The University of Alberta User Interface Management System

Mark Green
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada

ABSTRACT: In this paper the design and implementation of the University of Alberta user interface management system (UIMS) is discussed. This UIMS is based on the Seeheim model of user interfaces, which divides the user interface into three separate components. The Seeheim model of user interfaces is discussed along with its relationship to the design of UIMSs. The techniques used to design the three user interface components are briefly presented. A mixture of interactive and written notations are used in the design of the user interface. Some interesting features of this UIMS are interactive screen and menu layout, support for three dialogue notations, flexible interface to the application program, ability to adapt to different users, and the use of concurrent processes in user interface implementation The techniques used in the implementation of this UIMS are discussed.

KEYWORDS: user interface design, user interface management systems, human-computer interaction

1. Introduction

The user interface is the component of a computer system that stands between the user and the rest of the system. Good software engineering practice suggests that the user interface should be a separate program module. All interactions between the user and the program are handled by the user interface module (in this paper the term user interface will usually mean the user interface module that implements it). A separate user interface module naturally leads to the notion of a User Interface Management System (UIMS). A UIMS facilitates the design, construction, and maintenance of user interfaces. A good introduction to current research on UIMSs can be found in the reports of the Graphical Input Interaction Technique workshop sponsored by SIGGRAPH [17] and the Seeheim Workshop on User Interface Management Systems sponsored by Eurographics and IFIPS [18].

In this paper we discuss the design and implementation of the University of Alberta UIMS (for lack of a better name). This UIMS is based on the Seeheim model of user interfaces that was developed at the Seeheim workshop. This model is presented in section 2 of this paper. Some of the user interface design tools provided by the University of Alberta UIMS are briefly described in section 3. The set of design tools is fairly extensive and cannot be properly described in one paper. The fourth section describes the run-time structure of the user interfaces produced by this UIMS. This section presents a general implementation strategy for user interfaces based on the Seeheim model. The last section summarizes this work and provides suggestions for further research.

There were a number of reasons for developing the UIMS presented in this paper. First, we wanted to evaluate the feasibility of the Seeheim model as the basis for UIMSs. When this model was proposed it had not been used as the basis of a user interface or a UIMS, so there was no way of evaluating it. By basing our UIMS on the Seeheim model we have some evidence for its usability. Second, we wanted a test bed for our ideas on user interface design and implementation. We wanted a way of testing our ideas without building a complete user interface or UIMS. Third, we wanted a practical tool that can be used in other research projects within our department. The last two goals are to some degree contradictory. The last goal implies that the UIMS should be relatively stable so other users have a solid foundation to build on. On the other hand the second goal requires the UIMS to be relatively easy to modify. This question is discussed further in section 5.

2. The Seeheim Model of User Interfaces

In this section we briefly describe the Seeheim model of user interfaces, a more detailed description of this model is presented in [6]. This model was developed at the Seeheim Workshop on User Interface Management Systems by a working group whose members were: Jan Derksen, Ernest Edmonds, Mark Green, Dan Olsen, and Robert Spence. The Seeheim model is based on dividing the user interface into three components as shown in fig. 1. The presentation component is responsible for the physical appearance of the user interface including all the device interactions. The dialogue control component manages the dialogue between the user and the program. The application interface model forms the interface between the user interface and the rest of the program. It is the user interface's view of the application program.

The information flowing between the components is in the form of tokens. Each token consists of a type field, which identifies the token, and a number of data fields that depend upon the type of the token. This abstract representation is independent of the devices used by the user interface. The only component of the user interface that must deal with the details of devices is the presentation component. An input token is a token moving from the user towards the application and an output token is moving from the application towards the user.

2.1. Presentation Component

The presentation component can be viewed as the lexical level of the user interface. It is responsible for screen management, information display, input devices, interaction techniques and lexical feedback. The menus in an application are part of the presentation component. When the user selects an item from a menu the presentation component generates an input token that is sent to the dialogue control component. If multiple menus are used dialogue control sends output tokens to the presentation component indicating when the menus should be active. The presentation component guarantees that the user can always select from any of the active menus, but beyond this dialogue control has no control over menu appearance.

There are a number of advantages to having a separate presentation component. First, all the device interactions are isolated in this component. This increases the portability of the user interface since only the presentation component needs to be changed when the user interface is moved to a different display device. The presentation component can be designed to support a range of display devices and automatically adapt to the one being used. This is easier to do when the device interactions are isolated in one component. Second, a separate presentation component provides a convenient means of tailoring the lexical level of the user interface to individual users. The screen layout can be changed to accommodate both left and right handed users, default command options can be changed, and the user can select his favorite interaction or display technique for a particular type of data. Third, a separate presentation component encourages the development and use of a standard library of interaction techniques. This will reduce the cost of user interfaces and improve their quality.

2.2. Dialogue Control Component

The dialogue control component manages the dialogue between the user and the application. This component converts the stream of input tokens originating in the presentation component into a structure representing the commands and operands intended by the user. This structure is then converted into a sequence of input tokens sent to the application interface model in order to execute the command. Similarly the output tokens sent by the application interface model are interpreted by dialogue control and a sequence of output tokens for the presentation component is generated.

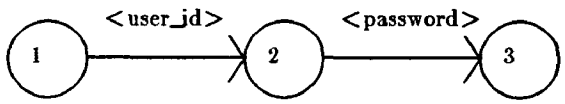
Most existing UIMSs have concentrated on the dialogue control component, therefore, we have more experience with it than the other components. There are three main notations for the dialogue between the user and computer. These notations are recursive transition networks, BNF grammars, and events.

2.2.1. Recursive Transition Networks

A recursive transition network (RTN) is a collection of directed graphs. Each directed graph has a set of nodes representing the state of the dialogue, and a set of arcs representing the actions the user can perform. An arc connects two nodes in the directed graph. The user interface moves from the state at the end of the arc to the state at its head if the user performs the action labeling the arc. In a given state the user must perform one of the actions that labels an arc leaving the node representing that state. The arc labels are either the name of an input token generated by the presentation component or the name of another directed graph. In the latter case the named directed graph must be traversed before the state at the end of the arc is reached. In the case of recursive transition networks a directed graph can reference itself.

The tokens to be sent to the application interface model or presentation component can be attached to either the arcs or the nodes (in some systems they can be attached to both). If a token is attached to an arc the token is sent when the arc is traversed. If a token is attached to a node it is sent when the node is entered.

The use of multiple directed graphs facilitates the description of large user interfaces and increases the descriptive power of the technique. The use of a transition network to describe the login sequence for a time sharing system is shown in fig. 2.



Actions:

1) print 'login:'

2) print 'password:'

3) print 'login junk ....'

Fig. 2 Transition diagram for login sequence

Transition diagrams have been used extensively in UIMSs. One of the earliest uses of transition diagrams is the work of Newman [15]. Another example of their use is the

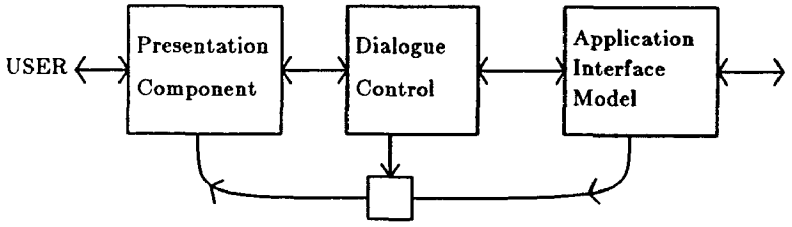


Fig. 1 The components of a user interface

SYNICS system developed by Edmonds [4]. An extension of RTNs called augmented transition networks (ATN) have been used for parsing natural languages [21]. In an ATN arbitrary functions can be attached to the arcs. These functions can store values in registers and use the register values to determine whether an arc should be traversed. This extension greatly increases the computational power of transition networks (ATNs are equivalent to Turing machines).

2.2.2. Context Free Grammars

The second notation for the dialogue control component is context free grammars or BNF. The terminals in these grammars are the input tokens produced by the presentation component. The non-terminals and productions are used to structure the dialogue. For example, there could be a non-terminal for each of the commands in the user interface. The productions with these non-terminals on the left side define the structure or syntax of the commands. A BNF grammar for the login example is shown in fig. 3.

```
login -> user_id password
user_id -> <character_string>
password -> <character_string>
```

Fig. 3 BNF grammar for the login sequence

The grammar in fig. 3 only describes the actions performed by the user, it does not cover the output produced by the program. In order to do this some way of associating tokens with the productions is required. Whenever a production is used in the parse of the user's input these tokens are sent to the presentation component or application interface model.

An unresolved issue with this approach to dialogue control is how to handle the output tokens passed from the application interface model to dialogue control. In some types of dialogues (mixed or system initiated) this flow of tokens may be just as important as the one originating in the presentation component.

Two examples of the use of grammars in the construction of user interfaces are the SYNGRAPH system of Olsen and Dempsey [14] and the work of Hanau and Lenorovitz [11].

2.2.3. Events

The third main notation for the dialogue control component is events. This notation is loosely based on the object oriented approach to user interface design used in Smalltalk [5] and related languages. In this notation the input tokens from the presentation component and the output tokens from the application interface model are viewed as events. These events are processed by event handlers. Each event handler has its own collection of local variables and a collection of procedures for processing events. When an event handler receives an event the associated procedure is executed. These procedures can perform calculations, send events to other event handlers, and send tokens to the presentation component and application interface model. The dialogue control component consists of a collection of event handlers that can change dynamically.

There are several important differences between the event notation and Smalltalk. The event handlers perform the same function as the objects and classes in Smalltalk. The main difference is that there is no explicit inheritance mechanism for event handlers. The main difference between messages and events is that messages are synchronous and events are asynchronous. When a Smalltalk object sends a message it suspends its execution and transfers control to the receiving object. When the receiving object completes its computation control returns to the sending object with a value for the message. In the case of events there is no hand shaking between the sending and receiving event handlers. An event has no value in the sending event handler and the receiving event handler may receive the event any time after it is generated (the sending event handler may not suspend its execution when it generates an event).

An event handler for the login sequence example is shown in fig. 4. This event handler responds to two types of events. The Init event is sent when the event handler is created. In response to this event the login message is printed. The other event is received whenever the user types a character string. The "state" variable is used to determine whether the character string is a user id or a password. In practice the print and process_login statements would be tokens sent to the presentation component and application interface model.

```
Eventhandler login Is

 Token
  keyboardstring s;

 Var
  int state = 0;
  string user_id, password;

 Event Init {
  print "login:";
 }

 Event s : string {
  if(state == 0) {
   user_id = s;
   state = 1;
   print "password:";
  } else {
   password = s;
   state = 0;
   process_login(user_id,password);
  };
 }

End login;
```

Fig. 4 Event handler for the login sequence

The obvious disadvantage to the event notation is that it looks more like a program than the other two notations (depending upon personal biases this may be an advantage). This disadvantage is offset by a number of advantages. First, the expressive power of events is greater than that of recursive transition networks or grammars (the event notations is equivalent to Turing machines while recursive transition networks and BNF grammars are equivalent to pushdown automata [8]). This implies that there are user interfaces that can be described by events that cannot be described by recursive transition networks or BNF grammars. The dialogues in these user interfaces typically depend upon the context of the interaction (the next step in the dialogue depends upon the values of previously entered operands, not just their syntax). The ATN notations mentioned in section 2.2.1 are also capable of describing these dialogues and have the same descriptive power as the event notations. It is important to note that the additional descriptive power of the event notation may not be useful or desirable. The main point of this observation is that dialo-

gues described in the other notations can always be translated into the event notation. This observation forms the basis of our implementation of the dialogue control component. Second, events support multi-threaded dialogues. Since each event handler has its own local state and multiple event handlers can be active at any one time, the user is free to move from any spot in the dialogue to another without completing the current command or explicitly saving the state of the dialogue. In this way event handlers can be developed for help, cancel and other special commands that must always be available. The event handlers processing these commands will always be available whenever the user enters them, it does not require special programming.

2.3. Application Interface Model

The application interface model is the user interface's view of the application. It contains descriptions of all the application's data structures and routines that are accessible to the user interface. The description of this component can be divided into two sections. The first section contains the descriptions of the application routines and data structures. These descriptions are at an abstract or logical level, they are not concerned with how the data structures or routines are implemented.

The description of the application's data structures include the type of information stored and how it is structured. This description might also include the routines that can be used to access and modify the data structures. The description of the application's routines include the name of the routine and the number and types or its parameters. The routine descriptions might also include pre- and post-conditions. The pre-conditions state the conditions that must hold before the routine can successfully be used. They can be used to detect semantic errors, such as manipulating a database before it is opened or printing a binary file. The post-conditions describe the effect of the routine. They can be used to generate help information or aid in undo processing.

The second section of the description of the application interface model covers how the user interface communicates with the application. There are three possible modes of communication called interaction modes. In the first interaction mode, the user initiated mode, the user interface calls routines in the application. This is similar to the external control model presented at the Seattle workshop [17]. In the system initiated mode the application calls routines in the user interface. This is similar to the internal control model. The third interaction mode, mixed initiative, is based on two communicating processes, one for the user interface and one for the application. In this case neither the user interface nor the application has control over the other. In the mixed initiative mode some mechanism for interleaving the execution of the user interface and the application must be used. This could take the form of multiple processes or coroutines. The user interface designer specifies the interaction mode and the UIMS establishes the procedures to implement it. The descriptions of the presentation component and dialogue control are independent of the interaction mode.

3. Designing the User Interface

The University of Alberta UIMS is divided into two main parts, which are: user interface design and run-time support. The design part of the UIMS supports the user interface designer. It provides tools for describing screen layout, device assignments, dialogue structure, and the interaction with the application program. The result of the design part of the UIMS is a detailed specification of the user interface that can automatically be converted into the code required to implement it. The run-time part of the UIMS supports the execution of the user interface. It uses the results of the design part to form a complete executable user interface. This division of the UIMS into design and run-time support is fairly standard and is discussed further in [20].

In this section the design tools provided by the University of Alberta UIMS are briefly described. This discussion serves as the background for the description of the implementation techniques presented in the next section.

3.1. Designing the Presentation Component

The presentation component is concerned with the lexical level of the user interface, including screen layout, menu design, interaction techniques, and icon design. This suggests an interactive approach to the design of this component (this approach has been successfully used in the University of Toronto UIMS [2]). In the University of Alberta UIMS an interactive layout program is used to design the presentation component and a window based graphics package, called WINDLIB [9], is used as the basis of its implementation.

WINDLIB is a window based graphics package similar to the GiGo package developed by Rosenthal [16]. WINDLIB has all the features normally associated with window systems including overlapping windows that can be moved and resized. Some of the nonstandard features of this package are device independence and a set of two and three dimensional graphics primitives. Three features of WINDLIB are used extensively in the presentation component. These features are events, event handlers, and contents structures.

All the input in WINDLIB is in the form of events. An event has a name, a position, and possibly some event specific data. The event name indicates the device that generated the event. In the case of keyboards and other devices that don't generate coordinate information the position of the display's pointing device is used as the position of the event. A window can have an event handler associated with it. An event handler is a procedure that processes the events that are directed at the window. The body of an event handler is usually a case statement on the name of the event. The event handlers can generate events to be sent to other windows. The window that receives a particular event is determined by examining the windows in priority order (from highest to lowest). The first window with an event handler covering the position of the event receives the event.

Contents structures are a hierarchical modeling scheme used for grouping together related pieces of graphical information. A contents structure can be displayed in any window that is currently on the screen. WINDLIB provides contents structures for its two and three dimensional graphics primitives. The programmer can define his own type of contents structure. Programmer defined contents structures are used to represent graphical information in a form that is more convenient to the application. For example, in a charting application the programmer could define contents structures for line graphs, pie charts, bar charts, and histograms. The application only needs to provide the data required for each type of chart, it does not need to produce the graphics primitives that draw the chart. When the programmer defines a contents structure he must provide a routine that traverses the contents structure converting it into graphics primitives. In this way the graphics programmer can provide the applications programmers with a set of routines and data structures that are tuned to their application.

The application programmers do not need to be experts in graphics or be concerned with how the data is displayed.

The design of the presentation component can be divided into three activities, screen layout, interaction techniques, and display techniques. These activities are supported by an interactive layout program, called ipcs (interactive presentation component specification), developed by G. Singh [19].

Ipcs allows the designer to divide the screen into a number of overlapping windows. The designer specifies the size and position of a window by pointing at two opposing corners. The designer can then specify the background colour of the window, its coordinate system, a name for the window, and an output token. The output token associated with a window is used to indicate when the window is to be displayed. When the presentation component receives this token the window is displayed on the screen. A menu can be associated with each of the windows. A menu can either be static (always displayed in the same position) or pop-up (the current cursor position is the upper left corner of the menu). Each menu is viewed as a collection of menu items. A menu item consists of an input token, and a text string or icon. When the menu item is selected its input token is sent to the dialogue control component.

A window can have an interaction technique associated with it. This interaction technique becomes the event handler for the window when it is displayed. The user interface designer specifies the interaction technique by entering the name of a C procedure. This C procedure performs the initialization required by the interaction technique and establishes its event handler. When the window associated with the event handler is removed from the screen a special finish event is sent to the event handler allowing it to deallocate any resources it has acquired. The user interface designer can select interaction techniques from a library or he can write his own.

Ipcs allows the designer to associate display procedures with each of the output tokens that can be processed by the presentation component. For each output token the designer specifies the name of a display procedure and a window where the information is to be displayed. The display procedure is either chosen from a library of standard display procedures or written by the designer. One of the standard display procedures calls WINDLIB to display the contents structure stored in the output token.

The description of the presentation component is stored in an FDB database [10]. This database stores the state of the design between ipcs sessions and is used to generate the presentation component at run time.

3.2. Designing the Dialogue Control Component

The University of Alberta UIMS supports all three notations for the dialogue control component. This gives the user interface designer considerable flexibility in his approach to the design of this component. In order to provide this flexibility the UIMS must have a common format that all three notations can be translated into. This common format forms the basis for the run-time support of the dialogue control component. Since the event notation has more descriptive power than the other two notations the common format, EBIF (Event Based Internal Form) is based on the event notation. EBIF is described in section 4.1.

3.2.1. Event Language

The event language used in the University of Alberta UIMS is based on the C programming language [12]. Since C is the main programming language used in our research group this significantly reduces the time required to learn the language. A program in the event language consists of a number of event handlers. The text of the program contains one or more event handler definitions. When the program is executed instances of these event handlers are created. It is the instances that perform computations, not the event handlers themselves. There may be several instances of the same event handler, parameters can be used to establish the state of an instance when it is created.

```
Eventhandler event_handler_name Is

 Token
  token_name event_name ;
        .
        .
        .

 Var
  type variable_name = initial_value ;
        .
        .
        .

 Event event_name : type {
    statements
 }
        .
        .
        .
 Event event_name : type {
    statements
 }

end event_handler_name;
```

Fig. 5 Structure of event handler declarations

The structure of an event handler declaration is shown in fig. 5. An event handler declaration is divided into three sections. The first section lists the tokens (either input or output) that the event handler can process. This information is used by the assembler (see section 4.2) to map tokens into events for event handlers. The event language compiler places the token information in a table separate from the event handlers. In this way the the assignment of token names, and the mapping between tokens and events can be changed (in the assembly process) without effecting the event handlers themselves.

The second section of an event handler declaration contains the declarations of the event handler's local variables. Each instance of the event handler has its own set of local variables, there is no sharing of storage between instances. A variable declaration consists of a type, a variable name, and an optional initial value. The type can be any valid C type that occupies the same amount of space as a pointer. This includes characters, integers, floating points numbers (single precision only) and pointers to any C type. This restriction simplifies the implementation of the language and may be lifted in the future.

The third section consists of event declarations. An event declaration starts with the keyword Event followed by the name of the event and its type. The body of the event declaration consists of one or more C statements. These

statements are executed when an instance of the event handler receives this event. The statements can reference the instance's local variables and the global variables in the program. The data associated with the event is assigned to the event name before the execution of the statements in the event declaration.

There are a number of special procedures that are used in event handlers. The format of these procedures is shown in fig. 6. The create_instance procedure is used to create a new event handler instance. The parameters to this procedure are the name of the event handler, the number of local variables to be initialized and their initial values. The local variables are initialized in the order they are listed in the variable declaration section. The value returned by this procedure is the name of the new instance. When an instance is created an Init event is automatically sent to it. The send_event procedure is used to send an event to an event handler instance. The parameters to this procedure are the name of the instance, the name of the event, and the data associated with the event. The send_token procedure is used to send a token to another component of the user interface. The parameters to this procedure are the component to receive the token, the direction of the token (input or output), the name of the token, and its value. The destroy_instance procedure is used to destroy the instance that is given as its parameter. Before destroy_instance deallocates the instance a Finish event is sent to it. This event allows the instance to free any resources it has accumulated in its execution.

1) create_instance(event_handler, n, v_1, v_2, ... , v_n)

2) send_event(instance_name, event_name, value)

3) send_token(destination, direction, name, value)

4) destroy_instance(instance_name)

Fig. 6 Event language support procedures

More details on the event language and its implementation can be found in [3].

3.2.2. Recursive Transition Networks

In the University of Alberta UIMS an interactive approach is taken to the design of recursive transition networks. There is a natural graphical representation for recursive transition networks, this suggests that an interactive graphical approach is appropriate for them.

The interactive transition diagram editor produced by S.C. Lau [13] is used to enter and edit RTNs. This editor is based on a graphical display of the transition network. The designer can use a tablet or mouse to enter and edit the nodes and arcs in a diagram. Each arc in the diagram has an input token, and optional output tokens to be sent to the presentation component and application interface model when the arc is traversed. One interesting feature of this editor is the ability to select and save a group of nodes and arcs. This group can then be added to another diagram in the user interface.

The transition diagrams are stored in an FDB database. This database is used to store the diagrams between editing sessions and is used to generate the EBIF for the dialogue control component. A separate program is used to convert the transition diagrams to EBIF. More details on the transition diagram editor and conversion to EBIF can be found in [13].

3.2.3. Grammars

At the present time a grammar based notation has not been implemented. A number of grammar based notations exist (for example [14]). The major activity in implementing this type of notation is developing the algorithms required to convert productions into event handlers or EBIF. We intend to do this sometime in the future.

3.3. Designing the Application Interface Model

At the present time support for the application interface model is under development. Currently only one interaction mode (user initiated) is supported and the main use of this component is to map between tokens and the routines in the application.

The mapping between tokens and application routines may not be one-to-one. A token may cause several application routines to be executed, or it may contain data used in a subsequent call of an application routine. In order to support this behavior the application interface model must provide storage for saving token values and a means of associating a sequence of actions with a token.

```
Var
  type variable_name;
        .
        .
        .

Token token_name : token_type {
  statements
}
        .
        .
        .
```

Fig. 7 Structure of the application interface model

In the University of Alberta UIMS a written notation is used for describing the application interface model. This notation is converted into C code and tables which become part of the user interface at run-time. The structure of the application interface is shown in fig. 7. The first part of this description defines the storage locations used by the application interface model. The variable declarations in this section have the same syntax as C variable declarations. The values of these variable are preserved from one token to the next. The second section of the description contains one entry for each token processed by the application interface model. This entry contains the name of the token, its type, and the statements to be executed when it is received. The statements are standard C statements that can call application routines and save the value of the token. Note the similarity between the application interface model and the event language discussed in section 3.2.1.

4. Implementation

In this section an overview of the implementation of the University of Alberta UIMS is presented. This discussion centers around the structure of the event based internal form and how it is interpreted by the run-time routines.

4.1. EBIF

All the program used to design the dialogue control component produce EBIF as output. An EBIF file consists of a number of event handler definitions. Each event handler definition is divided into two parts. The first part contains information used by the run-time routines to create instances of event handlers and route tokens between these instances. This information is placed in the three main tables that drive the run-time routines (see fig. 8).

The second part of the event handler definition is a C procedure containing all the executable statements in the event handler. This procedure is called each time an event must be executed. The body of this procedure is a case statement on the name of the event. This procedure has four parameters, which are: the name of the instance, the name of the event to be processed, the value of the event, and an array containing the values of the instance's local variables.

The three tables used by the run-time routines are shown in fig. 8. The event table has one entry for each event handler. This entry contains a pointer to the corresponding C procedure, and the number of local variables for each instance. There is one entry in the instance table for each active instance. This entry points to the array containing the instance's variable values, and the index in the event table of the corresponding event handler. The token table is used to map between tokens and the event handlers that process them. Each entry in this table contains the name of a token, the name of the event it is converted to, and the event handler that can process it.

4.2. User Interface Assembly

The assembly of the user interface is performed by a program called the assembler. The input to this program is the EBIF files produced by the dialogue control programs, a file of the input and output tokens associated with the presentation component (produced by ipcs), the output file from the design of the application interface model, and a token definition file. The output from the assembler is a file of C routines, which must be compiled, and the tables used by the run-time support routines. The process of converting a program in the event language into an executable user interface is shown in fig. 9.

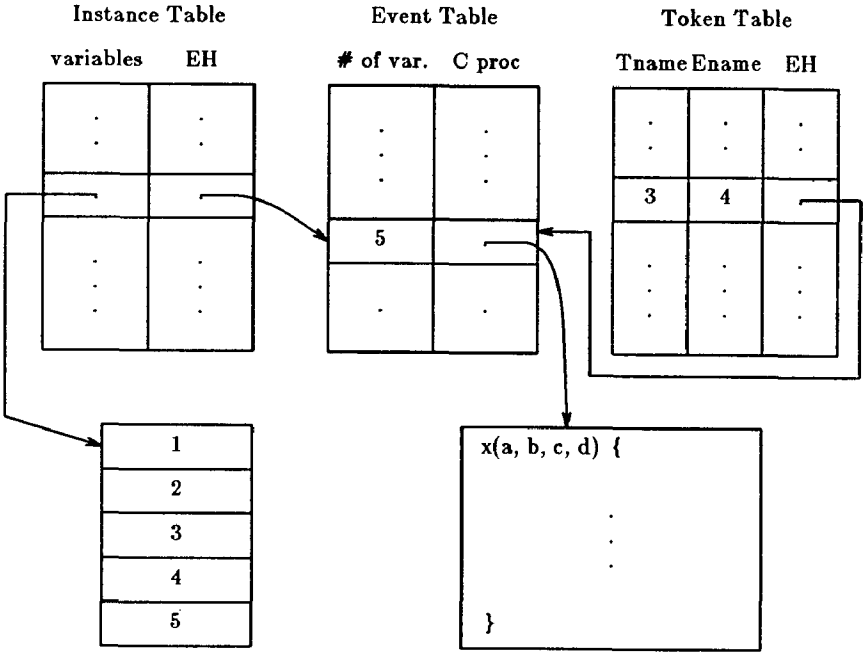


Fig. 8 Run-time tables

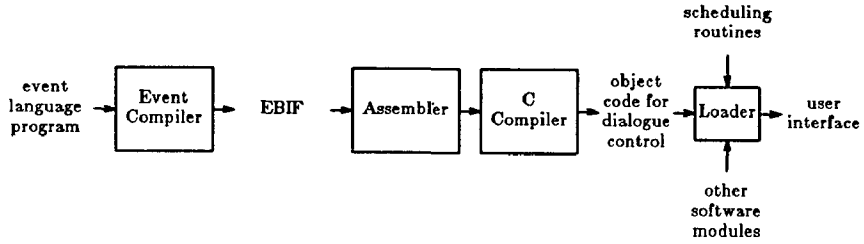


Fig. 9 Converting an event program into a user interface

4.3. Run-time Routines

In the design of the user interface it has been assumed that each of the components is a separate process. The only way of exchanging information between components is through tokens, which is an asynchronous communications mechanism. This illusion of concurrency must be maintained by the run-time support routines. The approach that we have used is to view the run-time routines as a scheduler that allocates processing time to the individual components. The unit of scheduling at the component level is the token.

When one component sends a token to another component that token is placed on a scheduling queue associated with the receiving component. The scheduler examines the scheduling queue associated with each of the components and selects one of the tokens for execution. Highest priority is placed on the presentation component, and lowest priority on the application interface model queue, with the restriction that a token will not be blocked for an arbitrary long time. The scheduler then calls the appropriate routine in the receiving component to process the token. This routine can be determined from the tables produced by the assembler.

1) If an input device has input ready call the presentation component to process it.

2) If there are pending events in dialogue control execute several of them

3) Select a token from one of the scheduling queues and execute it.

4) Goto step (1)

Fig. 10 Steps in the scheduling process

Both the presentation component and dialogue control have a lower level of scheduling. Before the higher level scheduler can process a token the presentation component determines if any of its input devices has a value ready. If this is the case the value is converted into a WINDLIB event and processed by the presentation component. Similarly if there are events waiting in the dialogue control component a small number of them are processed before the next token is processed. The steps in this two level scheduling process are shown in fig. 10.

This two level scheduling process has three main advantages. First, it gives priority to the components closer to the user, thus he will have fast feedback to lexical operations, and slower feedback to semantic operations. Second, it supports the view of the three components as separate processes allowing them to be designed separately. Third, it leaves the door open for the implementation of the user interface as three separate processes on three closely coupled processors, which may be feasible in the near future.

5. Summary

In this paper we have presented the design of the University of Alberta UIMS. The goals of this UIMS have been outlined along with discussions of its main components and implementation. The implementation has reached the point where we have built several small applications with the system. We are now working on rewriting all the interactive design programs so they use the UIMS.

One of our original aims was to produce a system architecture that allows for growth and experimentation, and at the same time supports production applications. We believe we have at least partially reached this goal by separating the design of the user interface from its run-time support. The run-time support is fairly stable and the users of the UIMS are largely not aware of the internal formats it uses. By taking this approach we hope to be able to incorporate comments from designers and users into future versions of the UIMS.

There are a number of enhancements and extensions to the University of Alberta UIMS that we would like to investigate. One of the most obvious extensions is adding a grammar based notation for the dialogue control component. This would give the user interface designer the full spectrum of design notations for this component. Another useful extension would be providing an interactive assembly program. This would be more convenient than the current batch approach to user interface assembly. It might also facilitate the management of different versions of the user interface intended for different workstations and user groups.

There are three extensions that have a significant research component. The first is developing a more flexible approach to screen layout. The current approach is based on a relatively static screen layout, the only variability is in pop-up menus. We would like to be able to automatically adjust the screen layout based on the type and amount of information displayed. For example, automatically changing the size of a window depending upon the amount of information stored in it, or automatic selection of display techniques based on the type of data and the amount of detail required. The second extension is automatic undo processing. This would be an extension to the application interface model that would allow the user to undo any recent action, or replay recent commands with different operands. This undo mechanism should be automatically provided by the UIMS. The third major extension involves the interaction between the UIMS and database systems. The schemas used by most database systems are a good first order approximation to the application interface model. Given a schema we would like to automatically produce the corresponding application interface model. The schema might also suggest commands and operations that should appear in the user interface. It might be possible to produce an augmented schema that can be used to produce both the database and the user interface. This issue is explored in [1].

References

[1] Armstrong W.W., M. Green, P. Srirangaptna, "A Database Management System and Associated Tools for a General Design Environment", Proceedings of the 1984 Canadian Conference on Very Large Scale Integration, p.183-187, 1984.

[2] Buxton W., M.R. Lamb, D. Sherman, K.C. Smith, "Towards a Comprehensive User Interface Management System", Siggraph'83 Proceedings, p.35-42, 1983.

[3] Chia M.S., *An Event Based Dialogue Specification for Automatic Generation of User Interfaces,* MSc Thesis, Department of Computing Science, University of Alberta, 1985 (expected).

[4] Edmonds E.A., "Adaptive Man-Computer Interfaces", in M.J. Coombs and J.L Alty, *Computing Skills and the User Interface,* Academic Press, London, 1981.

[5] Goldberg A., D. Robson, *Smalltalk-80: The Language and its Implementation,* Addison-Wesley, Reading Mass.,

1983.

[6] Green M., "Report on Dialogue Specification Tools", Computer Graphics Forum, vol.3, p.305-313, 1984.

[7] Green M., "The University of Alberta User Interface Management System: Design Principles", Human-Computer Interaction Project Report #1, Department of Computing Science, University of Alberta, 1984.

[8] Green M., "User Interface Models", Human-Computer Interaction Project Report #2, Department of Computing Science, University of Alberta, 1985.

[9] Green M., N. Bridgeman, "WINDLIB Programmer's Manual", Department of Computing Science, University of Alberta, 1985.

[10] Green M., M. Burnell, H. Vrenjak, M. Vrenjak, "Experiences With a Graphical Data Base System", Proceedings of Graphics Interface'83, p.257, 1983.

[11] Hanau P.R., D.R. Lenorovitz, "Prototyping and Simulation Tools for User/Computer Dialogue Design", Siggraph'80 Proceedings, p.271-278, 1980.

[12] Kernighan B.W., D.M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs NJ, 1978.

[13] Lau S.C., *The Use of Recursive Transition Networks for Dialogue Design in User Interfaces,* MSc Thesis, Department of Computing Science, University of Alberta, 1985 (expected).

[14] Olsen D.R., E.P. Dempsey, "SYNGRAPH: A Graphic User Interface Generator", Siggraph'83 Proceedings, p.43-50, 1983.

[15] Newman W.M., "A System for Interactive Graphical Programming, SJCC 1968, Thompson Books, 1968.

[16] Rosenthal D.S.H, "Managing Graphical Resources", Computer Graphics, vol.17, no.1, p.38-45, 1983.

[17] Graphical Input Interaction Technique Workshop Summary, Computer Graphics, vol.17, no.1, p.5-66, 1983.

[18] Pfaff G., P.J.W. ten Hagan, *Seeheim Workshop on User Interface Management Systems,* Springer-Verlag, Berlin, 1985.

[19] Singh G., *Automatic Generation of Presentation Component for University of Alberta UIMS,* MSc Thesis, Department of Computing Science, University of Alberta, 1985 (expected).

[20] Tanner P.P., W.A.S. Buxton, "Some Issues in Future User Interface Management System Development", in G. Pfaff and P.J.W. ten Hagen (ed), *Seeheim Workshop on User Interface Management Systems,* Springer-Verlag, Berlin, 1985.

[21] Woods W.A., "Transition Network Grammars for Natural Language Analysis", CACM vol.13, no.10, p.591-606, 1970.

A touch-sensitive tablet (touch tablet for short) is a