attribute editors. IUP interprets LED specifications.
- because SUIT is written on top of a portable graphics library, it provides a portable way of drawing application data in `canvases`. IUP provides a slightly more complex, but orthogonal solution: the `WID` attribute provides access to native `canvases`. Used with a platform-independent graphics package, this mechanism allows portable graphics on `canvases`.

CONTRIBUTIONS

IUP/LED has made a number of contributions to the 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 implemented in many different interface systems;
- LED is an expression language with a simple syntax that can be learned quickly;
- LED describes a dialog using its functions; appearance attributes are optional;
- LED allows applications to be customized at run time, for different users and platforms, by the users themselves. There is no need to recompile or relink an application to customize it;
- LED provides an attribute mechanism that allows specific adaptations to an interface system, and makes it possible to attach application information directly to the interface elements;
- IUP is a virtual toolkit that allows portable interactive programs to be built without forcing programmers 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 a fixed look-and-feel, which helps the user of only one application who needs to run it 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.

Although IUP does not contain primitives for drawing on `canvases`, complete portability of graphics applications that use IUP can be achieved by using a platform-independent graphics package, such as GKS or the one defined by PIWI. Such graphics packages only need to query the value of the `WID` attribute of `canvases` to gain access to the necessary information for using native graphics primitives. In other words, portability of interfaces to application data can be ensured by combining a passive portable graphics package with an interface element that receives events. IUP provides an abstraction for an event recipient (`canvas`) and a simple mechanism for linking these two components in a portable, orthogonal way.

CONCLUSIONS

In this paper, we have described a portable user interface development tool called IUP/LED. LED is an expression language for specifying dialogs, and IUP is a virtual toolkit for creating dialogs and for connecting dialogs specified in LED to the native system. IUP/LED 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 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 programmers from computing interface element sizes and positions;
- and it maintains abstract dialog layouts after the size is changed 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, interface designers can always find a geometrically regular layout that is able to expose the required information adequately.

The dialog specification 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:

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

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

- convert the declarations in LED to native interface system objects;
- create interface elements dynamically without using LED;
- bind application functions 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, specially 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. Optional information for interface elements are not provided by calling functions but instead 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 approach is that it is easy to extend IUP. On the other hand, programmers must learn not only the API but also which attributes are valid for each element. However, programmers do not need to learn about attributes until they know how to implement the desired functionality; this important first step is made easier by the small size of the IUP toolkit.

The main difficulty in developing IUP/LED was in building the algorithm to transform the abstract layout into a concrete one, because many natural alternatives exist for maintaining abstract layout.

The following improvements to IUP/LED are currently under development:

- developing an interactive dialog editor for IUP/LED,[36]
- allowing reference to interface elements by name rather than through their handle to allow reuse of parts of dialogs without copying;
- adding a platform-independent graphics package;
- implementing help and clipboard mechanisms;

- and creating APIs to other languages, such as Fortran, C++, and Lua.[37] Lua is an extension language developed by TeCGraf that is both interpreted and procedural and could be a more powerful replacement for LED.

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 does not directly deal with the user interface, it enables the integration of data between applications.

ACKNOWLEDGEMENTS

We would like to thank the staff at TeCGraf for using and testing IUP/LED in its industrial products. IUP/LED is being developed in partnership with the research center at PETROBRÁS (The Brazilian Oil Company). The authors are partially supported by research and development grants from the Brazilian Council for Scientific and Technological Development (CNPq), the Natural Sciences and Engineering Research Council of Canada, and WATCOM.

APPENDIX: ALGORITHMS FOR COMPUTING CONCRETE LAYOUT

Algorithm A. *Compute natural sizes; return expansion information*

```
function Nsize(n): integer
   if n is a dialog
       compute Nsize(child(n))
       expansion(n) ← both directions
       Nwidth(n) ← Nwidth(child(c))
       Nheight(n) ← Nheight(child(c))
   else if n is an hbox
       expansion(n) ← no direction
       Nwidth(n) ← 0
       Nheight(n) ← 0
       for each child c of n do
           expansion(n) ← expansion(n) combined with Nsize(c)
           Nwidth(n) ← Nwidth(n) + Nwidth(c)
           Nheight(n) ← max(Nheight(n),Nheight(c))
   else if n is a vbox
       expansion(n) ← no direction
       Nwidth(n) ← 0
       Nheight(n) ← 0
       for each child c of n
```

```
            do expansion(n) ← expansion(n) combined with Nsize(c)
            Nwidth(n) ← max(Nwidth(n),Nwidth(c))
        Nheight(n) ← Nheight(n) + Nheight(c);
    else if n is a canvas
        get natural size from native system
        expansion(n) ← both directions with high priority
    else if n is a fill
        Nwidth(n) ← 0
        Nheight(n) ← 0
        if n is inside an hbox then
            expansion(n) ← horizontal with low priority
        else if n is inside a vbox then
            expansion(n) ← vertical with low priority
    else
        get natural size from native system
        expansion(n) ← no direction
return expansion(n)
```

Algorithm B. *Compute current sizes and distribute empty space*

```
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)
    if n is a dialog
        Csize(child(n),Cwidth(n),Cheight(n))
    else if n is an 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))
  else if n is a 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)
```

Algorithm C. *Compute final sizes and positions*

```
function position(n,x,y)
  x(n) ← x
  y(n) ← y
  if n is a dialog
      position(child(n),x,y)
  else if n is an hbox
      foreach child c of n do
         position(c,x,y)
         x ← x + Cwidth(c)
  else if n is a vbox
      for each child c of n do
         position(c,x,y)
         y ← y + Cheight(c)
```


REFERENCES

1. H. R. Hartson and D. Hix, 'Human-computer interface development: concepts and systems for its management', *ACM Computing Surveys*, **21**(1), 5–92 (1989).
2. D. Heller, *XView Programming Manual*, O'Reilly & Associates, Inc., second edition, 1990.
3. Open Software Foundation, *OSF/Motif Programmer's Guide*, Prentice Hall, 1991.
4. Sun Microsystems Inc., *Open Look Intrinsics Toolkit Widget Set Programmers's Guide*, 1990.
5. C. Petzold, *Programming Windows: the Microsoft Guide to Writing Applications for Windows 3*, Microsoft Press, 1990.
6. Apple Computer, Inc., *Inside Macintosh*, Addison-Wesley, 1985.

7. M. Green, 'The University of Alberta User Interface Management System', *Proceedings of SIGGRAPH'85*, pp. 205–213.
8. H. R. Hartson, D. Hix, and R. W. Ehrich, 'A human-computer dialogue management system', *Proceedings of INTERACT'84*, London, September 1984, pp. 57–61.
9. A. Marcus and A. van Dam, 'User interface developments for the nineties', *IEEE Computer*, **24**(9), 49–57 (1991).
10. Microsoft, *Visual Basic Programming System for Windows version 2.0, Programmer's Guide*, Microsoft Corporation, 1992.
11. A. Marcus, *Graphic Design for Electronic Documents and User Interfaces*, Addison-Wesley, 1992.
12. 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*, 169–175 (1992).
13. D. Hix, 'Generation of User Interface Management Systems', *IEEE Software*, 77–87 (1990).
14. 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 (1990).
15. J. Foley, W. C. Kim, S. Kovacevic, and K. Murray, 'Defining interfaces at a high level of abstraction', *Software*, 25–32 (1989).
16. B. A. Myers, *Creating User Interfaces by Demonstration*, Academic Press, 1988.
17. B. A. Myers and M. B. Rosson, 'Survey on user interface programming', *Proceedings of CHI'92*, 1992, pp. 195–202.
18. B. A. Myers, 'State of the art in user interface software tools', in H. R. Hartson and D. Hix (eds.), *Advances in Human-Computer Interaction 4*, Ablex Publishing, 1993, pp. 110–150.
19. D. D. Cowan, C. M. Durance, E. Giguère, and G. M. Pianosi, 'CIRL/PIWI: A GUI toolkit supporting retargetability', *Software: Practice & Experience*, **23**(5), 511–527 (1992).
20. D. Frey, 'Unix vs. Unix', *Dr. Dobb's Journal*, **146**, 28–35 (1988).
21. G Blackham, 'Building software for portability', *Dr. Dobb's Journal*, **146**, 18–27 (1988).
22. C. M. Durance, 'An approach to application software mobility across user interface toolkits', *Master's Thesis*, Faculty of Mathematics, University of Waterloo, 1990.
23. IEEE, *Standard 1003.1-1988 (Posix)*, IEEE, 1988.
24. B. Kernighan and D. Ritchie, *The C Programming Language*, Addison-Wesley, second edition, 1988.
25. M. J. Rochkind, 'XVT: A virtual toolkit for portability between window systems', *USENIX*, 151–163 (1989).
26. R. Pausch, N. Young, and R. DeLine, 'SUIT: The Pascal of user interface toolkits', *Proceedings of UIST'91*, 1991, pp. 117–126.
27. TeCGraf, *Manual de Referência do IntGraf: Sistema de Interface Gráfica com Usuário*, PUC-Rio, 1991.
28. M. A. Linton, J. M. Vlissides, and P. R. Calder, 'Composing user interfaces with InterViews', *IEEE Computer*, 8–22 (1989).
29. G. Avrahami, K. P. Brooks, and M. H. Brown, 'A two-view approach to constructing user interfaces', *Computer Graphics*, **23**, 137–146 (1989).
30. D. E. Knuth, *The TeXbook*, Addison-Wesley, 1984.
31. D. D. Cowan and T. A. Wilkinson, 'Portable software: an overview', *Proceedings of the 1984 Canadian Conference on Industrial Computer Systems*, Ottawa, May 1984, pp. 68–1–68–7.
32. F. Neelamkavil and O. Mullarney, 'Separating graphics from applications in the design of user interfaces', *The Computer Journal*, 437–443 (1990).
33. Neuron Data, 'Open Interface Toolbox'. 156 University Avenue, Palo Alto, CA 94301, 1991.
34. J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley, second edition, 1990.
35. R. Pausch, M. Conway, and R. DeLine, 'Lessons learned from SUIT, the Simple User Interface Toolkit', *ACM Transactions on Office Information Systems*, **10**(4), 320–344 (1992).
36. R. O. Prates, 'Visual LED: uma ferramenta interativa para geração de interfaces gráficas', *Master's Thesis*, Departamento de Informática, PUC-Rio, 1994.
37. R. Ierusalimschy, L. H. de Figueiredo, and W. Celes Filho, 'Lua—an extensible extension language'. submitted to *Software: Practice & Experience*.