Tools for Creating User Interfaces: An Introduction and Survey

Brad A. Myers
January, 1988
CMU-CS-88-107(?)

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA
15213-3890
bam@cs.cmu.edu

ABSTRACT

Creating good user interfaces for software programs is a very difficult task. There are no guidelines or techniques that will guarantee that the software will be "easy-to-use," and software implementors have generally proven poor at providing user interfaces that people like. Consequently, user interface software must often be prototyped and modified repeatedly. In addition, user interface software is inherently difficult to write, because it frequently requires that multiple devices be controlled (for example, a keyboard and a mouse) each of which may be sending streams of input events asynchronously. Also, user interfaces typically have stringent performance requirements to insure that there is no perceived lag between a user's actions and the system's response. The most popular style of user interfaces (called "Direct Manipulation" interfaces) is one of the most difficult kinds to implement. Therefore, there is a great interest in software tools to aid in this process. This article discusses several different types of software tools and examples of their use.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976 under contract F33615-87-C-1499 and monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, OH 45433-6543. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency of the US Government.

1. Introduction

The *user interface (UI)* of a computer program is the part that handles the output to the display and the input from the person using the program. The rest of the program is called the *application* or the *application semantics*. This paper discusses programs, called *user interface tools*, that help create and manage user interface software. These tools come in two general forms: *user interface tool kits* and *User Interface Management Systems (UIMSs)*. A user interface tool kit is a library of *interaction techniques*, where an interaction technique is a way of using a physical input device (mouse, keyboard, tablet, rotary knob, etc.) to input a certain type of value (command, number, percent, location, name, etc.). Examples of interaction techniques are menus, graphical scroll-bars, and on-screen "light buttons." When using a user interface tool kit, the programmer is responsible for invoking and organizing the interaction techniques. A User Interface Management System (UIMS), on the other hand, is a tool that helps a programmer create and manage many aspects of user interfaces. In addition to a tool kit, UIMSs usually contain a *Dialogue Control Component*, which handles the sequencing of events and interaction techniques, and may also contain an *Analysis Component*, which helps study and evaluate the user interface after it has been created. UIMSs have been studied at a number of workshops and conferences [Thomas 83] [Olsen 84] [Pfaff 85] [Olsen 87a].

There are four different classes of people involved with any UIMS and it is important to have different names for them to avoid confusion. One person is the designer of the UIMS, who is called the *UIMS creator*. The next person is the designer of a user interface, and will use a UIMS. This person may be a programmer or a graphic artist, depending on the specification technique used by the UIMS, and will be called the *user interface designer* or just *designer* for short. Another person involved is the programmer that creates the application program which uses the user interface created by the user interface designer. This person is the *application programmer*. The final person involved is the person who actually uses the final product. This person is the *end user* or just *user*. Note that although this classification discusses each role as a different person, in fact, there may be many people in each role or one person may perform multiple roles.

This paper discusses why user interface software is hard to create and why user interface tools are desirable. It then presents a number of existing styles of user interface tools and discusses the advantages and disadvantages of each. The paper concludes with a summary of the predominant problems with most existing user interface tools.

2. Problems with User Interface Software

Creating a good user interface for a system is a difficult task, and a large number of existing programs have very poor user interfaces: they are hard to learn, hard to remember, inefficient to use, have limited on-line help, etc. In addition, user interface software is often large, complex, and difficult to debug and modify. The user interface for an application is usually a significant fraction of the code. One study found that the user interface portion was between 29% and 88% [Sutton 78]. In artificial intelligence applications, for example, surveys report that 40% to 50% of the code and run-time memory are devoted to user interface aspects [Bobrow 86] [Mittal 86]. Unfortunately, it is generally the case that as user interfaces become easier to use for the end user, they become more complex and harder for the UI designer to create. The easy to use "Direct Manipulation" interfaces popular with most modern systems are among the most difficult kinds to implement [Williams 83] [Smith 82]. In these interfaces, the objects of interest are visible on the screen and the user can operate on them by direct reference and rapid, reversible, incremental actions [Shneiderman 83]. Some reasons that Direct Manipulation interfaces are difficult to create are that they often provide (a) elaborate graphics, (b) multiple ways for giving the same command, (c) multiple asynchronous input devices (usually a keyboard and a locator or pointing device such as a mouse), (d) a "mode free" interface, where the user can give any command at virtually any time, and (e) rapid "semantic feedback." *Semantic feedback* is where determining the appropriate response to user actions requires specialized information about the objects in the program. An example is in the Apple Macintosh user interface, when icons highlight when another icon is dragged over them if they perform semantically meaningful operations on the icon being dragged.

In addition to being difficult to create, there are no design strategies that will guarantee that the resulting user interface will be learnable, easy to use, and "user-friendly." Consequently, the only reliable method for generating quality user interfaces is to test prototypes with actual end users and modify the design based on the users' comments [Buxton 80] [Swartout 82] [Mason 83] [Anderson 85]. As reported by Sheil [83] "complex interactive interfaces usually require extensive empirical testing to determine whether they are really effective and considerable redesign to make them so." This methodology is called "iterative design" and has been used in the creation of some of the best current user interfaces: the Xerox Star [Bewley 83], the Apple Lisa [Morgan 83], and the Olympic Messaging system [Boies 85]. A particularly compelling example is presented by Good [Good 84] where a mail system with a conventional textual command interface was iteratively modified. In the final version, without any instruction, 76% of the commands that novices naturally generated performed the expected operation, compared with 7% for the initial version.

Consequently, there is a great desire to make the creation of user interfaces easier and quicker, and to make them easier to modify once created.

3. Motivation for User Interface Tools

To make the user interfaces cheaper and easier to design and implement, a number of different tools have been created. Some of these have been very successful. For example, the Apple MacApp UIMS has been reported to reduce development time by a factor of four or five [Schmucker 86]. In general, the advantages of using user interface tools include:

- **The quality of the interfaces should be higher.** This is because:
  - Designs can be rapidly prototyped and implemented, possibly even before the application code is written.
  - It will be easier to incorporate changes discovered through user testing.
  - There can be multiple user interfaces for the same application.
  - More effort can be expended on the UIMS than may be practical on any single user interface since the UIMS will be used with many different applications.
  - Different applications are likely to have more consistent user interfaces if they are created using the same user interface tool.
  - Conversely, some user interface tools make it easier to investigate different styles for a user interface, and thereby provide a unique "look and feel" for a program.
  - A UI tool should make it easier for a variety of specialists to be involved in designing the user interface, rather than having the user interface created entirely by programmers. Graphic artists, cognitive psychologists, and human factors specialists may all be involved. In particular, professional user interface designers (sometimes called "User Interface Architects" [Foley 84]), who may not be programmers, may be in charge of the overall design.
  - The ability to rapidly modify interfaces should allow system designers and salesmen to try different interfaces on products in front of customers and end users, and immediately incorporate their suggestions.

- **The user interface code will be easier and more economical to create and maintain.** This is because:
  - There will be better modularization due to the separation of the user interface component from the application. This should allow the user interface to change without affecting the application, and a large class of changes to the application (such as changing the internal algorithms) should be possible without affecting the user interface.
  - The reliability of the user interface should be higher, since the code for the user interface is created automatically from a higher level specification.
  - Interface specifications can be represented, validated, and evaluated more easily.
  - It should be easier to port an application to different hardware and software environments since the device dependencies are isolated in the user interface tool.

A comprehensive user interface tool might handle all aspects of the user interface, which include handling all user-visible parts of the display and all aspects of the dialogue between the end user and the application. In particular, the UIMS should:

- handle the mouse and other input devices,
- validate user inputs,
- handle user errors,
- process user-specified aborting and undoing of operations,
- provide appropriate feedback to show that inputs have been received,
- provide help and prompts,
- allow the end user to customize the interface,
- update the screen display when application data changes,
- notify the application when the user updates application data,
- deal with field scrolling and editing,
- insulate the application from the window or screen management functions, and
- automatically evaluate the interface and propose improvements, or at least provide information to allow the designer to evaluate the interface.

The following sections survey a number of existing approaches to user interface tools and evaluate how well they satisfy the above goals.

4. Tool kits

Most window systems come with a tool kit containing routines that application programs can use. These typically include menus of various types, scroll bars, etc. Tool kits come in two basic varieties. The most conventional is simply a collection of procedures that can be called by application programs. Examples of this style include SunTools [Sun 84] and the Macintosh Toolbox [Apple 85]. The other variety uses an "object-oriented" programming style with inheritance [Goldberg 83] which makes it easier for the designer to customize the interaction techniques. Examples of this style include Smalltalk [Tesler 81], and the X.11 Toolkit for the X Window Manager [Scheifler 86]. The GROW tool kit [Barth 86] adds "constraints" to a conventional object-oriented tool kit. *Constraints* allow the designer to specify relationships among objects that are maintained by the system. For example, the designer can specify that a line is connected to two rectangles, and the system will automatically move the line whenever either rectangle is moved. With all tool kits, the designer writes programs in a conventional programming language to control the user interface.

Using a tool kit has the advantage that the final UI will look and act similarly to other UIs created using the same tool kit, but clearly the styles of interaction are limited to those provided. In addition, the tool kits themselves are often expensive to create: "The primitives never seem complex in principle, but the programs that implement them are surprisingly intricate" [Cardelli 85, p. 199].

Another problem with tool kits is that they are often difficult to use. There are typically hundreds of procedures in tool kits implementing various interaction techniques, and it is often not clear how to use the procedures to create a desired interface.

5. User Interface Management Systems

Due to the problems with tool kits, a number of User Interface Management Systems have been created to help the designer combine and sequence the interaction techniques. For example, Apple found that people were having difficulty using the Macintosh Toolbox, so they created the MacApp UIMS [Schmucker 86] (see section 5.1.6). Other UIMSs help designers create the tool kits themselves. Examples of this kind include Squeak [Cardelli 85] (section 5.1.4), Panther [Helfman 87] (section 5.1.7), and Peridot [Myers 87a] [Myers 87b] (section 5.2).

UIMSs come in a large variety of forms. One important way that they can be classified is by how the designer specifies what the interface should be. As shown in Figure 1, some UIMSs use special-purpose languages, some allow the interface to be graphically specified directly, and others automatically generate the interface from the specification of the

functionality of the application. Each of these types is discussed below. Of course, some UIMSs use different techniques for specifying different parts of the user interface. These UIMSs are classified by their predominant or most interesting feature.

Classification	*Examples*
Language-Based	
Menu Trees	Tiger [Kasik 82]
State Transition Diagrams	[Newman 68] [Jacob 85] RAPID/USE [Wasserman 82] [Jacob 86]
Grammar Oriented	Syngraph [Olsen 83]
Event Languages	Squeak [Cardelli 85] ALGAE [Flecchia 87] Sassafras [Hill 87a]
Declarative Languages	Cousin [Hayes 85] Domain/Dialogue [Schulert 85]
Object Oriented Languages	MacApp [Schmucker 86] GWUIMS [Sibert 86] HIGGENS [Hudson 86]
Other	Panther [Helfman 87]
Direct Graphical Specification	Menulay [Buxton 83] Trillium [Henderson 86] RAPID [Freburger 87] Grins [Olsen 85a] Peridot [Myers 87a]
Automatic Creation	Control Panel Interface [Fisher 87] MIKE [Olsen 86] IDL [Foley 87]

Figure 1.
Classification of User Interface Management Systems. These systems are discussed in the following sections.

5.1. Language-Based Techniques

With most UIMSs, the designer specifies the user interface using a special-purpose language. This language can take many forms, including simple menu trees, context-free grammars, state transition diagrams, declarative languages, event languages, object-oriented languages, etc. With most of these systems, the language is used to specify the syntax of the user interface; i.e., the legal sequences of input and output actions. Green [Green 86] provides an extensive comparison of grammars, state transition diagrams, and event languages.

5.1.1. Menu trees

One of the simplest forms of UIMS supports a hierarchy or network of menus. Selecting an option on one menu causes another menu to appear. The Tiger UIMS [Kasik 82] supports a sophisticated menu network which supports skipping levels, aborting, etc. Many of the hypertext systems [Conklin 87] could also be considered UIMSs that manage networks of menus.

5.1.2. State Transition Networks

Since many parts of user interfaces involve handling a sequence of input events, it is natural to think of using a state transition network to code the interface. A transition network consists of a set of states, with arcs out of each state labeled with the input tokens that will cause a transition to the state at the other end of the arc (see Figure 2). In addition to input tokens, calls to application procedures and the output to display can also be put on the arcs in some systems. Newman implemented a simple UIMS using finite state machines in 1968 [Newman 68]. Many of the assumptions and techniques used in modern systems were present in Newman's: different languages for defining the user interface and the semantics (the semantic routines were coded in a normal programming language), a table-driven syntax analyzer, and device independence. Newman's system only handled textual input, but it was apparently the first UIMS.

Jacob [Jacob 85] claims that defining a user interface using state transition diagrams is better than using formal context-free grammars (section 5.1.3) because the time sequence is explicit with diagrams. The specification can be created in a textual or graphical manner. Figure 2 shows a diagram created with Jacob's system. The diagrams can have recursive calls to other diagrams on arcs, so they are classified as Recursive Transition Networks. The interface among the various part of the interface and to the application is through a plethora of global variables, and for all states there must be explicit arcs for any possible erroneous inputs and any universal commands such as HELP and UNDO.

RAPID (RApid Prototyping of Interactive Dialogues) is another transition network system. The user interface part, RAPID/USE (User Software Engineering) [Wasserman 82], is just a small portion of a large system for supporting software engineering. The user interface portion is very similar to Jacob's except that it has more powerful output primitives.

State diagram UIMSs are most useful for creating user interfaces where a large amount of syntactic parsing is necessary or when the user interface has a large number of modes (each state is really a mode). However, most highly-interactive systems attempt to be mostly "mode-free" [Tesler 81], which means that at each point, the user has a wide variety of

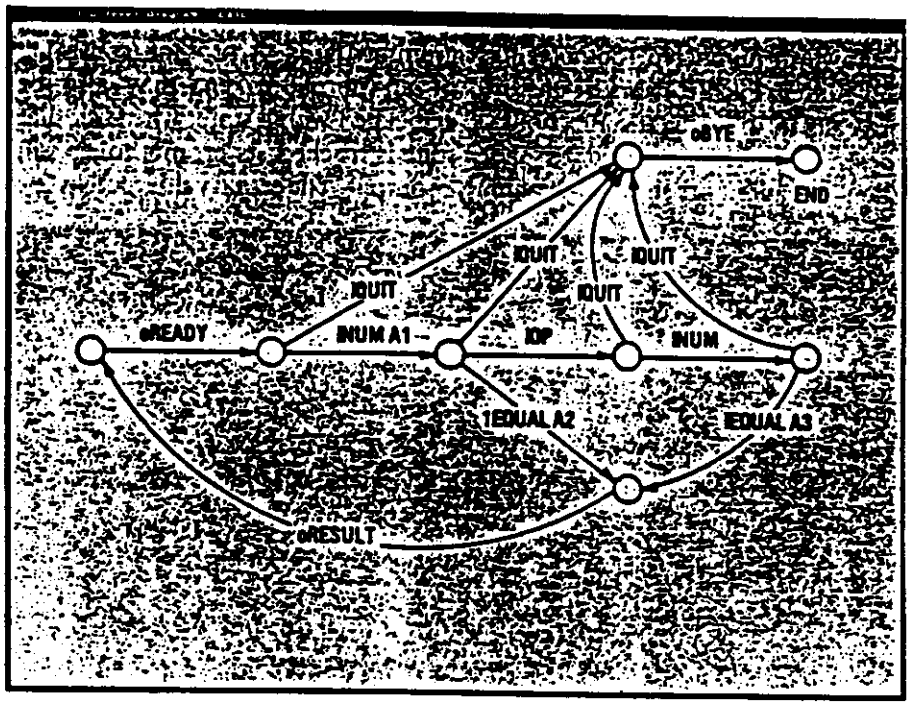


Figure 2.
State diagram description of a simple desk calculator [Jacob 85].

choices of what to do. This requires a large number of arcs out of each state, so state diagram UIMSs have not been successful for these interfaces. In addition, state diagrams cannot handle interfaces where the user can operate on multiple objects at the same time (possibly using multiple input devices concurrently [Buxton 86]). Another problem is that they tend to get very confusing for large interfaces, since they get to be a "maze of wires" and off-page (or off-screen) arcs can be hard to follow.

Recognizing these problems, but still trying to retain the perspicuousness of state transition diagrams, Jacob [Jacob 86] has invented a new formalism, which is a combination of state diagrams with a form of event languages (section 5.1.4). There can be multiple diagrams active at the same time, and flow of control transfers from one to another in a co-routine fashion. The system can create various forms of Direct Manipulation interfaces.

5.1.3. Context-Free Grammars

Most grammar-based systems are based on parser generators used in compiler development. For example, the designer might specify the user interface syntax using some form of BNF. These systems are good for textual command languages, but they have mostly failed when used for graphics programs [Green 86]. Syngraph (SYNtax directed GRAPHics) [Olsen

83] is a system that generates user interface programs in Pascal from a description written in a formal grammar using an extended BNF. The system handles prompting, echoing and errors. Syngraph provides menus and text input and also provides a few predefined interaction devices (locator, valuator, pick, etc.) with some limited ability for tracking. Syngraph concentrates on dealing with semantic error recovery, "Cancel" and "Undo" at the semantic level, and the problem of knowing what to select when multiple items are on the screen at the position of a "pick." Syngraph does not, however, provide semantic feedback or defaults since there is no way for application routines to affect the parsing.

Grammar-based UIMSs, like state diagram UIMSs, are not appropriate for specifying highly-interactive interfaces, since they are oriented to batch processing of strings with a complex syntactic structure.

5.1.4. Event Languages

With event languages, the input tokens are considered to be "events" that are sent immediately to event handlers. These handlers can cause output events, change the internal state of the system (which might enable other event handlers), or call application routines.

The ALGAE UIMS [Flecchia 87] uses an event language which is an extension of Pascal. The user interface is programmed as a set of small event handlers, which ALGAE compiles into conventional code.

The Sassafras UIMS, which implements an Event Response System (ERS) [Hill 87a] [Hill 87b], uses a similar idea, but with an entirely different syntax. This system also adds local variables called "flags" to help specify the flow of control. ERS is especially well suited for handling concurrent interfaces; for example, ones that use multiple input devices at the same time (also called "multi-threaded dialogues"), since it incorporates synchronization mechanisms. ERS can support Direct Manipulation interfaces since it facilitates efficient and frequent communication between the interaction techniques and the application program.

Squeak [Cardelli 85], a textual language for programming mouse-based interfaces, exploits concurrency. Squeak's processes are similar to event handlers and the messages sent to processes are similar to events. Squeak supports many input devices active at the same time, and the primitive input events are mouse button transitions, keyboard key presses, incremental movements of the mouse or other devices, and clock timeouts. Squeak attempts to compile the program into a non-sequential state machine. Although it provides a compact notation for specifying complex, time-dependent interfaces, correct code is unfortunately still fairly difficult to write in Squeak.

All of the UIMSs in this category are explicitly designed to handle multiple processes, which can be important in user interfaces. For example, research has shown that people can be more effective when operating multiple input devices as the same time [Buxton 86]. It is also often easier to program multiple interactions that are available at the same time (so the user can pick which one to use) using multiple processes. The disadvantage of these event languages is that it is often very difficult to create correct code, since the flow of control is not localized and small changes in one part can affect many different pieces of the program. It is also typically difficult for the designer to understand the code once it reaches a non-trivial size.

5.1.5. Declarative Languages

Another approach is to try to define a language that is declarative (stating what should happen) rather than procedural (how to make it happen). Cousin [Hayes 85] and Apollo's Domain/Dialogue [Schulert 85] both allow the designer to specify user interfaces in this manner. The user interfaces supported are basically forms, where fields can be text which is typed by the user, or options selected using menus or buttons. There are also graphic output areas that the application can use in whatever manner desired. The application program is connected to the user interface through "variables" which both can set and access. Based on Domain/Dialogue, Apollo has created a new UIMS called "Open Dialogue." Open Dialogue runs on the X window manager so it is more portable.

The advantage of using declarative languages is that the user interface designer does not have to worry about the time sequence of events, and can concentrate on the information that needs to be passed back and forth. The disadvantage is that only certain types of interfaces can be provided this way, and the rest must be programmed by hand in the "graphic areas" provided to application programs. The kinds of interactions available are preprogrammed and fixed. In particular, these systems provide no support for such things as dragging graphical objects, rubber-band lines, or drawing new graphical objects.

5.1.6. Object Oriented Languages

An important new class of UIMSs provides an object-oriented framework in which the user interface is programmed. These systems typically provide the higher-level "classes" that handle the default behavior and the user interface designer provides specializations of these classes to deal with specific behavior desired in the user interface. This uses the inheritance mechanism built into object-oriented languages [Goldberg 83].

These systems can handle highly-interactive, Direct Manipulation interfaces since there is a computational linkage between the input and the output which the application can modify to provide semantic processing. Although these systems make it much easier to create user interfaces, they are still programming environments, and are clearly inaccessible to non-programmers. Object oriented UIMSs and object-oriented tool kits (section 4) are receiving a great deal of attention now, and show a lot of promise. Many articles about these appear in the annual OOPSLA (Object-Oriented Programming: Systems, Languages and Applications) conference, sponsored by ACM SIGPLAN.

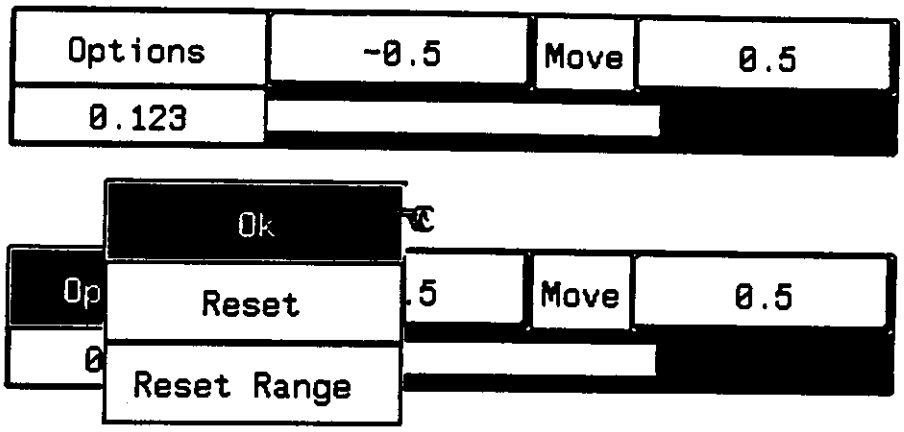
MacApp [Schmucker 86] is programmed in Object Pascal and makes it easier to create Macintosh programs. GWUIMS (George Washington User Interface Management System) [Sibert 86] uses object-oriented programming in Lisp, and provides a classification of interface operations and objects that fit into each class. HIGGENS [Hudson 86] adds a structured data description that allows the UIMS to automatically manage the recalculation and redisplay of objects in an intelligent way when data changes. The structure is also used to support UNDO and REDO.

5.1.7. Other Language Approaches

Panther [Helfman 87] allows interaction techniques to be specified textually using tables. Figure 3 shows an example of a Panther specification. The designer specifies what action should happen in each region of the screen. The options include: the highlighting style, the procedures for drawing the picture in the region, and the procedures to call when a button is pressed in the region. Panther supports menus, forms and sliders (see Figure 3), but the entire specification must be created using typed-in numbers and procedure calls.

5.2. Direct Graphical Specification

The UIMSs described in this section all allow the user interface to be defined, at least partially, by placing objects on the screen using a mouse. This is motivated by the observation that the visual presentation of the user interface is of primary importance in graphical user interfaces, and a graphical tool seems to be the most appropriate way to specify the graphical appearance. Another advantage of this technique is that it is usually much easier to use for the designer, and some of these systems, including Menulay, Trillium, RAPID, and Peridot, can be used by non-programmers. Some of the disadvantages with this style of UIMS are that the UIMS itself is usually more complicated to build, the range of interfaces that can be created is usually fairly limited, and application programs are required to handle such things as help,



(a)

```
name      coordinates x1,x2,y1,y2          highlight style  draw flag  parent name  draw routine                  selection routines
OPTS  {     1, 103,   1,  32,                1,               0,         BAR,         PANdbox("Options"),          BARpopup(POP),        0,  0 }
RAN0  {   104, 211,   1,  32,                1,               1,         BAR,         PANdran(RAN0,0),             PANpickran(0,1),      0,  0 }
MOVE  {   212, 253,   1,  32,                1,               0,         BAR,         PANdbox("Move"),             BARmoveparent(),      0,  0 }
RAN1  {   254, 360,   1,  32,                1,               1,         BAR,         PANdran(RAN1,1),             PANpickran(1,1),      0,  0 }
BAR0  {   104, 360,  38,  50,                2,               1,         BAR,         PANdbar(BAR0,VAL0),          PANupbar(0),          0,  0 }
VAL0  {     2, 102,  34,  55,                1,               1,         BAR,         PANdval(0),                  PANpickval(0),        0,  0 }
OK    {     1, 120,   1,  32,                1,               0,         POP,         PANdbox("Ok"),               BARok(),              0,  0 }
SET   {     1, 120,  33,  65,                1,               0,         POP,         PANdbox("Reset"),            BARrset(),            0,  0 }
RSET  {     1, 120,  66,  98,                1,               0,         POP,         PANdbox("Reset Range"),      BARranset(),          0,  0 }
BAR   {     0, 361,   0,  57,                0,               0,         0,           PANdpic(0),                  0,                    0,  0 }
POP   {     0, 121,   0,  99,                0,               0,         BAR,         PANdpic(0),                  0,                    0,  0 }
```

(b)

Figure 3.
Two interfaces (a) and their specification using an "input table" (b) in Panther [Helfman 87].

aborting, and prompting.

Some of these systems, including Menulay [Buxton 83], Trillium [Henderson 86], and RAPID [Freburger 87], organize the user interface as a hierarchy or network of mostly static "pages" or "frames." Each page contains interaction techniques and text that appear together, as well as commands that cause the system to erase the page and go to different pages. The interaction techniques themselves usually must have been previously coded by hand in a conventional programming language.

Menulay [Buxton 83] allows the designer to place text, graphical potentiometers, iconic pictures, and light buttons on the screen and see exactly what the end user will see when the application is run. The designer does not need to be a programmer to use Menulay. Each active item in the display is associated with a semantic routine which is invoked when the user selects that item with the pointing device. Like virtually all other UIMSs, the semantic routines are written in a conventional programming language. Menulay generates tables and code which link to its run-time support package that executes the user interface for the application.

Menulay generated its own user interface and it supports multiple input devices operating concurrently. However, it has a rigid table-driven structure, so the interaction between the semantic level and the user interface is limited. This prevents all forms of semantic feedback.

Trillium [Henderson 86], which is aimed at designing the user interface panels for photocopiers, is very similar to Menulay. One strong advantage that Trillium has over Menulay is that the frames can be executed immediately as they are designed since the specification is interpreted rather than compiled. Trillium also separates the behavior of interactions from the graphic presentation and allows the designer to change the graphics (while keeping the same behavior) without programming. One weakness is that it has little support for frame-to-frame transitions, since this rarely is necessary for photocopiers.

RAPID (RApid Prototyper for Interface Design) [Freburger 87] is similar to Trillium. It is designed to allow non-programmer design engineers to prototype control panels for residential and office products at Honeywell, such as security systems and thermostats.

The GRINS (GRaphical INteraction System) UIMS [Olsen 85] combines a grammar processor (an "Interactive Push-Down Automaton") with a constraint-based "input-output linkage" system to handle semantic feedback. It incorporates a graphical editor that allows the interaction techniques (menus, icons, and text areas) to be placed using a mouse.

RAPID/USE [Wasserman 82] (discussed in section 5.1.2) also allows interaction techniques to be placed using a mouse. Nevertheless, it supports only limited mouse-based interfaces.

Peridot [Myers 87a] [Myers 87b], which stands for Programming by Example for Real-time Interface Design Obviating Typing, is very different from the systems discussed above in that it allows the interaction techniques themselves to be created. The primitives that the designer manipulates with the mouse are rectangles, circles, text, and lines. Out of these, menus, scroll bars, sliders, light buttons, etc. can be constructed. The system generalizes from the designer's actions to create parameterized, object-oriented procedures like those that might be found in interaction technique tool kits. Figure 4 shows some snapshots during the creation of a menu using Peridot. The system created its own user interface and can also create most of the interaction techniques in the Macintosh Toolbox, so it is a very powerful system. Peridot can also be used by non-programmers.

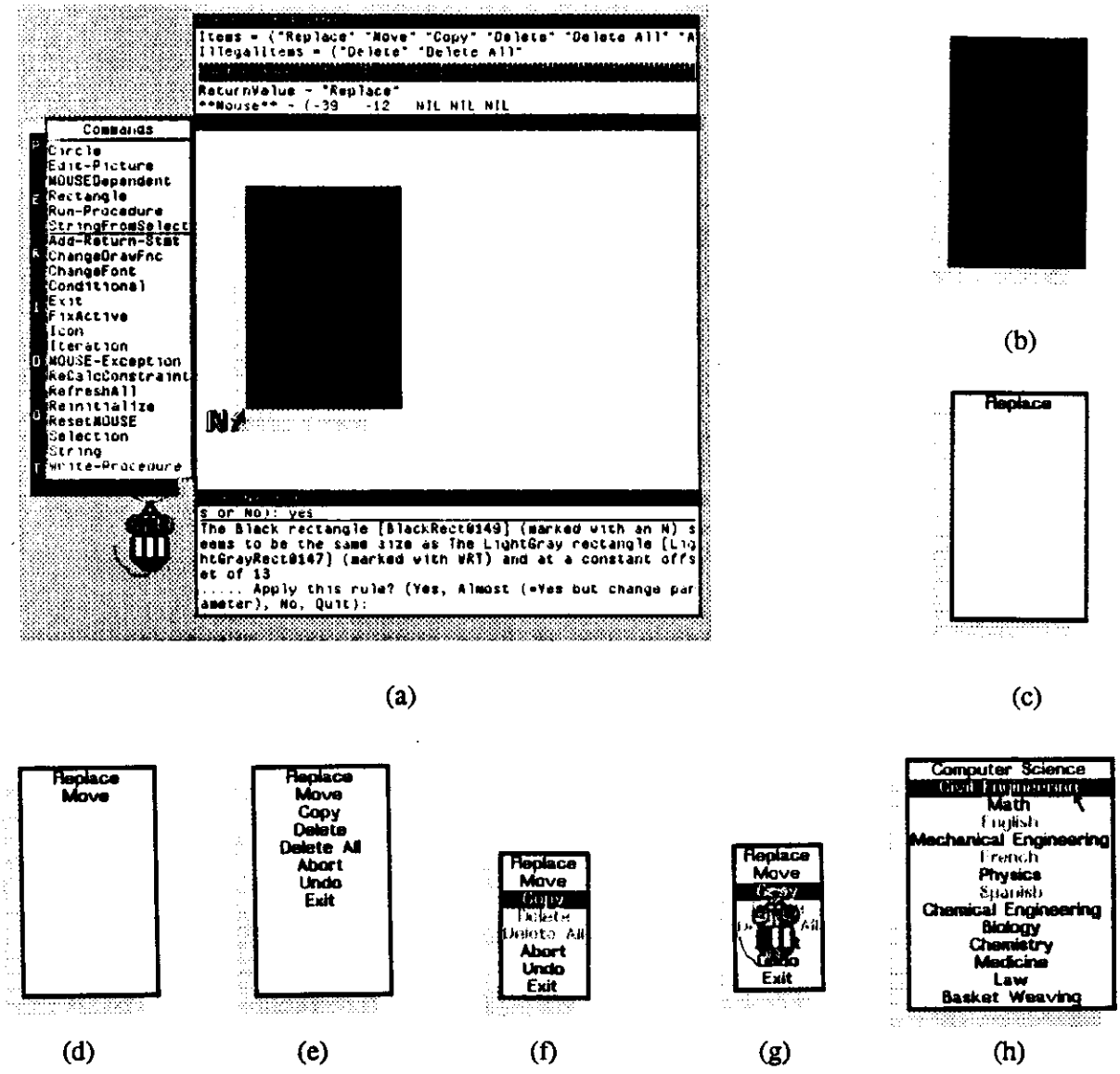


Figure 4.

A sequence of snapshots during the creation of a pop-up menu using Peridot [Myers 87b]. (a) shows the Peridot windows and command menu. The upper window contains the names of the parameters and active values used by "PopMenu" and example values for each. The center window contains the interface being created, and the bottom window contains the prompts and error messages from Peridot. In (a), a gray rectangle for the shadow has been drawn, followed by a black rectangle. Peridot has inferred that the black rectangle should be the same size as the gray rectangle but in a different place. The designer confirms that this guess is correct, and the black rectangle is adjusted to be exactly the size of the gray one (b). Similarly, in (c), a white rectangle has been drawn in the black one, and a string centered at the top of the white rectangle. The designer drew these in approximately the correct position and Peridot adjusted them to be exact. The string is the first element of the parameter "Items." The designer next places the second element of "Items" centered beneath the first one (d). Peridot now infers that an iteration is desired, and calculates how to display the rest of the elements in a similar manner (e). Next, the designer modifies the rectangle to be the size of the strings (f) and declares that the parameter "IllegalItems" controls which elements should be gray. The XORed black rectangle should follow the mouse, so the designer moves the mouse icon over it with its left button down (g). Now the menu is complete, and it can be used with a different set of parameter values (h).

5.3. Automatic Creation

A new class of UIMSs attempts to create the user interface directly from a specification of the application semantic procedures, and then allows the designer to modify the interface to improve it. One motivation for this is the difficulty people have using other types of UIMSs.

The Control Panel Interface [Fisher 87] uses the types of the parameters of a procedure to create a graphical interface using buttons for booleans, knobs or bars for integers, etc. (see Figure 5). The designer can specify different displays and change the values using the controls, and then execute the procedure.

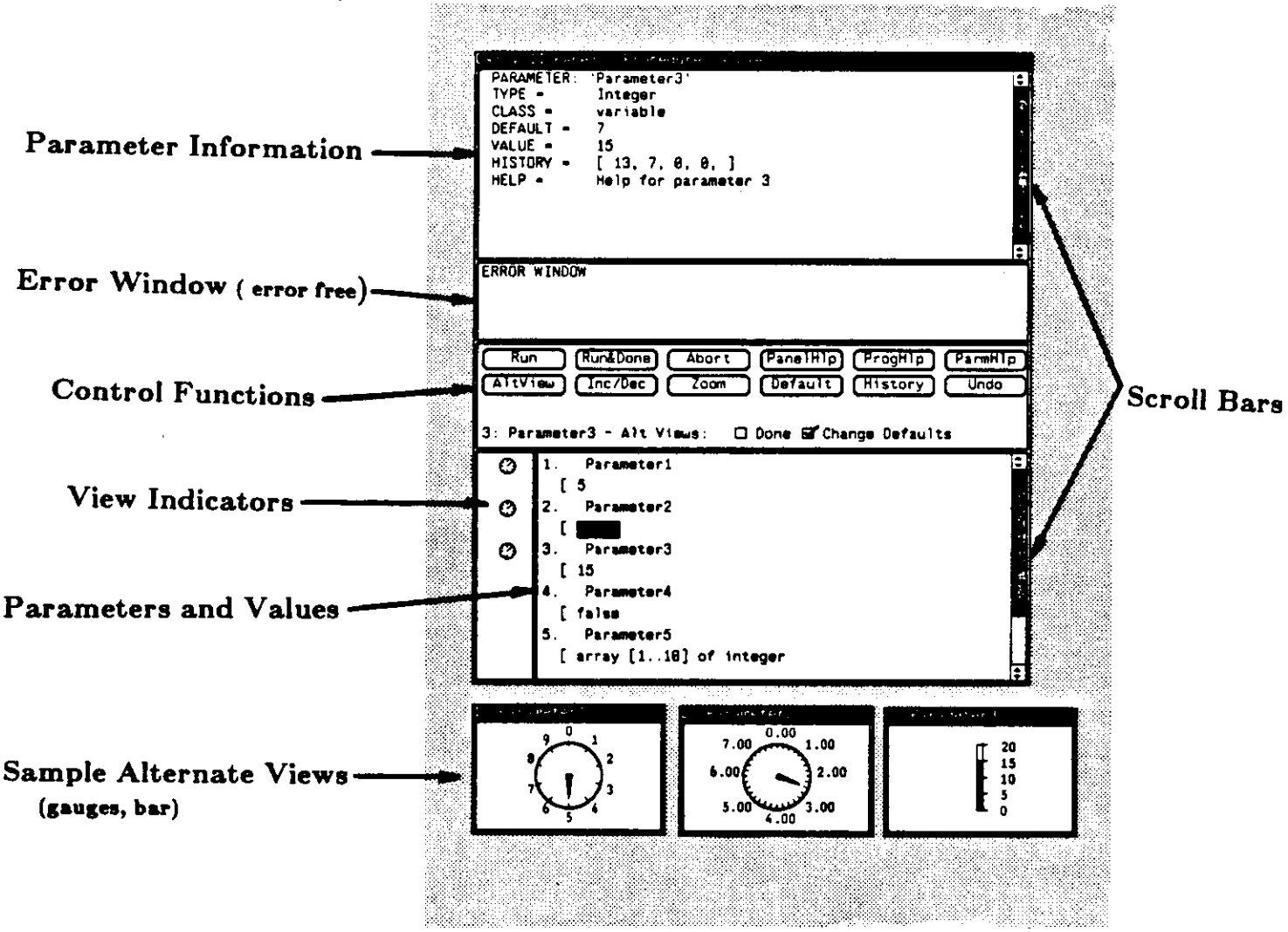


Figure 5.
A Control Panel Interface [Fisher 87] automatically created for a procedure. Three of the parameters are represented graphically in the bottom three windows.

MIKE, the Menu Interaction Kontrol Environment [Olsen 86], creates a user interface based on a list of the application procedures. The initial default interface is menu-oriented and rather verbose, but the designer can change the menu structure, use icons for some commands,

and even make some commands operate by Direct Manipulation. The designer uses a graphical editor, like those described in section 5.2, to specify these changes. Current research on MIKE is aimed at adding automatic user interface evaluation capabilities.

IDL (Interface Definition Language) [Foley 87] requires that the semantics of the application be defined in a special-purpose, Pascal-like language, and therefore might be included in section 5.1. It is placed here because the language is used to describe the functions that the application supports and not the desired interface. The system automatically generates a menu-based interface from that description. The interesting part of this system is that the user interface designer can apply "transformations" to the interface. These change the interface in various ways. For example, one transformation changes the interface to have a currently selected object instead of requiring that an object be selected for each operation. Another transformation provides functions specialized for various types ("delete-square" and "delete-triangle"), rather than general-purpose functions ("delete"). IDL applies the transformations and insures that the resulting interface remains consistent.

6. Communication

Another way to classify user interface tools is by how they communicate with application programs at run-time. With "Application Control" (also called "Internal Control"), the application simply calls user interface procedures when input is desired. This is the model used by user interface tool kits. With "UIMS control" (also called "External Control"), the user interface procedures call the application when the user gives a command. This is the model used by most UIMSs, and it can be further classified by how the UIMS interfaces with the application. The most popular method is to use "call-back" procedures, where the application passes to the UIMS the names of procedures to call, and the UIMS calls those procedures when necessary. The application is therefore organized as a set of procedures that the user interface calls. Another possibility is to use shared memory, with each side either polling the data to check for changes, or automatically being notified of changes. One way to implement the latter is by using "active values" [Stefik 86], which notify relevant programs when their value changes. Other UIMSs use multi-process message passing or event handling mechanisms to communicate to applications. The final communication choice is "mixed" control where either party can be in charge.

An important issue with communication is how high the bandwidth is between the application and the user interface. An early model for UIMSs promoted a narrow connection so that the application would be more independent from the user interface (see Figure 6a). This model has been used by some UIMSs, such as Cousin, and provides "coarse grain control."

That is, the user interface and the application communicate only rarely, for example when the user has completely specified an action to be performed along with all of its arguments. Unfortunately, this model makes it difficult or impossible to provide "fine-grain control" which is needed for semantic feedback. Here, the application and the user interface often need to communicate frequently, such as once for each incremental mouse movement (up to 60 times a second). Typically, the amount of information passed each time is fairly small, however. Newer UIMS models, such as shown in Figure 6b, have tried to provide for this kind of feedback by sharing application data with the UIMS. Szekely [Szekely 87] discusses various ways to enable this information sharing and still provide the advantages of modularization, but much more research is needed in this area.

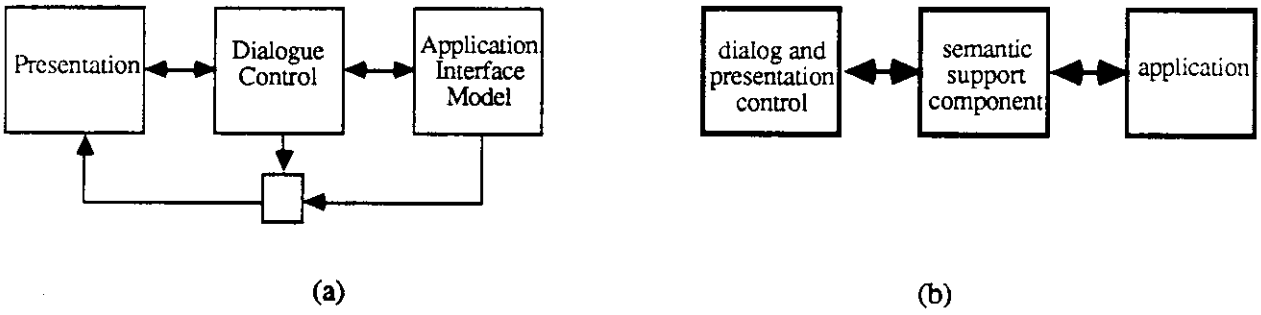


Figure 6.
(a) The "Seeheim" model for UIMSs [Green 85] has a narrow connection between the application and the dialogue control. (b) A newer model [Dance 87] proposes sharing the application information using a "Semantic Support Component" to support semantic feedback.

7. Problems with Existing User Interface Tools

The concept of user interface tools has gained general acceptance in the research community. This can be seen by the large number of papers on UIMSs that appear in conferences and magazines, and the frequent requests from businesses for tools to help build user interfaces. Although there is a rising use of user interface tool kits, very few commercial UIMSs have been used to any large extent [Myers 87c]. This section lists some problems with user interface tools in general.

(1) **Too difficult to use.** Tool kits typically contain hundreds of procedures which interact in subtle ways. Most UIMSs require that the designer learn a new special-purpose language in order to specify the interface. When the designers are programmers, as is usually the case, they are very reluctant to learn a new language to specify the interface [Olsen 87b]. Since these languages are usually like programming

languages, they are also inappropriate for professional user interface designers who are not programmers. The UIMSs which use direct graphical specification and automatic creation of user interfaces are addressing this problem, but these are still, for the most part, experimental.

(2) **Too little functionality.** Most UIMSs and tool kits only support a small part of the user interface design task. Whereas they are very good at handling menus and scrollbars, they rarely can be used to help control the display and manipulation of the real application data objects. Many of these tools do not make any attempt to handle output from the application program. For a Macintosh-like interface, this can be characterized as the tools controlling the "borders" of a window but not the "insides." In particular, few UIMSs can support Direct Manipulation interfaces. One reason for this is that Direct Manipulation interfaces often use semantic feedback, which is not supported by many UIMSs. Although they are easier to use, the direct graphical specification and automatic creation UIMSs typically have the most restricted functionality. The language approaches are usually more general, but still rarely address Direct Manipulation interfaces.

(3) **Not available and not portable.** Very few user interface tools are publicly available, and the ones that are available typically do not work on very many systems. For example, many people would like to use the Macintosh Toolbox and MacApp, but they cannot unless they are developing software to run on the Apple Macintosh. More portability for user interface software is just becoming possible with the advent of the X window manager and its tool kit, and UIMSs which run on it (such as Open Dialogue).

(4) **Lack of concrete evidence of their worth.** Although there is a lot of talk about the importance and difficulty of creating good user interfaces and how user interface tools will help, there are so few UIMSs in use that it is perceived as very risky to invest money in developing UIMSs. This is part of a general problem in convincing businesses that it is worth developing tools to increase programmer productivity. There are few conclusive studies that show how much money and time can be saved by using these tools. Clearly, problems (3) and (4) form a "chicken-and-egg" problem, since there will not be significant evidence of their worth until there are many UIMSs in use.

(5) **Difficulty with understanding and editing specifications.** Closely related to problem (2) is the issue of how to change the interface after it has been created. The specification languages used by many UIMSs to specify the user interface are poorly structured, in the Software Engineering sense: they use global variables, non-local

control flow, explicit gotos, etc. Some of the graphical techniques, such as state transition diagrams, are easy to understand when the user interface is small, but tend to become an incomprehensible maze of wires as the interface gets to be a non-trivial size. Consequently, with many UIMSs, understanding an interface specification can be very difficult.

(6) **Belief that the quality of the user interface will be *worse* if UI tools are used.** It is possible that the UI tools will significantly slow down the execution speed since often a layer of interpretation is added. The designers are also usually giving up the ability to hand-tune various aspects of the user interface, which may make the interface worse than one that was carefully crafted.

(7) **Unwillingness to give up control.** This is closely linked with problems (2) and (6). The problem here is that most user interface tools enforce a particular interface style on application programs. The advantage of this is that multiple applications will have similar user interfaces and the user interface designer does not have to worry about creating a style for the program. On the other hand, some designers are unwilling to give up this flexibility, because they want their products to have a unique "look and feel." This might be important for differentiating their company's products from the competitors'. A few experimental UIMSs, such as Peridot, attempt to be somewhat style independent, but this makes the design task more difficult. It is probably impossible for a user interface tool to be totally style independent if it provides more support than a conventional programming language.

(8) **Difficulty in building good user interface tools.** This is closely related to (4). There is clearly a great deal of research on how to make successful UIMSs and user interface tool kits. Unfortunately, no UIMS has been extremely successful. In addition, most of the tools that are useful have taken a large amount of effort to produce.

(9) **No support for evaluation.** Very few user interface tools provide any support for evaluating the interface after it has been created. Such tools might automatically evaluate the interface from the specification, or collect information that would be useful in evaluating the interface while it is being tested with users. Unfortunately, further research is necessary on how to perform the evaluation and what information should be saved, before this will be practical.

(10) **Difficulty in separating the "user interface" from the application.** One study [Rosson 87] reported that 50% of the user interface designers surveyed

> indicated that the user interface had not been considered distinct from the rest of the system during design; many seemed to have real difficulty in even imagining how such a separation might apply to the system they had designed, and a few made strong

statements about the inadvisability or impossibility of making such a distinction [Rosson 87, p. 140].

8. Conclusions

Although many of the user interface tools described here had only limited success, and there are still a large number of research issues to be addressed, the future for user interface tools is bright. The various tool kits are proving to be very popular and widely used, and a recent surge of new research in user interface management systems has produced a number of new approaches. It is certainly not clear at this point which approaches are the most fruitful for further research and commercial development, but it is clear that there is a rising need and demand for these tools so the pace of investigation is likely to accelerate.

ACKNOWLEDGEMENTS

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976 under contract F33615-87-C-1499 and monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, OH 45433-6543. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency of the US Government. For help and support with this paper, I would like to thank Dario Giuse, Doug Bunting, Pedro Szekely, Randy Pausch, Tom Cobourn, Len Bass, Stephan Greene, and Bernita Myers.

BIOGRAPHY OF Brad Myers

Brad A. Myers is a research computer scientist at Carnegie Mellon University. From 1980 until 1983, he worked at PERQ Systems Corporation. His research interests include User Interface Management Systems, user interfaces, Programming by Example, Visual Programming, interaction techniques, window management, programming environments, debugging, and graphics. Myers recently completed a PhD in computer science at the University of Toronto. He received the MS and BSc degrees from the Massachusetts Institute of Technology during which time he was a research intern at Xerox PARC. He is a member of SIGGRAPH, SIGCHI, ACM, and the IEEE Computer Society.

Myers's address is Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

REFERENCES

[Anderson 85] Nancy S. Anderson and Judith Reitman Olson, eds. *Methods for Designing Software to Fit Human Needs and Capabilities*, Proceedings of the Workshop on Software Human Factors. 1985: National Academy Press, Washington, D.C. 34 pages.

[Apple 85] Apple Computer, Inc. *Inside Macintosh*. Addison-Wesley, 1985.

[Barth 86] Paul S. Barth. "An Object-Oriented Approach to Graphical Interfaces," *ACM Transactions on Graphics*. Vol. 5, No. 2, April 1986. pp. 142-172,

[Bewley 83] William L. Bewley, Teresa L. Roberts, David Schroit, William L. Verplank. "Human Factors Testing in the Design of Xerox's 8010 'Star' Office Workstation," *Proceedings SIGCHI'83: Human Factors in Computing Systems*. Dec. 12-15, 1983. Boston, Mass. pp. 72-77.

[Bobrow 86] Daniel G. Bobrow, Sanjay Mittal, and Mark J. Stefik. "Expert Systems: Perils and Promise," *CACM*, Vol. 29, no. 9, September 1986. pp. 880-894.

[Boies 85] S.J. Boies, J.D. Gould, S. Levy, J.T. Richards, and J.W. Schoonard. "The 1984 Olympic Message System—A Case Study in System Design," *IBM Research Report.* RC-11138. 1985.

[Buxton 80] William A. S. Buxton and R. Sniderman. "Iteration in the Design of the Human-Computer Interface," *Proceedings: 13th Annual Meeting, Human Factors Association of Canada.* pp. 72-81.

[Buxton 83] W. Buxton, M.R. Lamb, D. Sherman, and K.C. Smith. "Towards a Comprehensive User Interface Management System," *Computer Graphics: SIGGRAPH'83 Conference Proceedings.* Detroit, Mich. Vol. 17, no. 3. July 25-29, 1983. pp. 35-42.

[Buxton 86] William Buxton and Brad Myers. "A Study in Two-Handed Input," *Proceedings SIGCHI'86: Human Factors in Computing Systems.* Boston, MA. April 13-17, 1986. pp. 321-326.

[Cardelli 85] Luca Cardelli and Rob Pike. "Squeak: A Language for Communicating with Mice," *Computer Graphics: SIGGRAPH'85 Conference Proceedings.* San Francisco, CA. Vol. 19, no. 3. July 22-26, 1985. pp. 199-204.

[Conklin 87] Jeff Conklin. "Hypertext: An Introduction and Survey," *IEEE Computer*, Vol. 20, No. 9, September 1987. pp. 17-41.

[Dance 87] John R. Dance, Tamar E. Granor, Ralph D. Hill, Scott E. Hudson, Jon Meads, Brad A. Myers, and Andrew Schulert. "Report on the Run-time Structure of UIMS-Supported Applications," *Computer Graphics*, Vol. 21, no. 2, April 1987. pp. 97-101.

[Fisher 87] Gene L. Fisher and Kenneth I. Joy. "A Control Panel Interface for Graphics and Image Processing Applications," *Proceedings SIGCHI+GI'87: Human Factors in Computing Systems.* Toronto, Ont., Canada. April 5-9, 1987. pp. 285-290.

[Flecchia 87] Mark A. Flecchia and R. Daniel Bergeron. "Specifying Complex Dialogs in ALGAE," *Proceedings SIGCHI+GI'87: Human Factors in Computing Systems.* Toronto, Ont., Canada. April 5-9, 1987. pp. 229-234.

[Foley 84] James D. Foley. "Managing the Design of User-Computer Interfaces," *Proceedings of the Fifth Annual NCGA Conference and Exposition.* Anaheim, CA. Vol. II. May 13-17, 1984. pp. 436-451.

[Foley 87] James Foley. "Transformations on a Formal Specification of User-Computer Interfaces," *Computer Graphics*, Vol. 21, no. 2, April 1987. pp. 109-112.

[Freburger 87] Karl Freburger. "RAPID: Prototyping Control Panel Interfaces," *OOPSLA'87 Conference Proceedings.* October 4-8, 1987, Orlando, Florida, Special issue of *SIGPLAN Notices*, Vol. 22, No. 12. December, 1987. pp. 416-422.

[Goldberg 83] Adele Goldberg and Dave Robson. *Smalltalk-80: The Language and Its Implementation.* Reading, MA: Addison-Wesley Publishing Company, 1983.

[Good 84] Michael D. Good, John A. Whiteside, Dennis R. Wilson, and Sandra J. Jones. "Building a User-Derived Interface," *CACM.* Vol. 27, no. 10. Oct. 1984. pp. 1032-1043.

[Green 85] Mark Green. "Report on Dialogue Specification Tools," *User Interface Management Systems*. Gunther E. Pfaff, ed. Berlin: Springer-Verlag, 1985. pp. 9-20.

[Green 86] Mark Green. "A Survey of Three Dialog Models," *ACM Transactions on Graphics*. Vol. 5, No. 3, July 1986. pp. 244-275.

[Hayes 85] Philip J. Hayes, Pedro A. Szekely, and Richard A. Lerner. "Design Alternatives for User Interface Management Systems Based on Experience with COUSIN," *Proceedings SIGCHI'85: Human Factors in Computing Systems*. San Francisco, CA. April 14-18, 1985. pp. 169-175.

[Helfman 87] Janathan Helfman. "Panther: A Tabular User-Interface Specification System," *Proceedings SIGCHI+GI'87: Human Factors in Computing Systems*. Toronto, Ont., Canada. April 5-9, 1987. pp. 279-284.

[Henderson 86] D. Austin Henderson, Jr. "The Trillium User Interface Design Environment," *Proceedings SIGCHI'86: Human Factors in Computing Systems*. Boston, MA. April 13-17, 1986. pp. 221-227.

[Hill 87a] Ralph D. Hill. "Event-Response Systems — A Technique for Specifying Multi-Threaded Dialogues," *Proceedings SIGCHI+GI'87: Human Factors in Computing Systems*. Toronto, Ont., Canada. April 5-9, 1987. pp. 241-248.

[Hill 87b] Ralph D. Hill. "Supporting Concurrency, Communication and Synchronization in Human-Computer Interaction—The Sassafras UIMS," *ACM Trans. on Graphics*, Volume 5, No. 3, July, 1986. pp. 179-210.

[Hudson 86] Scott E. Hudson and Roger King. "A Generator of Direct Manipulation Office Systems," *ACM Transactions on Office Systems*. Vol. 4, no. 2, April 1986. pp. 132-163.

[Jacob 85] Robert J.K. Jacob. "A State Transition Diagram Language for Visual Programming," *IEEE Computer*. Vol. 18, no. 8. Aug. 1985. pp. 51-59.

[Jacob 86] Robert J.K. Jacob. "A Specification Language for Direct Manipulation Interfaces," *ACM Transactions on Graphics*. Vol. 5, No. 4, October 1986. pp. 283-317.

[Kasik 82] David J. Kasik. "A User-Interface Management System," *Computer Graphics: SIGGRAPH'82 Conference Proceedings*. Boston, MA. Vol. 16, no. 3 . July 26-30, 1982. pp. 99-106.

[Mason 83] R.E.A. Mason and T.T. Carey. "Prototyping Interactive Information Systems," *Communications of the ACM*. Vol. 26, no. 5. May, 1983. pp. 347-354.

[Mittal 86] Sanjay Mittal, Clive L. Dym, and Mahesh Morjaria. "Pride: An Expert System for the Design of Paper Handling Systems," *IEEE Computer*. Vol. 19, no. 7, July, 1986. pp. 102-114.

[Morgan 83] C. Morgan, G. Williams, and P. Lemmons. "An Interview with Wayne Rosing, Bruce Daniels, and Larry Tesler," *Byte*. Vol. 8, no. 2, February, 1983. pp. 90-114.

[Myers 87a] Brad A. Myers. "Creating Interaction Techniques by Demonstration," *IEEE Computer Graphics and Applications*, Vol. 7, no. 9, Sept. 1987. pp. 51-60.

[Myers 87b] Brad A. Myers. *Creating User Interfaces by Demonstration*. PhD Dissertation, Department of Computer Science, University of Toronto. May, 1987.

[Myers 87c] Brad A. Myers. "Gaining General Acceptance for UIMSs," *Computer Graphics*, Vol. 21, no. 2, April 1987. pp. 130-134.

[Newman 68] William M. Newman. "A System for Interactive Graphical Programming," *Proceedings of the AFIPS Spring Joint Computer Conference.* 1968. pp. 47-54.

[Olsen 83] Dan R. Olsen, Jr. and Elizabeth P. Dempsey. "Syngraph: A Graphical User Interface Generator," *Computer Graphics: SIGGRAPH'83 Conference Proceedings.* Detroit, Mich. Vol. 17, no. 3. July 25-29, 1983. pp. 43-50.

[Olsen 84] Dan R. Olsen, Jr., William Buxton, Roger Ehrich, David J. Kasik, James R. Rhyne, and John Sibert. "A Context for User Interface Management," *IEEE Computer Graphics and Applications.* Vol. 4, no. 2. Dec. 1984. pp. 33-42.

[Olsen 85] Dan R. Olsen, Jr., Elisabeth P. Dempsey, and Roy Rogge. "Input-Output Linkage in a User Interface Management System," *Computer Graphics: SIGGRAPH'85 Conference Proceedings.* San Francisco, CA. Vol. 19, no. 3. July 22-26, 1985. pp. 225-234.

[Olsen 86] Dan R. Olsen, Jr. "Mike: The Menu Interaction Kontrol Environment," *ACM Transactions on Graphics.* Vol. 5, No. 4, October 1986. pp. 318-344.

[Olsen 87a] Dan R. Olsen, Jr., ed. "ACM SIGGRAPH Workshop on Software Tools for User Interface Management," *Computer Graphics*, Vol. 21, no. 2, April 1987. pp. 71-147.

[Olsen 87b] Dan R. Olsen, Jr. "Larger Issues in User Interface Management," *Computer Graphics*, Vol. 21, no. 2, April 1987. pp. 134-137.

[Pfaff 85] Gunther R. Pfaff, ed. *User Interface Management Systems.* Berlin: Springer-Verlag, 1985. 224 pages.

[Rosson 87] Mary Beth Rosson, Suzanne Maass, and Wendy A. Kellogg. "Designing for Designers: An Analysis of Design Practices in the Real World," *Proceedings SIGCHI+GI'87: Human Factors in Computing Systems.* Toronto, Ont., Canada. April 5-9, 1987. pp. 137-142.

[Scheifler 86] Robert W. Scheifler and Jim Gettys. "The X Window System," *ACM Transactions on Graphics*, Vol. 5, No. 2, April 1986. pp. 79-109.

[Schmucker 86] Kurt J. Schmucker. "MacApp: An Application Framework," *Byte*, August 1986. pp. 189-193.

[Schulert 85] Andrew J. Schulert, George T. Rogers, and James A. Hamilton. "ADM—A Dialogue Manager," *Proceedings SIGCHI'85: Human Factors in Computing Systems.* San Francisco, CA, April 14-18, 1985. pp. 177-183.

[Sheil 83] Beau Sheil. "Power Tools for Programmers," *Datamation.* Vol. 29, no. 2. Feb. 1983. pp. 131-144.

[Shneiderman 83] Ben Shneiderman. "Direct Manipulation: A Step Beyond Programming Languages," *IEEE Computer.* Vol. 16, no. 8. Aug. 1983. pp. 57-69.

[Sibert 86] John L. Sibert, William D. Hurley, and Teresa W. Bleser. "An Object-Oriented User Interface Management System," *Computer Graphics: SIGGRAPH '86 Conference Proceedings.* Vol. 20, no. 4, August 18-22, 1986. Dallas, Texas. pp. 259-268.

[Smith 82] David Canfield Smith, Charles Irby, Ralph Kimball, Bill Verplank, and Erik Harslem. "Designing the Star User Interface," *Byte Magazine*, April 1982, pp. 242-282.

[Stefik 86] Mark Stefik, Daniel G. Bobrow, and Kenneth M. Kahn. "Integrating Access-Oriented Programming into a Multi-Paradigm Environment," *IEEE Software*. Vol. 3, no. 1, Jan, 1986. pp. 10-18.

[Sun 84] Sun Microsystems, Inc. *SunWindows Programmers' Guide*, January, 7, 1984. 2550 Garcia Avenue, Mountain View, CA 94043.

[Sutton 78] Jimmy A. Sutton and Ralph H. Sprague, Jr. *A Study of Display Generation and Management in Interactive Business Applications*. IBM Research Report RJ2392. Nov. 9, 1978. 20 pages.

[Swartout 82] W. Swartout and R. Balzer. "The Inevitable Intertwining of Specification and Implementation," *Communications of the ACM*. Vol. 25, no. 7. 1982. pp. 438-440.

[Szekely 87] Pedro Szekely. "Modular Implementation of Presentations," *Proceedings SIGCHI+GI'87: Human Factors in Computing Systems*. Toronto, Ont., Canada. April 5-9, 1987. pp. 235-240.

[Tesler 81] Larry Tesler, "The Smalltalk Environment," *Byte*. Vol. 6, No. 8, Aug., 1981, pp. 90-147.

[Thomas 83] James J. Thomas and Griffith Hamlin, eds. "Graphical Input Interaction Technique (GIIT) Workshop Summary." ACM/SIGGRAPH, Seattle, WA. June 2-4, 1982. in *Computer Graphics*. Vol. 17, no. 1. Jan. 1983. pp. 5-30.

[Wasserman 82] A.I. Wasserman and D.T. Shewmake. "Rapid Prototyping of Interactive Information Systems," *ACM Software Engineering Notes*. Vol. 7, no. 5. pp. 171-180.

[Williams 83] Gregg Williams. "The Lisa Computer System," *Byte Magazine*, February 1983, pp. 33-50.