Generating Graphical Interfaces from High-Level Descriptions

Gurminder Singh† and Mark Green

Department of Computing Science, University of Alberta
Edmonton, Alberta T6G 2H1, Canada

†Current Address: Institute of Systems Science
National University of Singapore, Kent Ridge, Singapore 0511.
ISSGS@NUSVM.bitnet

ABSTRACT

The UofA* User Interface Management System (UIMS) generates graphical user interfaces based on a high-level description of semantic commands supported by the application. A main part of the UIMS, called Diction, generates the dialogue control component of interfaces. Diction enables the interface designer to implement prefix, postfix, and nofix (or order-free) syntax types in open-ended or close-ended selection modes. The aim of this paper is to discuss in detail the design and implementation of Diction.

Keywords: User Interface Design, User Interface Management System, Dialogue Design Tools.

1. Introduction

We have developed a UIMS, called the UofA* UIMS, which uses high-level descriptions of the semantic commands supported by the application to generate graphical user interfaces. The goals of the UIMS are:

- to automatically generate the lexical and syntactic design of graphical user interfaces.
- to enable the interface designer to refine the interfaces produced by the UIMS.

Figure 1 shows the overall organization of the UIMS, the heart of which consists of Diction, Chisel, and vu. Diction accepts a description of the semantic commands and produces the dialogue control component of the user interface. It also produces output which is used, in conjunction with (display) device description and optionally with the end user's preferences, by Chisel to produce the presentation component of the interface. The presentation component produced by Chisel can be refined by the designer by using an interactive graphical facility called vu.

The aim of this paper is to discuss the design and implementation of Diction. Greater details about the UIMS and how it is used can be found in [Singh87-Singh89a, Singh89c]. Diction generates dialogue control components capable of handling prefix, postfix, and nofix (or order-free) syntax types. It accepts a high-level description of commands supported by the application and generates "event handlers" to implement the dialogue control. The command description accepted by Diction is based on implicit I/O event ordering [Hayes85].

Over the past few years a number of UIMSs have been built which have proven to be effective in reducing the time and effort required for creating user interfaces. There are, however, a number of problems with most existing UIMSs. These systems require detailed descriptions of the user interface to be constructed. The descriptions cover details of the presentation component, dialogue control component, and the interface to the application. The first problem with this approach is that to be able to use the system the designer must be familiar with the notation used for describing the interface. Most notations are complex, and gaining fluency in their use takes much time and effort. The second problem is that since the descriptions are quite detailed it takes a long time to produce them, and the descriptions are usually difficult to modify. The third problem is that because the approach is expensive in both time and effort, it discourages experimentation.

Most of these problems have been alleviated in our UIMS by creating the interface from high-level descriptions. This approach facilitates a rapid development of the interface by directly converting the specification into its implementation. In this approach, the designer does not have to deal with low level details, instead his main concern is with macro level features of the interface

2. Generating a Dialogue Control Component

The UofA* UIMS provides the designer with a "semantic command" metaphor for defining the interface. The basic user interface definition is therefore in terms of the semantic commands supported by the application. This command description is produced in a simple high-level notation. For a distributed network editor the command description input to the UIMS is shown in figure 2. This editor is used by the distributed systems research group at the University of Alberta [Singh89b] to create and edit a network of process templates. The user can create connections between templates and associate attributes with them. The big advantage of this type of description is that it is compact, and therefore easy to produce and modify. From this description and a description of the display device, the UIMS automatically generates the presentation and dialogue control components of a graphical user interface. This interface allows the user to enter commands through menus and enter command arguments through

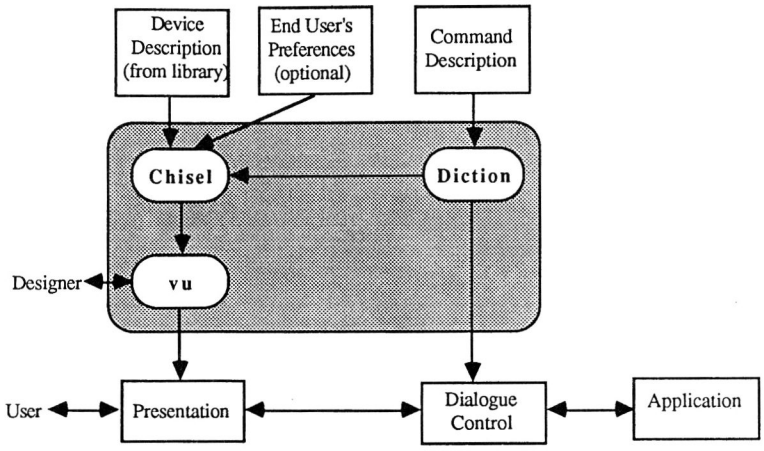

Figure 1. Structure of the UIMS

interaction with graphical interaction techniques. The default presentation component generated by Chisel can be refined by the designer by using vu (see figure 1). This paper focuses on how Diction implements dialogue control components of graphical user interfaces.

Once the designer has provided the input shown in figure 2, a working prototype of the dialogue control component can be generated by Diction. When linked with the presentation component, the application interface, and the semantic routines, it completes the creation of the interface. In the default dialogue control component generated by Diction, the commands are parsed in the prefix close-ended fashion, i.e. the command is selected before providing its argument values. The entry of argument values is unordered. After a command is executed it is deselected by the system.

Diction also generates textual help messages which explain the syntax and argument requirements of the commands. When the user selects a command, its corresponding help message is displayed by the UIMS, unless the help facility has been disabled by the user.

There are a number of features of the dialogue control component that can be controlled by the designer by making minor modifications to the command description. Diction enables the designer to implement commands in prefix, postfix, and nofix syntaxes. Commands can also be selected in open-ended or close-ended modes. Diction provides considerable flexibility in handling the command arguments. For example, to make the Add_First_Template command work in open-ended mode and to assume currently selected values (CSVs) for the icon_name, process_name, and the script_file arguments, the designer modifies the command as follows:

```
Add_First_Template( icon_name:ICONS {CSV}, place:
  WIND, process_name:PROCESS_NAME {CSV},
  script_file:FILE_NAME {CSV})
```

To execute this Add_First_Template command, the user only needs to provide the value for the place argument; other arguments assume the current values of their respective interaction techniques. Once selected, the Add_First_Template command can be used to add an arbitrary number of templates in the work area.

Diction cuts down tremendously on the time required for developing interfaces. In one of the experiments in which we developed an interface for a three dimensional skeleton editing system, the use of the UIMS resulted in a speed-up by a factor of 28 over the University of Alberta UIMS [Green85, Singh86]. The difference in time and effort would have been even greater if comparisons were made with a conventional programming language.

3. Design Goals of Diction

The chief design goal of Diction was that it should be able to handle a variety of syntaxes. Diction enables the interface designer to implement prefix, postfix, and nofix types of dialogues. For a command of the form

command (arg1, arg2)

the prefix syntax means that the command must be selected before the arguments are selected. The arguments could be selected in an arbitrary sequence. In the case of a postfix command, arguments must be selected before the command is selected. The nofix syntax means following the prefix, postfix, or any arbitrary sequence for selecting the arguments and the command.

Diction also enables the designer to implement individual commands in the interface in open-ended or close-ended fashion. Open-ended commands accept an arbitrary number of complete arguments. For example, an interface could support an Add-Object command which, once selected, allows multiple objects to be added, one after the other, until another command is selected. A typical input sequence for the above command would be

```
command
    arg1 arg2
    arg1 arg2
    arg1 arg2
      .
      .
      .
command1
    args
```

```
/* The specification may start with a global declaration of syntax and selection types to be applied to all commands. The declaration
 at the command level overrides the global declaration of syntax and selection types. If the global declaration is omitted, as in
 this case, the default syntax and selection types (PREFIX CLOSE_ENDED) are assumed. An example of this declaration is:

   SYNTAX = NOFIX
   SELECTION = OPEN_ENDED

 Declaration of global arguments follows. ICONS is an enumerated type argument, having six enumerations.
 The APPLICATION_NAME, PROCESS_NAME, FRAME_NAME, and ROUTINE_NAME arguments are of character type.
 The WIND argument is of type pick (2d).*/

ICONS : (executive inpipeline pipeline terminal assimilator manager);
APPLICATION_NAME : char;
PROCESS_NAME : char;
FRAME_NAME : char;
ROUTINE_NAME : char;
FILE_NAME : char;
WIND : pick;

/*Commands are declared as follows. The command name is optionally followed by its syntax and/or selection type which is followed
 by a variable-length list of command arguments. For each argument, its name followed by type (or range/enumerations) are specified.
 Argument type can be one of globals or it can be a standard type. Argument type may be followed by the default value in the case of
 local arguments, or it may be followed by CSV (Currently Selected Value) in the case of global arguments.

 For example, the Add_First_Template command has four arguments: icon_name, place, process_name, and script_file. All arguments are
 of global type.*/

Load (application_name : APPLICATION_NAME)
Add_First_Template (icon_name : ICONS, place : WIND,
 process_name : PROCESS_NAME, script_file : FILE_NAME)
Add_Second_Template (icon_name : ICONS, place : WIND)
Remove_Template (place : WIND)
Make_Contractor (process_name : PROCESS_NAME)
Remove_Contractor (process_name : PROCESS_NAME)
Add_Resource_Manager (process_name : PROCESS_NAME,
 place : WIND, script_file : FILE_NAME)
Add_Routines {OPEN_ENDED} (process_name : PROCESS_NAME {CSV},
 calling_frame : FRAME_NAME, reply_frame : FRAME_NAME,
 routine_name : ROUTINE_NAME)
Remove_Resource_Manager (place : WIND)
Add_Link (template1 : WIND, template2 : WIND,
 calling_frame : FRAME_NAME, reply_frame : FRAME_NAME)
Remove_Link (template1 : WIND, template2 : WIND)
Save (application_name : APPLICATION_NAME)

/*The application_name arg of the Close command is a CSV type arg.*/
Close (application_name : APPLICATION_NAME {CSV})
Quit ()
```

Figure 2. Command Description for the Distributed Network Editor

Diction also enables the interface designer to use default and initial values for command arguments. The distinction between the default and initial values is that initial values apply to global arguments whereas default values apply to non-global (or command-level) arguments. Globally declared arguments are set to their initial values when the interface is first started. The user can change the argument values through interaction with the interface. The arguments which are not global can have default values. The value of such an argument is set to its default value every time the command containing the argument is selected. This argument value can also be changed by the user. The difference between the use of default and initial values is that initial values are set just once, when the interface is initialized, whereas the default values are set every time the command containing the default arguments is selected.

The second major design goal of Diction was that it should facilitate rapid prototyping of interfaces. This means that it should enable the designer to produce a number of variants of an interface without much additional effort. In Diction, the designer can produce interfaces which are different from each other by changing the global parameters of Diction, or by changing command-level parameters. The macro as well as micro level behavior of the interface can easily be controlled by the interface designer.

The third design goal of Diction was that it should be easy to use. Unlike a number of existing UIMSs (e.g. SYNGRAPH [Olsen83], the University of Alberta UIMS [Green85], ALGAE [Flecchia87], Sassafras [Hill86], and Scott and Yap's UIMS [Scott88]), Diction does not require detailed specification to produce the dialogue control component. It accepts a high-level specification of commands and converts the specification into program modules which imple-

ment the dialogue control. Even though the command description is at a much higher level than in many UIMSs, it enables the designer to control the interface behavior to a very low level.

4. Output of Diction

The dialogue control components produced by Diction consist of program modules called "event handlers". An event handler is a process (defined by a procedure or module) that is capable of processing certain types of events. When an event handler receives one of the events it can process, it executes a procedure. This procedure can perform some computation, generate new events, call application procedures, create new event handlers, or destroy existing event handlers.

The behavior of an event handler is defined by a template. A template consists of several sections that define the parameters to the event handler, its local variables, the events it can process, and the procedures used to process these events. When an event handler is created, its template must be specified, along with values for its parameters. The result of the creation process is a unique name that is used to reference the event handler. Several event handlers can be created from the same template. Each of the event handlers created from a template can have a different local state.

In the event model a dialogue control component is described by the set of templates that define the event handlers it uses. At the start of execution an instance of one of these templates is created to serve as the main event handler in the dialogue control component. This event handler will then create (possibly indirectly) all the other event handlers. Conceptually, all the event handlers in the dialogue control component execute concurrently, processing events as they arrive. Event handlers have been used in a number of UIMS, such as the University of Alberta UIMS [Green85] and ALGAE [Flecchia87] for implementing dialogue control components.

The brief description of the event model presented here should be sufficient to understand how Diction uses event handlers to implement dialogue control components. However, if additional detail is of interest, [Green86] and [Singh89c] can be consulted.

The complete dialogue control component designed by Diction consists of a number of event handlers, and each event handler is responsible for one well defined function. For example, an event handler may be responsible for parsing a particular command or for generating help messages. Diction produces one event handler per command, which is responsible for handling default values for the command arguments, parsing the command, notifying the presentation component when errors occur, and notifying the application when the command is successfully parsed. In addition to producing event handlers for commands, Diction produces two event handlers responsible for house-keeping and producing help information for commands.

The structure of event handlers produced by Diction is shown in figure 3. The keywords in figure 3 are printed in bold. An event handler declaration is divided into four sections. The first section declares the name of the event handler. In the event handlers produced by Diction, the event handler name is the same as the name of the command parsed by the event handler; only the case is inverted.

```
eventhandler sample is
    token
        token_name  event_name
        .
        .
        .
    var
        type variable_name = initial_value;
        .
        .
        .
    event event_name{
        statements
    }
        .
        .
        .
    event event_name{
        statements
    }
end sample;
```

Figure 3. Structure of an Event Handler

The second section lists the input tokens that the event handler can process. For each token name, this section also declares an event name. This information is used to map tokens into events for the event handler. In the event handlers produced by Diction, command arguments are represented by input tokens, and the event name for a token name is produced by prefixing it with IN_.

The third 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. In the event handlers produced by Diction, this part of event handlers is rarely used.

The fourth section consists of event declarations. An event declaration starts with the keyword "**event**" followed by the name of the event. The body of the event declaration consists of a number of 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 rest of this section explains how event handlers are used to implement the dialogue control.

When the user interacts with the presentation component, input tokens are produced. These tokens are added at the end of a token queue reserved for the dialogue control component. The run-time control removes these tokens from the front of the queue, one at a time, and processes them. Processing a token involves sending the event corresponding to the token to the event handlers (see figure 4). An event handler receives only those events which it can process.

When the application is first started, the run-time control instantiates two event handlers. The first event handler is called the HOUSE_KEEPER. It remains active throughout the interactive session, and its main responsibilities include initializing interaction techniques, instantiating event handlers

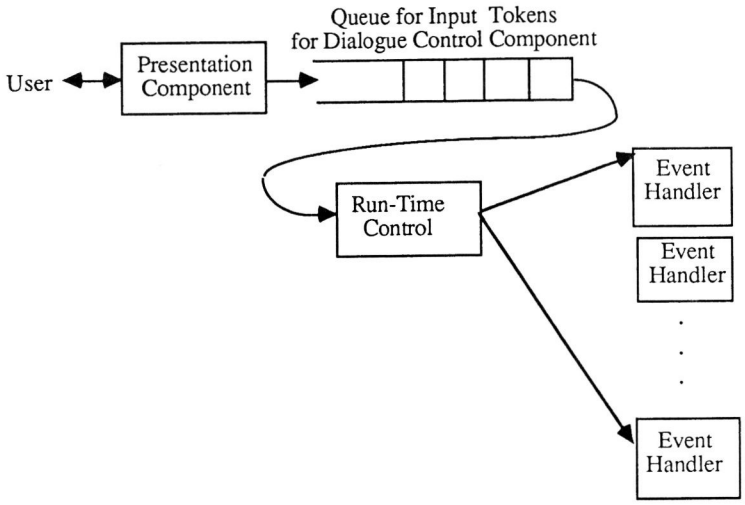

Figure 4. Run-Time Control

for commands, and maintaining data-structures used by the dialogue control component. The second event handler, called HELPER, generates help messages for commands selected by the user.

When the user selects a command, an input token identifying the command is generated by the presentation component. The run-time control converts this token into an event and sends the event to the HOUSE_KEEPER and the HELPER. The HELPER generates a number of output tokens for the presentation component which produce help messages for the selected command. The HOUSE_KEEPER, on receiving the event, instantiates the event handler for the command selected by the user. On instantiation, an INIT event is sent to the newly created event handler automatically by the run-time system. The new event handler, with some help from the HOUSE_KEEPER, is responsible for parsing the command.

When the user interacts with the presentation component to provide argument values, tokens are converted into events and sent to the HOUSE_KEEPER. The HOUSE_KEEPER updates the status and values of the command arguments tied to the event received. After doing so it generates a CHECK token which is sent to the active command event handler. The command event handlers treat the CHECK token as a signal for a change in argument status and values. So on receiving this token the event handlers check whether the argument values they need are available or not. When the required argument values become available a token is generated for the application. On receiving this token the application executes the selected command.

For example, consider the Add_First_Template command shown in figure 2. The event handler produced by Diction for this command is shown in figure 5. In the event handlers produced by Diction, argument names provided by the designer are replaced by the argument names prefixed with the command name they belong to. This is necessary in order to create unique argument names.

When the user selects the Add_First_Template command, its event handler shown in figure 5 is instantiated by the HOUSE_KEEPER, and an INIT event is sent to the Add_First_Template instance automatically. On receiving this event, the Add_First_Template event handler sets the status of icon_name, place, process_name, and script_file arguments to UNDEF. When the user provides any of the argument values, corresponding events (IN_ICONS, IN_WIND, IN_PROCESS_NAME or IN_FILE_NAME) are sent to the HOUSE_KEEPER by the run-time control. For each of the events, the HOUSE_KEEPER updates the status and value of the corresponding argument, and generates a CHECK token. The CHECK token triggers the Add_First_Template event handler to determine whether the command can be executed or not. When all the arguments are defined, the Add_First_Template event handler generates a token for the application. After doing so it informs the presentation component to deselect the command, and it commits suicide.

5. Implementation

Diction has been implemented in Lex [Lesk75] and Yacc [Johnson75] on a VAX 11/780 running UNIX† 4.3 BSD. The run-time environment is implemented in the C programming language. The complete sequence of converting a high-level command description into a user interface is shown in figure 6. The box named "Convert" converts the event handlers into C programs.

6. Comparison with Existing Systems

The majority of existing UIMSs require a detailed specification of the dialogue control component. The specification may take the form of modified-BNF [Bleser82, Olsen83, Reisner81, Shneiderman82], transition networks [Green85, Jacob83, Newman68, Wasserman85], and invented languages [Flecchia87, Green85, Hill86]. Some notations are easier to use than others, but a common feature of all the specifications is that they explicitly specify what sequences of I/O events constitute a valid dialogue between the user and the application, and these specifications are quite detailed. As a result, producing a specification takes a great deal of time, and since the specification is quite large it tends to be error-prone and difficult to modify. Also, because these specifications are based on explicit event ordering it is very hard to support modeless interaction; modelessness has to be

```
/* Event Handler generated by Diction for the Add_First_Template command from figure 2. */

eventhandler aDD_fIRST_tEMPLATE is  /* PREFIX CLOSE_ENDED */
  token    /* token declaration */
    CHECK  IN_CHECK;

/*On receiving the INIT event from the run-time component, set the status of arguments to UNDEF. This happens on instantiation of the
 event handler. */

  event INIT{
    Add_First_Template_icon_name.status = UNDEF;
    Add_First_Template_place.status = UNDEF;
    Add_First_Template_process_name.status = UNDEF;
    Add_First_Template_script_file.status = UNDEF;
    break;
  }

/*On receiving the CHEcK token from the HOUSE_KEEPER, first check whether all the argument values are available. If not do nothing.*/

  event IN_CHECK{
    if (Add_First_Template_icon_name.status == DEF &&
      Add_First_Template_place.status== DEF &&
      Add_First_Template_process_name.status== DEF &&
      Add_First_Template_script_file.status== DEF){

/*If the values are available, create a list of argument values and generate a token for the application to execute the Add_First_Template
 command. The list of argument values is sent as value of the token.*/

      values = (int *) calloc( 4, sizeof( int ));
      values[0] = Add_First_Template_icon_name.value;
      values[1] = Add_First_Template_place.value;
      values[2] = Add_First_Template_process_name.value;
      values[3] = Add_First_Template_script_file.value;
      send_token( APPLICATION, OUTPUT, Add_First_Template, values);

/*As the Add_First_Template command is a CLOSE_ENDED command, generate a token for the presentation component to deselect
 the command*/

      send_token( PRESENTATION, INITIAL, "cmenu", "-1");

/*Set the status of the Add_First_Template command to OFF and commit suicide.*/

      cmd_Add_First_Template.status = OFF;
      destroy_instance( self_id );
    }
  }
end aDD_fIRST_tEMPLATE;
```

Figure 5. Event Handler for the Add_First_Template Command

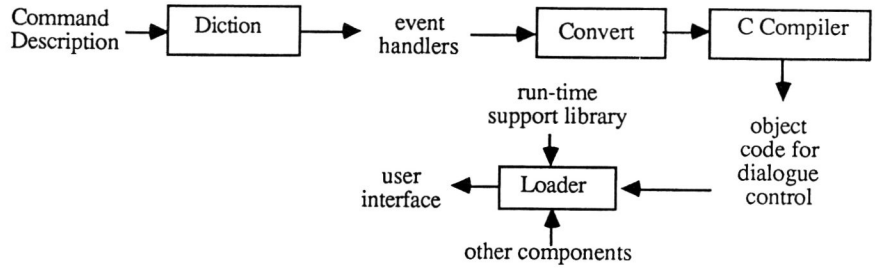

Figure 6. Converting Command Description into a User Interface

† Registered trademark of AT&T in the USA and other countries.

programmed into the interface specification. When compared with these UIMSs, Diction is distinctly superior. The first advantage of using Diction is that the input it accepts is in a simple high level notation. In an informal experiment, a number of users were asked to create user interfaces for a variety of applications. All the users found the notation to be easy to learn and understand. Gaining fluency in the use of the notation also did not take much effort or time. The second advantage of Diction is that it significantly shortens the time required to develop a dialogue control component, and also reduces the chance of error to a minimum as the designer is dealing with a compact and higher level of specification. The third advantage of Diction is that it facilitates rapid-prototyping of interfaces by using a specification which is compact and therefore, easy to modify. The final advantage is that the specification accepted by Diction specifies sets of I/O events required by commands without mentioning any specific ordering. As a result, it is easy to support modeless interaction.

Approaches similar to the one followed by Diction have been used in MIKE [Olsen86] and UIDE [Foley88, Foley89]. MIKE accepts a high-level specification of semantic commands supported by the application and generates programs to implement dialogue control in prefix close-ended fashion. It parses for command arguments in the specific order in which they are specified in the command definition. This is the only style which is supported by MIKE. A number of ambiguities would have to be resolved if MIKE were to parse for command arguments in any order, or if it were to implement other syntax types. Providing these facilities may require a complete overhaul of MIKE's control mechanism.

UIDE provides a high-level conceptual design tool in which the designer describes the user interface as a knowledge-base. UIDE can algorithmically transform the knowledge base into a number of functionality equivalent interfaces, each of which is slightly different from the original interface. The transformed interface definition can be input to a UIMS, called SUIMS (Simple UIMS) which implements the user interface. SUIMS implements the dialogue control in prefix close-ended fashion. The command is selected before its arguments which can be selected in an arbitrary sequence. The main difference between SUIMS and Diction appears to be the way in which the dialogue control is implemented. SUIMS behaves like an interpreter whereas Diction generates event handlers which implement the dialogue control. Also, SUIMS's support for syntax is restricted to prefix close-ended only (other syntax types are planned but not implemented yet [Foley89]).

7. Summary and Conclusions

An overview of the UofA* UIMS which generates graphical user interfaces has been presented. An important part of the UIMS, called Diction, handles the dialogue control component of user interfaces. The general goal of Diction is to enable the interface designer to quickly create and test various types of dialogues. The approach it follows to achieve this goal is based on implicit I/O event ordering. This approach alleviates common problems associated with systems which are based on explicit I/O event ordering and hence require detailed interface specifications.

Diction generates dialogue control components capable of handling prefix, postfix, and nofix syntax types. As far as we know, Diction is the first system to provide support for these syntax types, and is the first system to demonstrate that various syntax types can co-exist in the same interface.

The UofA* UIMS has been used by a number of users to create user interfaces for a variety of applications including a 3-dimensional skeleton editor used by the animation research group, a distributed network editor used by the distributed systems research group, a stickman animation system, a fish animation system, and a paint program. These applications have exercised a variety of Diction's capabilities, including flexible syntax and automatic help. Based on this experience a number of conclusions can be drawn. First, by using a specification which is based on implicit I/O event ordering, and by using a simple high level notation for the specification, Diction eliminates much of the complexity associated with constructing dialogue control components. Second, Diction facilitates rapid prototyping of user interfaces by using a specification which is compact and therefore, easy to modify. Third, the use of Diction results in substantial savings in time and effort required for creating user interfaces. As described earlier, the use of the UofA* UIMS resulted in a speedup by a factor of 28 over that of the University of Alberta UIMS for creating the 3-dimensional skeleton editor interface. A major part of the speed-up resulted because of Diction.

8. Acknowledgments

We would like to thank Chris Shaw of the University of Alberta for reading and providing useful comments on an earlier version of this paper.

References

Bleser82.
T W Bleser and J D Foley, Towards Specifying and Evaluating the Human Factors of User-Computer Interfaces, *Conf. on Human Factors in Computer Systems, Gaithersburg MD.*, Mar. 1982, 309-314.

Flecchia87.
M A Flecchia and D R Bergeron, Specifying Complex Dialogs in ALGAE, *Proc. CHI+GI'87 Human Factors in Computing Systems*, Toronto, Ont., Canada, Apr. 5-9, 1987, 229-234.

Foley88.
J D Foley, C Gibbs, W C Kim and S Kovacevic, A Knowledge-Based User Interface Management System, *Proc. CHI'88 Human Factors in Computer Systems*, Washington, D.C., May 15-19, 1988, 67-72.

Foley89.
J D Foley, W C Kim, S Kovacevic and K Murray, Defining Interfaces at a High Level of Abstractions, *IEEE Software*, Jan. 1989, 25-32.

Green85.
M Green, The University of Alberta User Interface Management System, *Computer Graphics 19*, 3 (July 1985), 205-213. (Proc. SIGGRAPH'85 Conf., July 22-26, 1985, San Francisco, California).

Green86.
M Green, A Survey of Three Dialogue Models, *ACM Transactions on Graphics 5*, 3 (July 1986), 244-275.

Hayes85.
P J Hayes, P A Szekely and R A Lerner, Design Alternatives for User Interface Management Systems Based on Experience with COUSIN, *Proc. CHI'85 Human Factors in Computing Systems*, San Francisco, Apr. 14-18, 1985, 169-175.

Hill86.R D Hill, Supporting Concurrency, Communications and Synchronization in Human-Computer Interaction-The Sassafras User Interface Management Systems, *ACM Transactions on Graphics 5*, 3 (July 1986), 179-210.

Jacob83.
R J K Jacob, Executable Specifications for a Human-Computer Interface, *Proc. CHI 1983 Human Factors in Computing Systems*, Boston, MA, Dec. 12-15, 1983, 28-34.

Johnson75.
S C Johnson, *Yacc: Yet Another Compiler-Compiler*, Technical Report 32, AT&T Bell Labs, Murray Hill, New Jersey, 1975.

Lesk75.
M E Lesk and E Schmidt, *Lex - A Lexical Analyser Generator*, Technical Report 39, AT&T Bell Labs, Murray Hill, New Jersey, 1975.

Newman68.
W M Newman, A System for Interactive Graphical Programming, *Proc. Spring Joint Computer Conf.*, 1968, 47-54.

Olsen83.
D R Olsen and E P Dempsey, SYNGRAPH: A Graphical User Interface Generator, *Computer Graphics 17*, 3 (July 1983), 43-50. (Proc. SIGGRAPH'83 Conf., July 25-29, 1983, Detroit, Michigan).

Olsen86.
D R Olsen, MIKE: The Menu Interaction Kontrol Environment, *ACM Transactions on Graphics 5*, 4 (Oct. 1986), 318-344.

Reisner81.
P Reisner, Formal Grammar and Human Factors Design of an Interactive Graphics System, *IEEE Transactions on Software Engineering SE-7*, 2 (Mar. 1981), 229-240.

Scott88.
M L Scott and S Yap, A Grammer-Based Approach to the Automatic Generation of User-Interface Dialogue, *Proc. CHI'88 Human Factors in Computer Systems*, Washington, D.C., May 15-19, 1988, 73-78.

Shneiderman82.
B Shneiderman, Multiparty Grammars and Related Features for Defining Interactive Systems, *IEEE Transactions on Systems, Man and Cybernetics SMC-12*, 2 (1982), 148-154.

Singh86.
G Singh and M Green, Automatic Generation of Graphical User Interfaces, *Proc. Graphics Interface '86* , Vancouver, B.C., May 26-30, 1986, 71-76.

Singh87.
G Singh and M Green, Visual Programming of Graphical User Interfaces, *Proc. 1987 Workshop on Visual Languages*, Linkoping, Sweden, Aug. 19-21, 1987, 161-173.

Singh88a.
G Singh and M Green, **vu** - visual user-interface design workshop, Graphics Interface '88 - Film Show , *Edmonton, Alberta, Canada, June 6-10, 1988, video tape.*

Singh88b.
G Singh and M Green, Designing the Interface Designer's Interface, *Proc. ACM SIGGRAPH Symposium on User Interface Software*, Banff, Alberta, Canada, Oct. 17-19, 1988, 109-116.

Singh89a.
G Singh and M Green, A High Level User Interface Management System, *Proc. CHI'89 Human Factors in Computing Systems*, Austin, Texas, Apr. 30-May 4, 1989, (in press).

Singh89b.
A Singh, *FrameWorks Model of Distributed Computing in Workstation Environment*, Ph.D. Thesis, Univ. of Alberta, Edmonton, Alberta, Canada, 1989. (expected).

Singh89c.
G Singh, *Automating the Lexical and Syntactic Design of Graphical User Interfaces*, Ph.D. Thesis, Dept. of Computing Science, University of Alberta, Edmonton, Canada, 1989.

Wasserman85.
A I Wasserman, Extending State Transition Diagrams for the Specification of Human-Computer Interaction, *IEEE Transactions on Software Engineering SE-11*, 8 (Aug. 1985), 699-713.