THE ADAPTABLE USER INTERFACE

by

E. Kantorowitz and O. Sudarsky

Technical Report #526
November 1988
The Adaptable User Interface

Eliezer KANTOROWITZ and Oded SUDARSKY

Technion—Israel Institute of Technology

ABSTRACT

Most user interfaces are based on a single method of communication between the user and the system (a single “dialogue mode”). Some systems provide several user interfaces, e.g. a menu interface for novice users and a command language for experienced users. We propose a single adaptable user interface (AUI) which allows the user to switch between dialogue modes at any time—even in the middle of a command. This is useful when the user is partly familiar with the system, and when different dialogue modes are appropriate for the various parameters of a single command. The freedom to employ the most suitable dialogue mode is expected to contribute to user productivity and satisfaction. A user interface management system (UIMS) was implemented in order to test the practicality of AUIs and their automatic generation.

This research was supported by the Gutwirth Fellowship-Fund and by the Nicholas M. and Gisela Deutch Fellowship in Memory of Their Martyred Parents and Families.
INTRODUCTION: THE AUI CONCEPT

Software systems interact with users in a large variety of ways (dialogue mode). These methods may be classified into menu-type and command-language-type. In a menu-type dialogue mode (MM), the user controls the system solely through the selection of options from a number of choices presented to him. It is assumed that only choices that make sense are presented to the user, and that it is therefore not possible to select illegal options. In a command-language-type dialogue mode (CLM), the user controls the system by instructions given in a certain command language. The user must know this language in order to use the system. In contrast with an MM, where the user is guided by menus and can only enter legal choices, the user of a CLM may enter erroneous instructions. A CLM should, therefore, be designed to detect ill-formulated instructions and to recover correctly from their effects.

The advantages of an MM over a CLM are well known. Since the user does not have to learn any command language, he may become productive with a new system after a very short time. A new user may explore the operations provided by a system simply by browsing through the menus. If such a system also provides adequate “help” for every menu option, the user may operate the system without ever needing a manual.

Menu systems may, however, be less satisfactory for frequent users, who have to work through a large number of menus to get their work done. Waiting for a menu to appear on the screen, finding the right menu entry and making the selection take some time. Proficient users who do not need guidance therefore tend to prefer a concise CLM, where a few key strokes, entered at full typing speed, achieve the same effect as a number of relatively slow menu selections.

“Knowledgeable users often remark that they would prefer to type commands and believe that they can work more rapidly by just typing commands” [9].

Realizing the different needs of beginners and experts, some systems (e.g. [1], [2], [7], [11]) provide two distinct user interfaces, a menu interface and a command-language interface. In these systems, the user has to select in advance (in the beginning of the session or before each command) the one of these two interfaces that suits him best. If a user wants to change the dialogue mode, he must first complete the current command, and then request the system to switch to the desired user interface. However, there are
cases when it is useful to change the dialogue mode in the middle of a command. To demonstrate this, we will employ an example command for drawing a black box with the lower left corner at (0,0) and the upper right corner at (1,1):

BOX BLACK 0 0 1 1

The first situation to be discussed is when a user realizes that he has forgotten some command language element while he is in the middle of composing a command, and may need the assistance of menus [5], [7]. Assume that the user has typed the word BOX of the example command, and is uncertain about what colors are available. He may wish to shift to an MM in order to be able select the color from a menu. A similar situation occurs when the user has made a mistake, e.g. entered a color that does not exist. After reading the error message, the user may wish to switch to the color menu in order to make a legal choice.

Another criterion for deciding whether to use a CLM or an MM is the nature of the input data and the physical properties of the I/O devices. In the above example, the coordinates of the corners are known by their exact numerical values ((0,0) and (1,1)), and it is therefore appropriate to enter them by typing them at the keyboard (a CLM). However, if a corner is only known by its position on the screen, it is must be entered by pointing with a mouse or another locating device. By our definition, this method for entering coordinates is an MM, where the points of the screen are the choices and the selection is made with the mouse. We note that depending on the nature of the actual input data, the same command is sometimes entered in a CLM and in other cases in an MM. Sometimes it is required to employ two different dialogue mode within the same command. For instance, one of the corners of the box is known by its position on the screen (and must therefore be entered by pointing at it), while the other corner is known by the numerical values of its coordinates (and should be entered through the keyboard).

It is sometimes useful to have more than one MM or more than one CLM. As an example, let us consider a system with two different CLMs for the same command language: a voice input mode and a keyboard input mode. Voice input is preferred when the user has to operate away from the terminal or when his hands are occupied. On the other hand, keyboard input may be quicker or less error-prone in some situations.
We propose an adaptable user interface (AUI), in order to allow the user to switch dialogue modes in the middle of a command. An adaptable user interface is defined as a user interface which:

- supports a number of different dialogue modes. More than two modes may be provided;
- allows the user to switch between dialogue modes at any point of time, i.e. even in the middle of a command;
- makes the switch between dialogue modes smooth and natural;
- makes it easy for the user to learn how to use the different dialogue modes, especially the CLMs, which usually require a longer training period.

In order to enable simple and natural switching between dialogue modes, a number of assumptions and requirements are proposed. The central assumption is that all the dialogue modes of an AUI are different representations of a single underlying dialogue language. This common language is assumed to be constructed of a number of elementary syntactic components, to be called tokens. Every token is required to have a distinct representation in each of the dialogue modes. In an MM, a token is represented by a single menu selection, while the corresponding representation in a CLM is an "atom" of the command language. For example, in the Guide system, to be described later, CLM tokens are represented by character strings. Each token may be entered in any one of the available dialogue modes, independently of the modes employed for the other tokens. Two subsequent tokens may thus be entered in two different modes.

Beginners and casual users will employ the AUI in an MM. As they become more familiar with the system, they will gradually learn the CLM instructions that they need. A user can exploit the CLM commands that he has already learned, and employ an MM for all the other commands. Users will not have to learn CLM commands that are rarely used, since they may be entered in an MM.

Additionally, each token can be entered in the most suitable way. For example, in the BOX command, one corner of the box may be entered using a mouse, while the opposite corner can be entered by typing its coordinates.
The implementation of a user interface is usually a major effort. This is especially true for an AUI, in which several input devices must be monitored simultaneously. It is therefore desirable to have a user interface management system (UIMS) [12], [13], [17] which automatically generates AUIs. However, none of the existing UIMSs allows the easy production of AUIs. Most systems can only generate single-dialogue-mode user interfaces. In systems which offer several modes, the end user is usually required to select a single dialogue mode at the beginning of the session. The Workspaces system [3] allows only partial adaptability (keyboard parameters must be entered first, followed by the parameters given by other input devices). The IOT [18], Switchboard [16] and Sassafras [8] systems may possibly be extended by the user interface designer with code that supports several dialogue modes; However, writing such code is a difficult task which requires insight in parallel programming of I/O devices.

THE GUIDE SYSTEM

In order to test the practicality of AUIs and of a UIMS which generates AUIs, the GUIDE (Graphic User Interface Design Environment) system was implemented [15]. In this section we will define the terms used in GUIDE, describe the structure of applications developed with GUIDE and the tools which GUIDE provides to construct user interfaces.

GUIDE considers an application program as a system for the processing of objects. Every object is a data item of the application. (It is not necessarily an object in the sense of object-oriented programming.) The set of all data items of the same kind is called an object type; Every object in the type is also called an instance of the type. Every object is graphically represented on the screen, and is uniquely identified by a character string called the name of the object. When the end user points at an object's graphic representation or types the object's name, that object is selected for processing by the application.

The application program is controlled by graphic menus. Every menu option has a rectangular region on the display, and is uniquely identified within the menu by a name. When the end user points inside the rectangular region of an option or types the option's name at the keyboard, the option is selected. The selections of menu options and of object instances are the atomic input actions (the tokens) of GUIDE.
Structure of a GUIDE-Developed Application

An application program developed with GUIDE has three main modules, called the lexical, syntactic and semantic components. The lexical component identifies the tokens in the stream of input events. It also displays the objects and menus. The syntactic component analyses the stream of tokens it receives from the lexical component, and invokes the semantic component when required. The semantic component is a collection of application routines, written by an application programmer in an "ordinary" programming language. These routines perform creation, deletion, updating and processing of application objects.

The syntactic and lexical components constitute the user interface of the application program. GUIDE generates the user interface from specifications given by the designer. These specifications are entered through interactive graphic design tools, which do not require any programming knowledge. The user interface designer may therefore be a human factor expert who is not a programmer. The user interface specifications do not cause the generation of any code; Rather, they are stored in a database, and are then interpreted by a run-time environment. The code of this run-time environment is identical for all GUIDE-developed applications; Only the database and the semantic component are different. A change in the user interface specifications only requires a modification of the database. The effect of such a change can be seen immediately, since no compilation and linkage are required. This facilitates rapid prototyping of user interfaces, since the designer can try several alternative solutions within a short time.

The Syntactic Component

The syntactic component of the user interface employs a recursive transition network (RTN) as the definition of the dialogue language. An RTN interpreter executes this definition when it analyses the stream of input tokens.

RTNs were chosen for the syntax representation because they are as powerful as deterministic, context-free grammars, yet easier to use than BNF representation, especially for non-programmers [4], [6], [10]. One of the design goals of GUIDE was to let the user interface be designed by human factor experts who have no programming knowledge.
An RTN is a collection of subnets. Each subnet is represented by a directed graph. The following kinds of states (nodes) and transitions (edges) may be appear in a subnet (see Fig. 1):

- **initial state**—the state in which the execution of the subnet begins;
- **return state**—causes control to return to the calling subnet;
- **subnet call state**—causes control to pass to another subnet (or, recursively, to the same subnet);
- **application call state**—causes an application routine to be executed;
- **input state**—causes control to wait for the reception of a token from the lexical component. A menu may be associated with this state; If required, this menu will be displayed automatically when this state is reached;
- **output state**—causes the display of a message to the end user;
- **plain transition**;
- **return transition**—appears after an application call state, and is traversed if the routine has returned the return code associated with this transition;
- **option transition**—appears after an input state, and is traversed if the user has selected the menu option associated with this transition;
- **parameter transition**—appears after an input state, and is traversed if the user has picked an object of the type associated with this transition.

Fig. 2 shows an example of an RTN subnet called **PickNode**. The execution of this subnet starts at the initial state S1, and then immediately passes to the input state S2. A menu called **NodeMenu** is associated with this state; The user has to select one of the two options in this menu. If, for instance, the option named **Del** is selected, the option transition T2 will be traversed, and the application call state S3 will be reached. The semantic procedure DNode associated with this state will then be executed. Finally, the return state S5 will be encountered, and the execution of the subnet will terminate.
GUIDE provides an interactive graphic editor called SynEdrr. This editor allows the user interface designer to construct the RTN which defines the syntax of the dialogue language. An actual SynEdrr screen is shown in Fig. 2. SynEdrr checks the consistency and completeness of the RTN to make sure that it can be executed. The user interface for SynEdrr itself was generated by GUIDE.

The Lexical Component

When the RTN interpreter (in the syntactic component) encounters an output state, it calls a lexical routine to display the required message. When an input state is reached, the interpreter calls another lexical routine. This routine displays a menu if required, and waits for user input. When the user selects a menu option or an object instance, the lexical routine will return a corresponding token to the syntactic component. The RTN interpreter will then traverse the transition which corresponds to the selected option or object type.

GUIDE's lexical component currently supports two dialogue modes. Every menu option and every object instance can be picked in one of two ways: by pointing at the option or object with a mouse (an MM), or by typing the option's or object's name (a CLM). Text typed by the user at the keyboard appears in a special CLM text area at the bottom of the screen (see Fig. 3). If the user employs the mouse, the name...
Fig. 2: An RTN subnet being edited by SynEdit

of the selected option or object will be copied by Guide into the CLM area, as if the user has typed them in. The CLM area thus shows the command-language representation of the command being entered. This helps the user to learn the command language. Note again that each of the tokens in the command may be entered in a different dialogue mode. Furthermore, note that the user does not have to tell the system to switch between dialogue modes—he simply uses the device (mouse or keyboard) he wishes to employ.

The different dialogue modes are managed solely by the lexical component. When the RTN interpreter receives a token from the lexical component, it has no knowledge of the mode in which this token was entered. This makes the system relatively easy to adapt to future dialogue modes and input devices (e.g. a speech recognizer), since only the lexical component will have to be changed, while the syntactic and semantic components will remain unchanged.
GUIDE represents application objects by pictures called *graphic objects*. All graphic objects of the same type appear identical, with the possible exception of the contents of some text strings. These strings are called the *fields* of the graphic object. Typically, fields are used to provide a verbal description of the objects and to identify the object instances (i.e. one of the fields is usually the object’s name).

Two kinds of graphic objects are provided: icons and links. A *link* is a line connecting two icons. The designer has some control over the appearance of links: The shape of a link may be straight, curved or rectilinear; The line style can be solid, dashed or dotted; An arrowhead may be shown at the line’s start or end; Fields and text strings may be placed along the line. Links are useful when the user interface is in the form of a graph with various kinds of nodes and edges: Icons can represent the nodes, while links represent the edges.

GUIDE provides a program called LexEdit, which allows the user interface designer to define icons, links, menus and messages. In the icon editor screen (see Fig. 4), the icon is drawn using lines, arcs, rectangles, ellipses, text strings and fields. The icon is shown twice on the screen: life-sized in the upper right-hand corner, and enlarged three times in the main area of the screen. In the link editor screen (see Fig. 5), the designer can specify the attributes of the link, and place fields and text strings along the link. In the

---

![Fig. 3: Screen layout for a GUIDE-developed application](image-url)
menu editor screen (see Fig. 6), the menu's graphic appearance is drawn. The name and the rectangular region occupied by each menu option can be defined. The designer has a choice of three menu styles: static, pop-up and pull-down. These allow the implementation of various currently popular user interface styles.

Fig. 4: LexEdit's icon editor screen

EXAMPLES OF GUIDE-DEVELOPED APPLICATIONS

In order to test the applicability of GUIDE in various areas, user interfaces for three different applications were constructed. The applications are: a directed graph editor, a specification program for management information systems (MISs), and the RTN editor of the GUIDE system.

The directed graph editor is a minimal application. It was implemented as a simple example for the GUIDE system's manual, to demonstrate the use of GUIDE to both user interface designers and application
Fig. 5: LexEddy's link editor screen

programmers. The program lets the end user edit a directed graph interactively (see Fig. 7). It has commands to add, delete and move nodes, and to add and delete directed edges between nodes. The entire user interface for this application was developed in less than two hours.

The second application is a program to assist system analysts in the development of design specifications for MISs. It was developed by D. Reider at the Computer Science Department at the Technion [14]. The program supports a top-down design methodology which combines data modeling and structured analysis. The system analyst has three tools at his disposal: an organizational structure diagram, an entity-relationship diagram (ERD) and data flow diagrams (DFDs). Each of these diagrams is composed of different kinds of icons and links. The design methodology requires that certain complex relationships are maintained between the diagrams. The system lets the user edit these diagrams. The consistency and completeness of each diagram and the relationships between the diagrams are checked automatically. The
diagrams are stepwise-refined, and are therefore changed frequently. The resulting diagrams can be quite complex, and would be difficult to manage manually. The user interface for this application was developed in about forty hours.

The third application is the RTN editor SynEdrr, which is employed in Guide to specify the syntax of dialogue languages (see the section The Syntactic Component and Fig. 2). The RTN states (shown in Fig. 1) are represented as icons, while the transitions are represented as links. The editor is employed to construct the subnets which constitute the RTN, and to check the correctness of the RTN. Since SynEdrr could not be used to enter the specifications of its own RTN, these data had to be entered manually into the specifications database. The rest of the specifications for SynEdrr were input with LexEdrr, as usual.
Fig. 7: The directed graph editor screen (shown with a pop-up menu)

IMPLEMENTATION

GUIDE was implemented on an IBM PC AT compatible computer, under the MS-DOS operating system. Most of the software was written in the dBASE III PLUS [1] relational database programming language, and compiled with the Clipper compiler. The graphics are displayed by a CGI (computer graphics interface standard) subroutine library. A few small routines were written in C language. These routines are used mostly for communication between the dBASE programs and the CGI library.

The dBASE language was chosen because the use of a relational database allowed rapid and simple development of GUIDE. This is primarily because in a relational database system, a record is referred to by its key, rather than by a pointer to the record (as done in conventional programming languages). This saves numerous record and pointer type checking and definitions. The database system also provides some other services, including automatic file handling.
The CGI graphics standard was employed in order to provide portability to different I/O devices. GUIDE and GUIDE-developed applications can work with different graphic displays and various input devices (mouse, light pen, arrow keys etc.). Porting the program to another device configuration does not involve any re-programming and re-Compilation.

A GUIDE-developed user interface consists of the run-time environment and of 15 database tables. The database contains the specification of the dialogue language syntax (the RTN) as well as the lexical specifications (the graphic appearance of menus, objects etc.). GUIDE's run-time environment occupies about 230 Kbytes of executable code. This number includes the Clipper compiler's run-time system, which occupies about 120 Kbytes. (The semantic routines are not a part of the user interface, and are not included in the above figures.)

Possible improvements to GUIDE include:

— providing context-sensitive help. This can be implemented in a simple way by associating help text with each RTN input state. When an input state is encountered, its help text may be displayed at the end user's request.

— implementation of a macro facility, which enables the end user to store an often-occurring sequence of tokens. The sequence can simply be recorded as a path in the RTN, and replayed by traversing this path.

— integration of LexEdef, SynEdef and the run-time environment into a single program. This integration will shorten the edit-try cycle time and thus allow more rapid prototyping of user interfaces.

— rewriting the entire GUIDE system in a general-purpose programming language, e.g. C language. The current version of GUIDE can only be executed on computer systems where dBASE is available. Furthermore, it is relatively slow, since the dBASE language cannot be fully compiled. Such a re-implementation is expected to make GUIDE sufficiently efficient for personal computers.
CONCLUSIONS

It has been demonstrated that it is sometimes useful to employ a number of different dialogue modes (such as typing text at the keyboard and pointing with a mouse) within the same command. This cannot be achieved satisfactorily in current systems, which provide a separate user interface for each dialogue mode. In order to provide a "natural" solution for the end user, this paper introduced the adaptable user interface (AUI) concept. It is assumed that a single dialogue language may be defined for all dialogue modes of each application program. It is further assumed that the dialogue language may be constructed from a number of abstract tokens, and that each token has a distinct representation in any one of the provided dialogue modes. In an MM (menu mode), a token corresponds to a single menu selection, while in a CLM (command language mode), a token is represented by a command language "atom." An AUI may then be constructed such that each token can be entered by the user in any one of the supported dialogue modes, independently of the modes employed for the other tokens. The user can therefore employ the dialogue mode that he feels is most suitable for every token.

The freedom of the AUI user to adapt dialogue modes to his actual needs is useful at all levels of experience: A novice user starts by using menu modes (MMs), where he is guided by choices presented to him by the system, and gradually learns how to utilize the command language; An experienced user employs the faster command language modes (CLMs), but can switch to menus in the middle of a command if he is uncertain about how to finish it. An AUI also enables the user to choose the dialogue mode which is most suitable for the kind of input data and the working environment. For example, the coordinates of a point known by its screen position may be entered by pointing at it with a mouse, while the keyboard may be employed when the exact numeric values of the coordinates are known. Note that it is the user that decides, at data entry time, which dialogue mode is most appropriate at that particular situation. A further advantage of having a single underlying language for all dialogue modes is that this facilitates the learning of the system. In order to become proficient, a user has only to learn the syntax of the single dialogue language and the few methods employed for entering tokens in the different dialogue modes.

The desirability of a UIMS which automatically generates AUIS was discussed, and the implementation of such a system, Guide, was described. User interfaces generated by Guide are AUIS which give the
end user a choice of two different dialogue modes: keyboard input (a CLM) and selection by a mouse (an MM). The user can switch between these modes simply by using the preferred input device, i.e. the keyboard or the mouse. He does not have to tell the system in advance which mode he is going to use.

Whenever the user enters a token (in either MM or CLM), Guide copies the command language representation of the token into a special CLM area at the bottom of the screen. This area thus shows the part of the command that has already been entered. Novice users can see in this area the CLM equivalents of their MM inputs, and thereby learn to use the CLM.

In order to generate a user interface with Guide, its specifications are entered through two interactive graphic editors, LexEdit and SynEdit. The validity of these specifications is automatically checked by the system. The editors are simple to operate, and do not require any knowledge of programming. The user interface may therefore be designed by an expert in man-machine interaction who has no programming experience.

Guide stores all user interface specifications in a database. Rather than generating code according to these definitions, a run-time environment interprets the stored specifications. Since changing the specifications does not involve compilation or linkage, the edit-try cycle time is short. This allows rapid prototyping of user interfaces.

A Guide-generated user interface consists of a lexical component and a syntactic component. The dialogue modes are managed entirely by the lexical component. The syntactic component receives abstract input tokens from the lexical component, and is not concerned with the way in which they were entered. This simplifies the design of the syntactic component. Furthermore, adding new dialogue modes and adapting to new I/O devices only requires a modification of the lexical component.

The implementation of the Guide system suggests that both AUIs and a UIMS which generates AUIs are practical on personal computers. The storage requirements of Guide are reasonable, but the execution speed in our experimental system needs some enhancement. However, the proposed re-implementation in a compilable programming language is expected to provide adequate performance.

Experience with Guide seems to confirm the expectations. User interfaces for three different applications were implemented relatively quickly and conveniently. With these applications, switching between
different dialogue modes appears natural and gives the user a new degree of freedom in exploiting the system. In conclusion, it is felt that the ability to employ the most suitable dialogue mode at any point of time results in higher user productivity and satisfaction.

ACKNOWLEDGMENT

We thank David Reider for his kind help in system analysis and in the development of one of the examples.

REFERENCES


