The University of Alberta

An Event Based Dialogue Specification for Automatic Generation of User Interfaces

by

Meng-Song CHIA

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Fall, 1985

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Meng-Song CHIA

TITLE OF THESIS: An Event Based Dialogue Specification for Automatic Generation of User Interfaces

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Master of Science

YEAR THIS DEGREE GRANTED: 1985

(Signed) [illegible]
Permanent Address:
Lot 241, Central Avenue
Central Road East
Kuching, Sarawak
Malaysia

Dated 17 May 1985

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **An Event Based Dialogue Specification for Automatic Generation of User Interfaces** submitted by **Meng-Song CHIA** in partial fulfillment of the requirements for the degree of **Master of Science.**

[illegible]

Supervisor

[illegible]

[illegible]

[illegible]

Date [illegible]

To my parents

ABSTRACT

This thesis presents an event based specification language for designing and implementing flexible human-computer dialogues. This language allows a user interface designer to give a high-level specification of a user interface in a device independent manner. The specification is used to generate code for the user interface. The event language is part of a user interface management system (UIMS) developed at the University of Alberta. The UIMS is based on the Seeheim model of user interfaces that was developed at the Seeheim workshop. The Seeheim model divides the user interface into three components. The rationale for building a UIMS and a survey of a number of different UIMSs are also discussed.

The important contribution of this thesis is that the event language provided is capable of supporting multi-threaded dialogues. The language can be used in the design and implementation of user interfaces for solving real application problems and it also serves as a vehicle for experimenting with user interfaces.

Acknowledgements

I would like to thank a number of people who helped make this thesis possible. First of all I would like to thank my supervisor, Dr. Mark Green, for introducing me to this topic and offering valuable advice, encouragement, and criticism at each stage of my research. He has suffered through many drafts of this thesis, and improved its readibility. I am also grateful to him for his financial support.

Further thanks are due to the members of my examining committee, Dr. Tony Marsland, Dr. Duane Szafron, and Dr. Jack Mowchenko for their helpful suggestions and comments. I would also like to acknowledge both the financial and technical support provided by the Department of Computing Science.

My fond thanks to Lee-Choo whose affection and help through the darker times were always there.

Most of all, I would like to thank my family, especially my parents, for their continuing support and encouragement throughout my education.

Table of Contents

Chapter — Page

Chapter 1: Introduction 1
1.1. Rationale for Developing a UIMS 1
1.2. Purpose of the Thesis 3
1.3. Terminology 4
1.4. Outline of the Thesis 5

Chapter 2: Survey of Existing UIMSs 6
2.1. Classification of UIMS 6
2.1.1. External vs Internal Control 6
2.1.2. Glue Systems vs Module Builders 8
2.2. MENULAY 9
2.3. SYNGRAPH 11
2.4. TIGER 12
2.5. SYNICS and SYNICS2 13
2.6. Seeheim Model 14
2.6.1. Presentation Component 15
2.6.2. Dialogue Control Component 16
2.6.2.1. Recursive Transition Network Notation 17
2.6.2.2. Context Free Grammar Notation 20
2.6.2.3. Event Notation 21
2.6.3. Application Interface Model 23

Chapter 3: Designing the Dialogue Control Component 25
3.1. The Event Model 26
3.2. The Event Based Internal Form 27
3.2.1. Scheduler 27
3.2.2. Event Handler File 29

Chapter 4: The Event Language 31
4.1. Design Objectives 31
4.2. Introduction to the Event Language 32
4.2.1. Token Declaration 34
4.2.2. Variable Declaration 34
4.2.3. Event Declaration 35
4.3. Multi-Threaded Dialogues 37
4.4. Summary 38

Chapter 5: Implementation 40
5.1. The Compiler 40
5.1.1. Syntactic Specification 40
5.1.2. Lexical Analysis 42
5.1.3. Compilation Process 43
5.1.4. Keywords 44
5.1.5. Summary 45
5.2. Scheduler Routines 46
5.3. Process Scheduling 47
5.3.1. Environment 48
5.3.2. Scheduling 49

Chapter 6: Examples 51
6.1. A Simple Geometrical Object Editor 51
6.2. An Advance Geometrical Object Editor 56
6.3. Comments 61

Chapter 7: Conclusions 62
7.1. Summary of Thesis Contributions 62
7.2. Evaluation 62
7.3. Extensions and Further Work 64

References 66

A1: YACC Specification of the Event Language 69

A2: Lexical Analyzer for the Event Language 79

List of Figures

Figure		Page
1.1	Automatic Generation of User Interface	1
2.1	Structure of Internal Control UIMS	7
2.2	Structure of External Control UIMS	7
2.3	UIMS with Glue System and Module Builder	9
2.4	A Typical Sequence for Dialogue Construction	10
2.5	TIGER Application Architecture	12
2.6	Logical Model of a UIMS	15
2.7	Transition Diagram for Login Sequence	18
2.8	BNF Grammar for The Login Sequence	20
2.9	Event Handler for The Login Sequence	21
3.1	Structure of the Dialogue Control Component	25
3.2	Relationships between Scheduler Tables	28
3.3	Format of An Event Handler File	29
4.1	Structure of An Event Handler Declaration	33
5.1	BNF Description of Event Declaration	41
5.2	User Interface Generation Process	48
6.1	Screen Layout for A Simple Geometrical Object Editor	51
6.2	Event Handler for the Simple Geometrical Object Editor	54
6.3	Screen Layout for An Advance Geometrical Object Editor	56
6.4	Event Handler for the Advance Geometrical Object Editor	59
6.5	Event Handler for Processing Help Messages	60

Chapter 1

Introduction

1.1. Rationale for Developing a UIMS

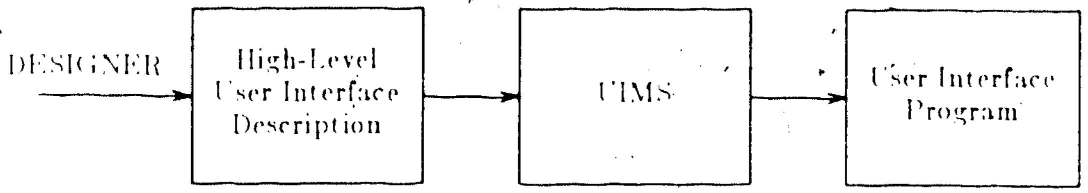


Figure 1.1 Automatic Generation of User Interface

An important reason for the developing of a User Interface Management System (UIMS) is because the design and implementation of good user interfaces for interactive computer systems is an expensive and time-consuming activity. Although the cost of computer hardware has been decreasing, the cost of software has been increasing owing to the expensive nature of designer's time. It has been noted that on the average 59% of interactive business application programs involve the management of the user interface [Benbasat and Wand 84]. The time required for a highly skilled programmer to construct the interface averaged about 35 months. Therefore, to reduce both the cost and time required to construct good user interfaces, the notion of a UIMS, a software tool that automatically constructs the user interface given a high level description, has been introduced as shown in Figure 1.1 [Buxton et al, 83, Kasik 82, Olsen 83, Seattle 83, Tanner and Buxton 84].

The second reason for working on such a system is related to the desire to develop more user-friendly interactive systems, which has further compounded the cost of interface construction. A good user interface should support a spectrum of users efficiently. Nowadays, interactive computer systems are available in offices, schools and homes. This has led to the emergence of a new class of users who are not programmers. Unlike expert programmers, these users are often not willing to spend a lot of time learning the systems. Novice users would generally prefer a computer-guided system with extensive help and error correction facilities, however, expert users would prefer a user-guided system that accepts abbreviated commands and allows entry of multiple commands to speed up the dialogue. Furthermore, over a short learning period novice users would become more experience users. Therefore, there is a need for a UIMS that allows the designer to efficiently modify the user interface to suit the evolving needs of the users and the demands of users with different levels of expertise

The third reason for developing a UIMS is related to system development by prototyping [Benbasat and Wand 84, Edmonds 81, Green 82]. These prototypes can vary from a simple rough sketch to detailed working models, they are done to introduce the users to the system quickly, so their feedback can be used in refining the prototoypes. This method allows the designer to evaluate his design before it is put into actual practice.

The fourth reason for the development of a UIMS is the increased reliability and ease of modification of automatically generated software over hand coded methods. The traditional hand construction of the user interface for each new application is not only time consuming and costly, but also tends to produce low quality user interface. This need not be the case since the same user interface techniques can be used in a wide variety of applications. Therefore, the development of a UIMS has been introduced. The UIMS has significant impact on the structure of applications. It separates the specification of the interaction dialogue from the programming of the application and enables the former to be altered with minimum effect on the latter. This separation provides a higher degree of user interface consistency across the applications. The UIMS removes the burden of physical interaction handling from the designer and allows him to concentrate on the design of better dialogue sequences. In addition, the UIMS permits the user dialogue to be programmed by someone, such as an interaction specialist, who is not a programmer. Therefore, the user interface design can be tailored to the user rather than to efficiency of implementation.

This research has also been motivated by the lack of a sound methodology for the design of user interfaces. Experiments involving reimplementing entire applications are prohibitively expensive, but the ease with which user interfaces can be changed using a UIMS makes realistic experiments possible. The UIMS can be used as a test bed for the evaluation of various human-computer interaction techniques and physical devices. The notion of a UIMS is similar to that of a Data Base Management System in many ways [Date 81].

1.2. Purpose of the Thesis

Although several UIMSs have been designed and implemented over the last few years, the design of a good user interface remains to a great extent an art. One major problem with all the designs is the lack of experience in their use. Therefore, in order to overcome this deficiency, a UIMS based on the Seeheim model of user interfaces has been developed at the University of Alberta [Green 84c, 85a]. This model divides the user interface into three components: presentation component, dialogue control component and application interface model. A discussion of these components is given in Chapter Two. In this thesis the design and implementation of the dialogue control component using the event model is presented. This dialogue control component is based on study of desirable features in dialogue specification languages for automatic generation of user interfaces. In the event model, the user interface is described by a set of event handler definitions using an event language. The definition of an event handler specifies the events it can handle and the statements that are executed in response to these events. The event language is an extension of the C programming language [Kernighan and Ritchie 78]. The main aim of this thesis is to test our ideas on user interface design and implementation. It is hoped that this thesis will shed some light on the issues raised. Besides using the UIMS as a test bed for evaluating the feasibility of the Seeheim model as the basis for UIMS, it will also be used as a practical tool for other research projects within our department.

An alternative approach to the design of the dialogue control component using the recursive transition network notations can be found in [Lau 85]. The presentation component is implemented by G. Singh [Singh 85]. The other components of the UIMS will be implemented by other authors in the near future. This UIMS project is one of the first user interface projects to incorporate the Seeheim model of user interface into practice.

1.3. Terminology

For the purpose of this thesis a UIMS is defined as a set of software tools that support the specification, design, implementation, maintenance, and evaluation of human-computer dialogues. The name "Abstract Interaction Handler" (AIH) used in [Kamran and Feldman 83] and [Rogers and Feldman 81] also refers to the same software tools.

In this thesis the term "user" refers to the end user of the application system, the person for whom the user interface was designed. The "user interface" is defined as the software module that stands between the user and the rest of the program [Green 84a]. It is the only part of the program that directly deals with the user, all the other modules obtain user input from the user interface and rely on the user interface to present output to the user. The "interaction designer" or simply the "designer" is the person who designs the interactive dialogue. He is familiar with various interaction techniques and human factors information. The "application programmer" is the person who writes the application. An "interaction dialogue" is the set of possible user inputs where each input causes some information to be sent to the application, possibly resulting in some syntactic or semantic feedback on the display [Tanner and Buxton 84].

1.4. Outline of the Thesis

This thesis is organized as follows. Chapter Two contains a description of criteria for classifying UIMSs and a survey of different design models used for describing them. The third chapter of this thesis describes the dialogue control component of the University of Alberta UIMS. The event language is described in Chapter Four. Chapter Five contains the details of the implementation of a scheduler and a compiler for the event language. A number of examples of the use of the event language is given in Chapter Six. The final chapter presents conclusions and suggestions for further work. The appendices contain the Lex source program and the YACC specification of the syntax of the event language.

Chapter 2

Survey of Existing UIMSs

This chapter gives a description of the classification criteria, and a survey of the design notations and models used for describing UIMSs. The survey provides the background for the design of the University of Alberta UIMS and the event based dialogue specification described in the following chapters.

2.1. Classification of UIMS

Despite the growing interest in user interface design, the nature of human-computer interaction is still not well understood. There does not exist a large body of knowledge on the design of UIMSs. Most of the UIMS generators that have been produced have a user interface model, but this is usually *not* emphasized in their descriptions. In this section, we look at different conceptual models that have been used for classifying UIMSs. A UIMS is usually classified along the following axis: external vs internal control of application, and glue systems vs module builders.

2.1.1. External vs Internal Control

One of the goals of UIMS development has been to separate the specification of the user dialogue from the programming of the application [Kamran and Feldman 83; Rogers and Feldman 81, Rosenthal 82]. Two alternative models describing the relationship between the UIMS and the application have been suggested by Rosenthal and Yen [Seattle 83].

The first model is an internal control UIMS where the application program is in charge of the flow of control within the graphics system, as shown in Figure 2.1. The application invokes input functions to obtain appropriate input data from various abstract devices.

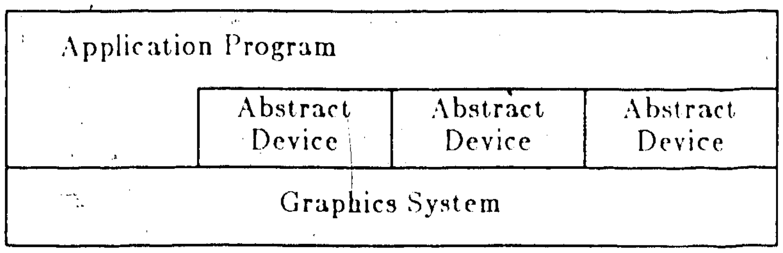


Figure 2.1 Structure of Internal Control UIMS

The other model is an external control UIMS where control lies within the UIMS, as illustrates in Figure 2.2. The application is viewed as a set of discrete functional modules that will be invoked in response to user inputs. All UIMS implementations that are discussed in the following sections have used the external control model.

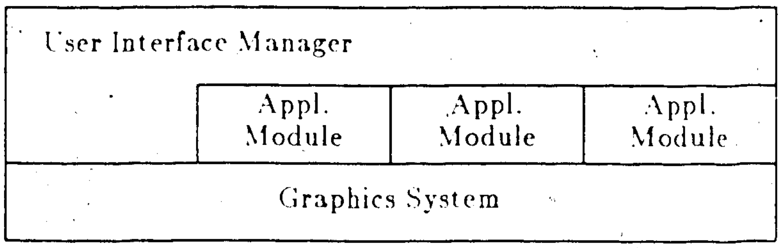


Figure 2.2 Structure of External Control UIMS.

However, the problem with either of these models is that it is not always clear where the boundary line should be drawn between the UIMS and the application. For example, should semantic feedback, such as telling the user that the requested operation has been completed, be handled by the UIMS or the application package? The UIMS cannot perform semantic feedback when this requires knowledge and manipulation of the application data base. On the other hand, the application cannot perform semantic feedback without informing the UIMS. It has been argued that the UIMS must always call an application routine whenever it wants to change the application data structure in order to ensure all the modifications are legal.

2.1.2. Glue Systems vs Module Builders

Different UIMSs have made some trade-off between generality and ease-of-use in the design of their pre-processor modules. This has led to two different types of systems called glue systems and module builders.

In a glue system, a designer uses a library of interaction techniques to design the interaction dialogues. Therefore, the power of such a system depends largely on the range and power of the library. This system allows the designer to construct the dialogues at a high level with minimum complexity.

In a module builder, the designer uses a special language for defining the interaction dialogues. This system is quite general. The designer is not restricted to the set of interaction techniques provided in some libraries, and will usually create his own library of software modules. However, this system requires the designer to learn a new programming language.

Since the glue system and the module builder both have their own strengths and weaknesses when implemented separately in different systems, they should co-exist in a UIMS in order to compliment each other as shown in Figure 2.3. In this case, a programmer can use the module builder to design software modules that require a more specialized knowledge of programming concepts, and store these modules in the module library. An interaction designer who is not a programmer can then use the glue system to construct dialogue sequences by patching together these ready-made modules [Tanner and Buxton 84].

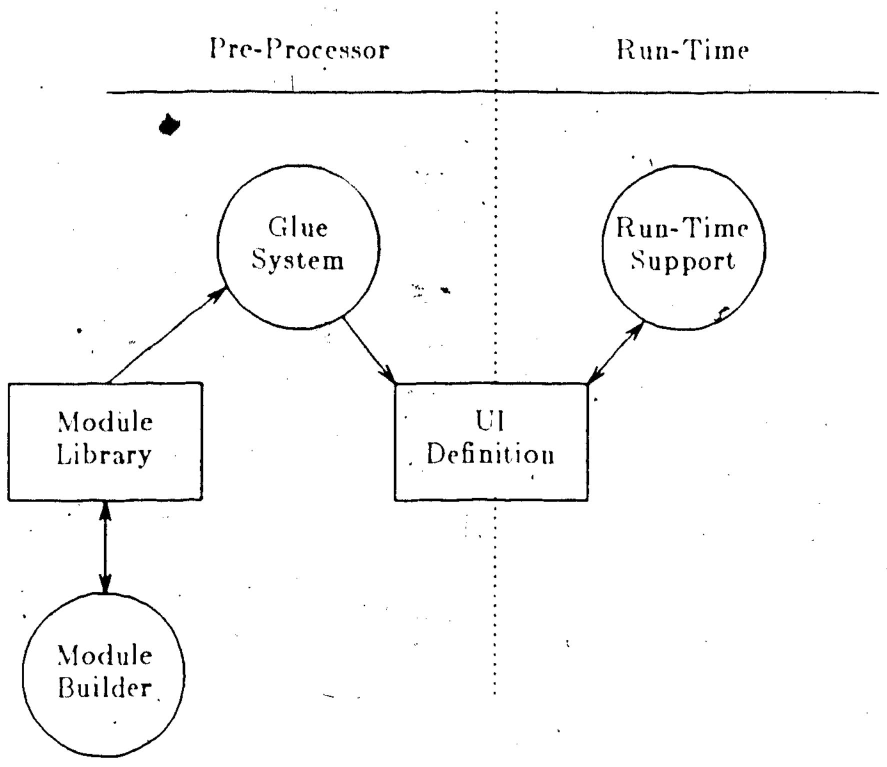


Figure 2.3 UIMS with Glue System and Module Builder

2.2. MENULAY

The UIMS [Buxton et al. 83] developed at the University of Toronto consists of two main modules. The first is a pre-processor, called MENULAY, that enables the designer to design graphics menus and their functionality. The second main module is the run-time support package that handles interaction between the user and the system.

The pre-processor MENULAY is a glue system that establishes the relationship between the user interface and application program. The designer is provided with a library of ready-made modules that include specialized iconic cursors, a graphical potentiometer, a graphical piano keyboard, an audio support package for driving a digital sound synthesizer, and some routines to manipulate light buttons within the menu driven system. He can graphically define the icon for an event and layout the event on the screen. The name of a procedure associated with an event is entered through a keyboard. The output from the pre-processor is converted by a companion program MAKEMENU into C programs that build a number of tables used in the user interface. The resulting code is linked with application-specific routines.

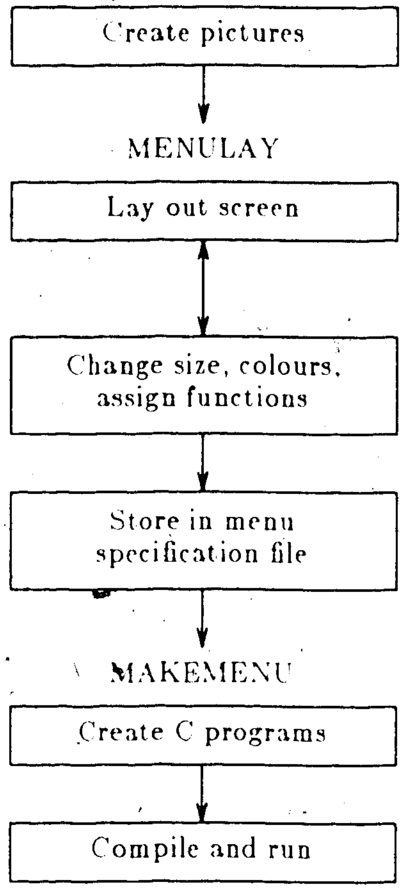


Figure 2.4 A Typical Sequence for Dialogue Construction

The run-time support package handles the user interactions with the resulting executable module. This package is designed for supporting event-driven interaction and a variety of input devices. A typical sequence for constructing an interactive dialogue using this UIMS is shown in Figure 2.4.

One main advantage of this model is that it is easy to design interaction dialogues using MENULAY. The designer can modify the dialogues easily by associating different application routines with the input events. Another advantage is that the UIMS is language independent. To produce code in a different language would only involve rewriting the MAKEMENU and the run-time support routines. The major disadvantage of this UIMS is that the range of user interactions is limited by the power of the library of interaction techniques.

2.3. SYNGRAPH

The SYNGRAPH (SYNtax directed GRAPHics) system [Olsen 83, Olsen and Dempsey 83a, 83b] is a UIMS that uses a module builder as its pre-processor. The system automatically generates graphical user interfaces. It produces interactive Pascal programs from a lexical specification, a syntactic specification, and the underlying semantics of interaction.

The lexical specification defines the various input devices and interaction techniques that can be used by the user. These input devices and techniques are bound to logical token names. There is no conceptual limit to the number of such tokens that can be created. The mapping of these tokens into the interactive resources is handled syntactically.

The syntactic specification specifies both the sequence of valid input tokens and the organisation and arrangement of their prompts and echos. The syntax of the dialogue is expressed in terms of an extended BNF (Backus Naur Form) that uses the logical token names defined in the lexical specification as well as non-terminals declared in the grammar. Semantic actions can be inserted in the grammar to invoke the application routines provided. The system performs automatic prompting by examining the set of input tokens that are acceptable for a given state. It is also possible to add additional help information to the grammar, and to provide rubout and cancel facilities for handling error recovery.

The SYNGRAPH system is written in Pascal and the interaction semantics are handled using Pascal statements. Each non-terminal is generated as a recursive Pascal procedure whose control structure is the syntax tables produced from the grammar. Each such non-terminal can have parameters and local declarations. Pascal statements can be inserted into the production to perform semantic actions.

One advantage of this model is that the designer is free to create his own library of interaction techniques in addition to those being provided. A disadvantage is that the designer needs to learn a new language for describing the dialogue sequences.

2.4. TIGER

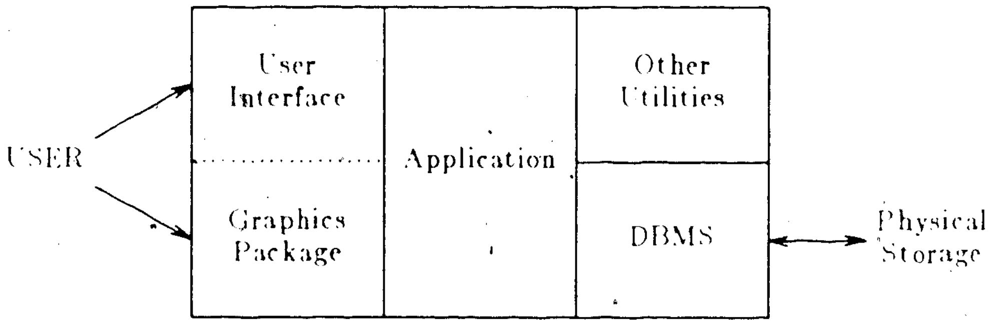


Figure 2.5 TIGER Application Architecture

The TIGER (The Interactive Graphics Engineering Resource) system of Kasik [Kasik 82] is another example of a UIMS using a module builder. The graphics package, shown in Figure 2.5, contains output only capabilities. The user interface handles all interaction. The TIGER system consists of two main components: a specification language and a run-time interpreter. The TIGER Interactive Command and Control

Language (TICCL) permits the designer to concentrate on the design of dialogue sequences rather than the low-level steps that must be taken to accomplish the task. The designer can give a high-level specification of the user interface using the TICCL language that is strictly block structured.

The run-time interpreter assumes both input and output responsibility for the interactive dialogue. The results of the compiled TICCL language are used as input to the interpreter. The interpreter displays all information to the user in the form of menus. It collects interrupts and processes them according to the constraints imposed by the static TICCL structure and the dynamic inputs provided by the application at run time. The interpreter passes specific information to the application, which is written in Pascal, via parameter lists that are specifically tailored to a particular input.

Like the SYNGRAPH system, the TIGER system also requires the designer to learn a new control language.

2.5. SYNICS and SYNICS2

In this section, we discuss two software tools, SYNICS and SYNICS2, developed by Edmonds and Guest [Edmonds 81, 82a, 82b, Guest 82] to facilitate the design of user interfaces.

SYNICS is another UIMS using the module builder approach. It is a translator writing system that uses a grammar notation to specify user interfaces. The system provides a compiler for converting a BNF specification into executable codes. The specification consists of a set of SYNtax and semantICS rules. The system also provides special features such as an *if-then-else* statement for condi[illegible] parsing, an *interrupt* node for calling a predefined routine, and a *redo* facility for selective back-up. A set of FORTRAN subprograms is provided for linking the compiled SYNICS output with the application routines. One problem with this system is that most designers do not seem to know what BNF is.

In order to provide an alternative tool for human-computer interfaces, Edmonds and Guest have developed another dialogue design system, SYNICS2 [Guest 82, Guest and Edmonds 84], using the recursive transition network notation for describing the interfaces. The system consists of three sets of programs, and the loader from the SYNICS system. The main program is the dialogue processor which takes a dialogue language and produces executable output. The other two programs are the dialogue tester, which drives the dialogue system, and the dialogue subroutines, which can be called by the dialogue control program.

The dialogue language is made up of a description of nodes which represent the states of the dialogue. Each node can be viewed as a circle with at least one entry arc, except for the initial start node or a called node, and at least one arc leaving, except for a terminal or return node. Each node is described by a node number, and a set of actions to be performed upon entering this node. These actions may include branching statements that represent actions labeled on the arcs, procedure call with parameter passing, and exit commands.

An advantage of the SYNICS2 system is that transition networks are easy to understand. The user input can be described easily using the dialogue language. The main disadvantage is that the description for a real application is usually very large.

2.6. Seeheim Model

This section describes the Seeheim model of user interfaces developed at the workshop on User Interface Management System sponsored by Eurographics and IFIPS on Nov. 1-3, 1983, in West Germany [Green 84b, 84c, 85a, Guest and Edmonds 84, Seeheim 84]. This model divides the user interface into three components as shown in Figure 2.6. The presentation component is responsible for physical interactions. The graphics package has been absorbed into this component. The dialogue control component deals with dialogue between the user and the application program. The application interface model describes the application's data structures and routines that are accessible to the user interface. A pipeline has also been provided to allow the flow of data from the application interface model to the presentation component. The UIMS and the application are treated as separate processes. This model can work with both internal and external control, and it consists of both glue system and module builder approaches. It suggests the use of "parallel" control.

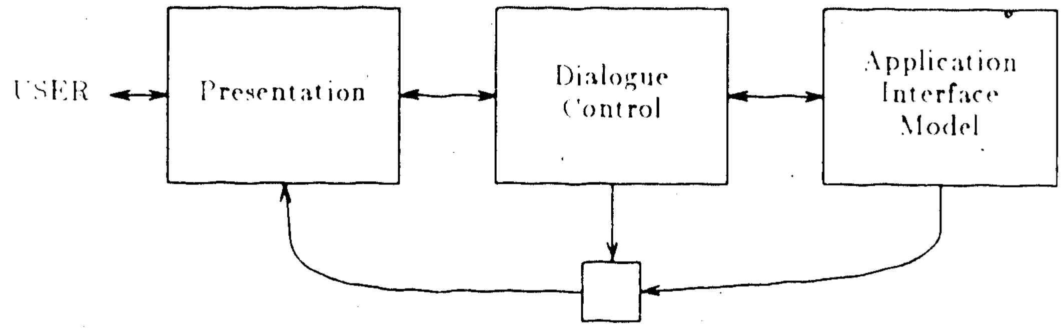


Figure 2.6 Logical Model of a UIMS

2.6.1. Presentation Component

The presentation component deals with the lexical aspect of the user interface. It is responsible for handling screen layout, graphical display, input and output devices, lexical feedback, external-internal mapping, and interaction techniques. When a user interacts with the input devices by selecting an item from a menu or pressing a button on a keyboard, the presentation component generates an input token that consists of a type field and a number of data fields that depend on the type of the token. The input token is then sent to the dialogue control component. In return, the dialogue control component may send some output tokens to the presentation component resulting in some graphical displays, or lexical feedback such as moving a cursor or echoing characters. The external-internal mapping determines the conversion of the user's interactions with the input devices into input tokens and the output tokens into images on the output devices. The dialogue control component has some control over this mapping. It can choose the display techniques used for particular output tokens and the interaction techniques used for particular input tokens from predefined sets. Besides menu selection, some other interaction techniques commonly used in graphics application programs include dragging, rubberbanding and picking. The presentation component is the only component of the UIMS that must deal with the physical devices.

A separate presentation component not only increases portability and ease of modification of a user interface, but also decreases the costs of its construction. Since the presentation component is the only device dependent component of the UIMS, only this component needs to be changed when the user interface is moved to a different display device. The separation provides a convenient means of tailoring the lexical aspect of the user interface for individual users. It allows the users to change default commands and select their favorite interaction techniques or display procedures for a particular type of application. A separate presentation component also encourages the construction and use of a standard library of interaction techniques. This will improve the quality of user interfaces and reduce the time and costs of their construction.

2.6.2. Dialogue Control Component

The dialogue control component defines the structure of the dialogue between the user and the application program. This component receives a sequence of input tokens, representing data supplied or requests made by a user, from the presentation component and routes them to the appropriate routines in the application program. Similarly, the dialogue control component also receives a sequence of output tokens from the application program and routes them to the appropriate parts of the presentation component. These output tokens may represent requests for data or answers to user requests. Therefore, the dialogue control component must deal with two dialogues, one initiated by the user, and the other initiated by the application program. Since the actions performed by the dialogue control component depend upon the context of the dialogue, it must be capable of handling dialogue states and state changes. The dialogue control component has control over the flow of data from the application to the presentation component. It assigns the formats to the output data, and establishes a pipe line between the application and the presentation component. Once the pipe line has been set up, the dialogue control component does not take part in the information transfer. This is analog to direct memory access. The flow of data is represented by the arc connecting the application interface model and the presentation component in Figure 2.6. This approach is effective for transfering large amount of data from the application interface model to the presentation compenent.

Most existing UIMSs have concentrated on the dialogue control component, therefore we have more knowledge about it. Three main notations have been developed for this component. They are recursive transition networks, context free grammars, and events. The following sub-sections describe these notations and their descriptive power.

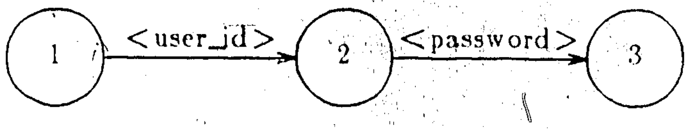
2.6.2.1. Recursive Transition Network Notation

A recursive transition network (RTN) consists of a number of transition diagrams. Each transition diagram has a collection of nodes connected by labeled arcs. Each node represents a state of the dialogue, and each arc represents an action or a class of actions that can be performed by the user.

The user interface will move from one state to another as the user interacts with the system. At the beginning of a session, the transition diagram is placed at a special state called the start state. When the user performs an action, the user interface enters a new state by traversing the arc labeled by the user's action. In a given state the user can only perform one of the actions that labels an arc leaving the node representing that state. All other actions are treated as errors. At any one time only one state will be entered.

An arc label can be either the name of a token or the name of a sub-diagram. When an arc is traversed the token associated with the arc may be sent to the application interface model to retrieve some data, or it may be sent to the presentation component to display some messages on the screen. Because most of the transition diagrams for real user interface tend to have large number of states, the sub-diagram approach has been developed for partitioning the networks. A sub-diagram has its own set of states and transitions that describe one part of the user interface. If the arc label is the name of a sub-diagram, the sub-diagram will be traversed from its start state to a terminal state when the arc is entered. In the case of recursive transition networks a sub-diagram can reference itself. The use of sub-diagrams not only facilitates the description of large user interfaces, but also increases the descriptive power of the technique.

A sub-diagram may also be referenced by attaching it to a node instead of to an arc in a transition diagram. In this case, when the user interface enters the node, the sub-diagram is traversed.



Actions:
1) print 'login:'
2) print 'password:'
3) print 'login message ...'

Figure 2.7 Transition Diagram for Login Sequence

Figure 2.7 shows the use of a transition network to describe a simple login sequence for a time sharing system. In state 1, the user is prompted for his login identification. After the identification has been entered, the user interface enters state 2, and the user is prompted for his password. Control is transfered to state 3 after the user typed in the password.

The main advantage of this notation is that the information in the transition diagram is easy to grasp and understand. A major disadvantage with transition diagrams is they only describe half the dialogue between the user and the program. They can describe the actions performed by the user easily, but say nothing about the actions generated by the application routines. Another disadvantage is that this notation often leads to large transition diagrams which are hard to construct and manage. It is also difficult for the designer to specify global help and cancel commands using this notation.

An example of UIMS that uses recursive transition network to specify the dialogue control component is the SYNICS2 system [Guest and Edmonds 84].

2.6.2.2. Context Free Grammar Notation

The context free grammar or BNF notation uses techniques from programming languages to describe and implement the dialogue control component. A grammar consists of a number of terminals and non-terminals. The terminals are the input tokens generated by the presentation component. The non-terminals and productions are used to structure the dialogue. The productions with these non-terminals on the left side define the syntax of the commands in the user interface. A BNF grammar for the login example is shown in Figure 2.8.

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

Figure 2.8 BNF Grammar for The Login Sequence

One of the main advantages of grammars is their familiarity and the existence of algorithms to produce efficient parsers. This notation also suffers the same disadvantage as the transition network notation in that it only describes the actions performed by the user, and says nothing about the response from the user interface [Green 84d, 85a]. This can be resolved by attaching tokens to the productions. Whenever a production is used in parsing the user's input, these tokens are sent to the appropriate component. However, the order in which the tokens are sent depends on whether a top-down or bottom-up parse is used. Hence, a description of the dialogue control component using this notation is dependent upon a particular UIMS. Another way to resolve the problem is to use another grammar to describe the dialogue produced by the user interface. One of the main problems with this solution is linking the two grammars together so that correct response is generated for all interactions.

Some examples of UIMSs based on context free grammars are SYNICS [Edmonds and Guest 78] and SYNGRAPH [Olsen and Dempsey 83a, 83b]. The studies of Edmonds and Guest [Edmonds 81, Guest 82], have found that the recursive transition network based system, SYNICS2, is more appealing to most designers than the grammar based system, SYNICS.

2.6.2.3. Event Notation

The event notation is not as highly developed as the other two groups of dialogue control notations. This notation is loosely based on the object oriented approach to user interface design as in Smalltalk [Goldberg and Robson 83]. The event notations are based on the concepts of events, and special procedures called event handlers. Input tokens from the presentation component and output tokens from the application interface model are viewed as events processed by the event handlers. Each event handler consists of a collection of local variables and a set of statements for processing events. When an event handler receives an event the associated statements are executed. These statements can perform some computation, create new event handlers, destroy existing event handlers, send events to other event handlers, and send tokens to the presentation component and the application interface model. At any one point in time more than one event handler can be active in the dialogue control component.

Figure 2.9 shows an event handler for the login example. This event handler is capable of processing two types of events. The "INIT" event is sent to the event handler when it is created, so the login prompt is displayed at the beginning. The other event is received whenever the user enters a character string. The "state" variable is used to determine whether the character string is a user identification or a password.

```
eventhandler login is

 token
  keyboardstring s;

 var
  int state = 0;
  string user_id, password;

 event INIT {
  display ( login_prompt ) ;
 }

 event s : string {
  if(state == 0) {
    user_id = s;
    state = 1;
    display ( passwd_prompt ) ;
  } else {
    password = s;
    state = 0;
    process_login ( user_id, password );
  }
 }

end login;
```

Figure 2.9 Event Handler for The Login Sequence

The major advantage of the event notation is that it supports multi-threaded dialogues. By providing each event handler with its own state and allowing multiple event handlers to be active at any one time, the user is free to switch to different parts of the dialogue without explicitly saving the state of the dialogue or completing the current command. This enables the event handlers to process help, cancel, escape and other special commands that must always be available. The event handlers processing these commands are active at all times so that they will always be available when requested by the user. Another advantage is that the event notation can specify a complete dialogue sequence using a set of input and output tokens whereas the context free grammar and recursive transition network notations can only describe half the sequence. A disadvantage to the event notation is that it looks more like a program than the other two notations. This is due to the procedural nature of the event handlers.

An example of UIMS based on the event notation is the MENULAY system [Buxton et al. 83].

2.6.3. Application Interface Model

The application interface model is a representation of the functionality of the application. It describes the application from the viewpoint of the user interface and the user. The description of this component can be divided into two parts.

The first part describes the application's data structures at an abstract level. It might describe the type of data stored, but not how the data is structured or implemented. The description might include the names and the parameters of the application routines that can be invoked by the user interface for accessing the data structures. The routine descriptions might contain constraints such as pre-conditions and post-conditions that may allow the user interface to perform some semantic error detection and recovery.

The second part describes the interaction mode used for communication between the user interface and the application. There are three possible modes of communication. In the system initiated mode, the application calls routines in the user interface. This is similar to the internal control UIMS. In the user initiated mode, the user interface calls the application routines. This is similar to the external control model. In the third interaction mode, mixed initiative, the user interface and the application are viewed as separate processes that execute in parallel. Some interprocess communications mechanism is used to pass information between the two processes. In this mode the user interface and the application are treated as equals. That is neither the user interface nor the application has control over the other. The presentation and dialogue control components are independent of the interaction mode.

The application interface model has not explicitly appeared in any existing UIMSs, therefore, there have been no notations developed for it. Much work remains to be done in this area before a classification scheme for its notations can be presented.

Chapter 3

Designing the Dialogue Control Component

In this chapter we discuss the design of the dialogue control component of the University of Alberta UIMS. This UIMS is based on the Seeheim model of user interfaces [Green 84b, 84c, 85a, Seeheim 84] as described in Chapter Two.

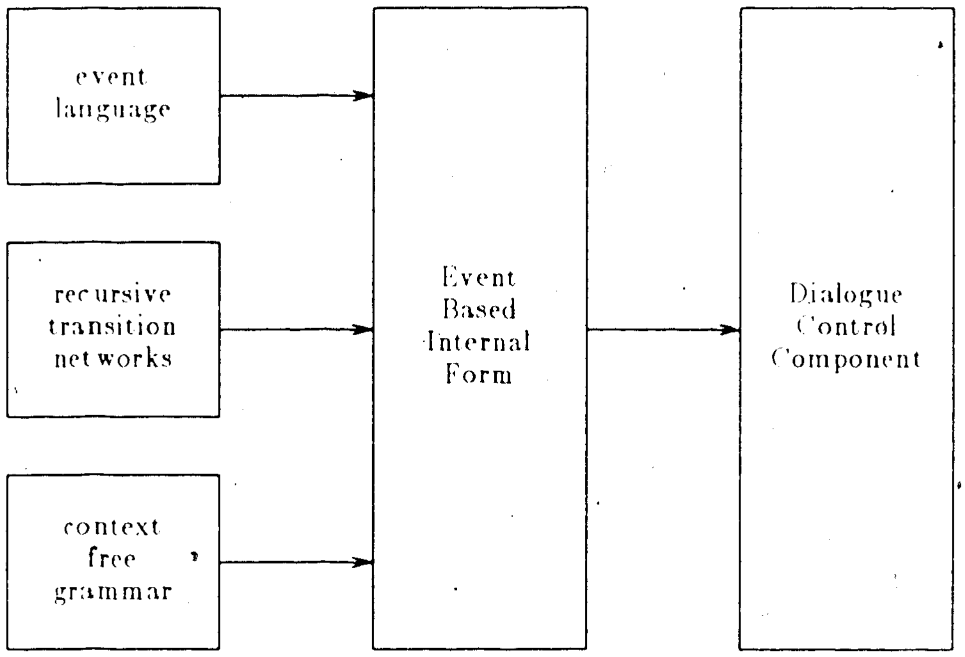

event language
recursive transition networks
context free grammar
Event Based Internal Form
Dialogue Control Component

Figure 3.1 Structure of the Dialogue Control Component

The basic design strategy used in the University of Alberta UIMS is to develop an Event Based Internal Form (EBIF) for the dialogue control component. All the three notations used for the dialogue control component will then be compiled into this common format, as shown in Figure 3.1. The common format allows new notations to be added easily.

In this thesis only the design and implementation of the event model is described. The design of the recursive transition network notation can be found in [Lau 85]. The context free grammar notation has not been implemented in our UIMS. However, the recursive transition network and context free grammar notations used for dialogue control will be compared to the event notation in the final chapter.

3.1. The Event Model

The event model is based on the concept of input events. There are an arbitrary number of event types, new event types can be defined by the user interface designer. When an event is generated, it is added to the end of an event queue. An event scheduler is used to remove an event from the head of the queue and invoke the corresponding event handler to process the event. An event handler is a procedure that is capable of processing certain types of events. When an event handler receives an event, it executes a number of statements that perform some computation, create new event handlers, destroy existing event handlers, generate new events, or send tokens to the presentation component and the application interface model. This is similar to messages and objects in Smalltalk [Goldberg and Robson 83].

In the event model a distinction is made between the definition of an event handler and its instances. This distinction is similar to that between types and variables of that type. At any point in time there may be more than one active instance. An event handler by itself does not process any events. At least one instance of the event handler must be created to process events. Each active instance of an event handler is assigned a unique name at creation time. This name is used to reference the instance when an event is sent to it, or when it is destroyed. The description of an event handler specifies the events it can handle and the statements that are executed in response to these events. Each instance has its own set of local variables that cannot be accessed by any other instances.

In the event model a user interface is described by a set of event handler definitions. At the start of execution an instance of one of these event handlers is created to serve as the main event handler in the user interface. This instance may then create other instances in the user interface. Since each instance has its own set of local variables, conceptually all the instances can execute concurrently, processing the events as they arrive. However, each instance can only handle one event at a time.

3.2. The Event Based Internal Form

The EBIF can be divided into two parts. The first part is a scheduler that sequences the execution of the event handlers. The second part is based on a representation for the event handlers in a file.

3.2.1. Scheduler

The scheduler consists of a number of tables and a set of C procedures. The three main tables in the scheduler are the instance table, the event handler table, and the token table, as shown in Figure 3.2. An entry is created in the instance table for each active instance of the event handlers at run time. Each entry contains a pointer to a local array storing the instance's variables, and the index of the corresponding event handler in the event handler table. The number of entries in the instance table varies as new instances are created and old ones are destroyed. The event handler table contains one entry for each event handler in the system. Each entry contains the number of local variables belonging to the event handler, and a pointer to its C procedure. The token table has entries for all the input and output tokens. Each entry contains an event name corresponding to the token, and the index of the event handler in the event handler table. The event handler table and the token table are static, and they are constructed when the event handler definitions are compiled.

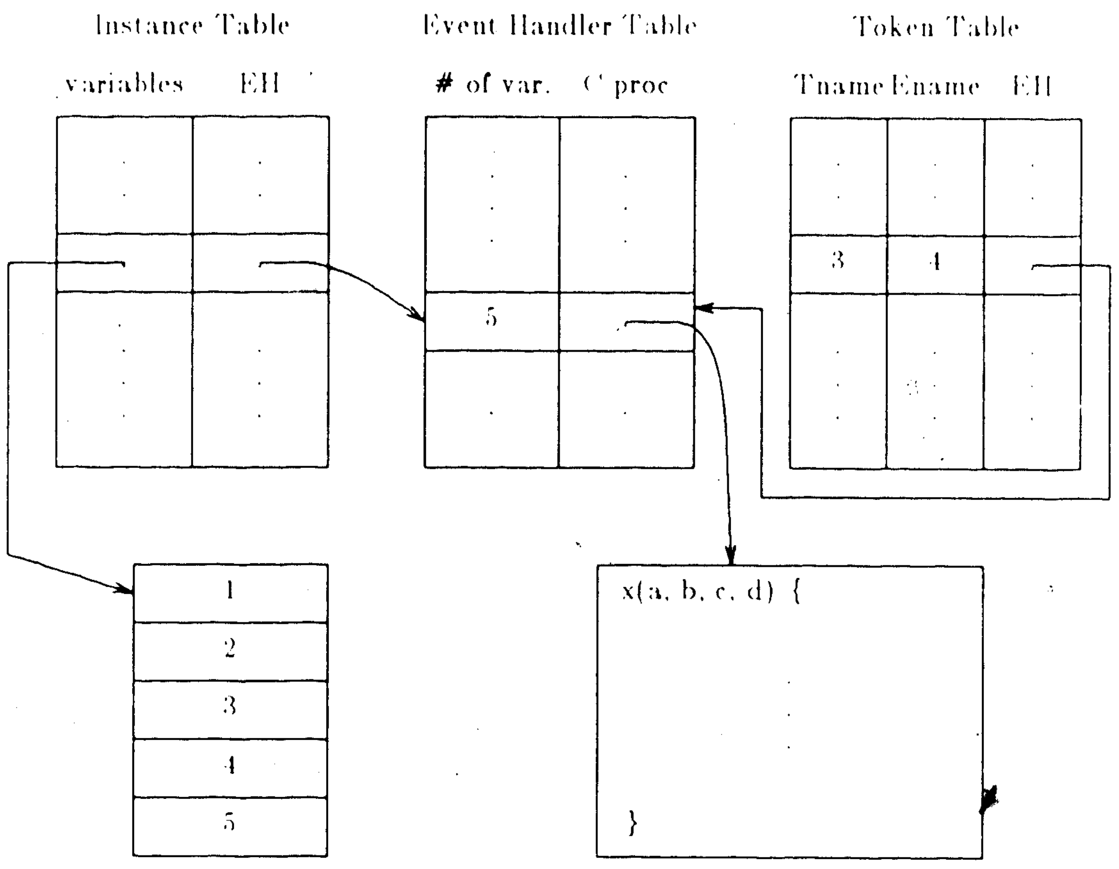


Figure 3.2 Relationships between Scheduler Tables

Some of the C procedures in the scheduler can be called by the event handlers at run time. The create_instance routine is used to create a new instance of an event handler. It creates an entry in the instance table and initializes the contents of this entry. The destroy_instance routine destroys the event handler instance named by its parameter by removing its entry from the instance table. The send_event routine can be called to send an event to an active instance of an event handler. This routine provides a mechanism for all the active instances of the event handlers to communicate with each other within the dialogue control component. The send_token routine is used to send a token to another component of the user interface. The token is used as a communications mechanism among the presentation component, the dialogue control component, and the application interface model.

3.2.2. Event Handler File

A complete event system is assembled from one or more files containing the event handlers for the system. Each file can have one or more event handlers separated by double-percent "%%" marks. Each event handler is divided into three parts by two single-percent "%" marks. Figure 3.3 shows the format of an event handler file.

```
Event_handler1
5 3 6
event1 event2 event3 event4 event5 INIT
%
token1  event1
token2  event3
token3  event4
%
event_handler1( inst_name, event_name, event_value, vars )
int inst_name ;
int event_name ;
int event_value ;
int vars[] ;
{
  switch (event_name) {
    case INIT : {
          ...
        }

    case event1 : {
          ...
        }

          .
          .
          .

    case event5 : {
          ...
        }
  }
}
%%
```

Figure 3.3 Format of An Event Handler File

The first part of an event handler provides information for constructing the event handler table and the token table that are used by the scheduler. The first line of each event handler contains a constant that is the same as the name of the event handler except the first letter is complimented (to upper case if it is in lower case, and vice versa). The constant is used by the assembler (see Chapter 5) to reference the event handler in calls to create_instance. The next line contains the number of variables in each event handler, the number of tokens processed by the event handler, and the number of events processed by the event handler. Following this line is a list of the names of the events. These names will be converted to constants by the assembler.

The second part of the event handler contains the entries for the token table. In this part of the event handler, each line defines a mapping of a token onto an event name. The number of tokens mapped onto event names is the same as the number of tokens processed by the event handler described in the first part.

The third part of the event handler is a C procedure that performs the event processing. This procedure has four parameters, the name of the instance, the name of the event to be processed, the value of the event, and the values of the local variables associated with the instance. The event name is an integer and the event value is an integer or a pointer. The body of the procedure is a switch statement with one case label for each of the events that can be processed by the event handler. When the event handler is assembled, the C procedure is copied verbatim to an output file that will be used as input to the C compiler. The resulting object code will be linked with the object code of the scheduling routines to form a complete event system.

Chapter 4

The Event Language

4.1. Design Objectives

The event language is intended to be used by interaction designers for describing the dialogue control component of the user interface in terms of a high level specification. The design objectives for this language are:

1. Parallel processing
2. Ease of learning
3. Ease of programming
4. Device independence

Today parallelism has been recognized as a desirable feature for programming languages. This is because an important characteristic of modern interactive systems is the ability to handle multi-threaded dialogues. People are inherently multi-threaded, being capable of maintaining several dialogues with rapid switching between them. Current graphics standards allow for multiple threads of control within the graphics system itself, by the concepts of SAMPLE and EVENT modes. This allows several input devices to be active simultaneously, either by reporting their values when polled (SAMPLE) or when triggered by the user (EVENT). Hence, to exploit the power of the system, we must design new programming languages with features that permit the use of parallel processing.

One valuable feature of a good language is ease of learning [Horowitz 84]. Simplicity in the language design is an important strategy that can be used to reach this goal. An interaction designer who fully understands his tool can tackle more complex jobs and complete them more reliably and efficiently. However, the language must be powerful enough to enable him to describe real-world applications in an easy manner.

It has been noted [Green 81a, 81b] that good user interfaces are hard to program, therefore, a language should free designers from worrying about the low level detail of the user interfaces. The language should enable the designers to concentrate on the general flow of the user interfaces and produce better quality human-computer dialogues. By allowing the designers to give a high level specification of a user interface, it will also be easier for him to modify the dialogue sequences as the needs of the users change over time.

An important goal in the design of high level languages is the ability to move programs from machine to machine [Green 79, Horowitz 84]. If a graphics system has a number of possible input and output devices, then the programming effort required to support all combinations of these devices will be minimized when we use a language that has a high degree of mobility. However, it has been realized that this goal is terribly difficult to achieve, but it continues to be worth striving for.

4.2. Introduction to the Event Language

The event language is an extension of the C programming language [Kernighan and Ritchie 78]. The main reason for basing the design of the event language on C is because all our graphics programs are written in C. The C programming language also satisfies all our design objectives for the event language except for parallel processing. C offers only single-threaded control flow constructions such as looping and condition testing.

A program in the event language consists of one or more event handler declarations. These event handlers are compiled by the event compiler into the Event Based Internal Form (see chapter 3), which in turn is assembled by an assembler (see chapter 5) to produce the object code for the event system. Figure 4.1 illustrates the structure of an event handler declaration. Notice how the keywords *eventhandler*, *is*, and *end* form a frame for the event handler description. The name of the event handler is "sample". The body of an event handler is divided into three sections: token, variable, and event declarations. Comments may appear whenever a name is legal; they are enclosed in /* ... */, as in C.

```
eventhandler sample is

  token          /* token declaration */
    token_name  event_name ;
         .
         .
         .

  var            /* variable declaration */
    type var_name = initial_value ;
         .
         .
         .

  event event_name: type {     /* event declaration */
    statements
  }
         .
         .
         .

  event event_name: type {
    statements
  }

end sample ;
```

Figure 4.1 Structure of An Event Handler Declaration

4.2.1. Token Declaration

The keyword *token* is used to declare the beginning of the first section of the event handler. This section presents a list of input and output tokens that can be processed by the event handler. This information is used by the assembler to build the token table that defines the mapping of tokens onto event names. This feature supports the localization of information and ease of modification. For example, the assignment of token names, and the mapping between tokens and events can be changed in the assembly process without affecting the event handlers themselves. In this way, the event handler can be changed easily to handle a different set of tokens. Each pair of token-event mapping in the declaration is separated by a semicolon. For example:

```
token1 event1 ;
token2 event2 ;
```

The token declaration can be omitted if the event handler does not process any input or output tokens. In this case the event handler may only process events generated by other event handlers.

4.2.2. Variable Declaration

The second section of an event handler declaration contains the declarations of the event handler's local variables. A variable declaration consists of a type and a list of one or more variables of that type, as in

```
var
 int a, b, c;
 char d, *name ;
```

The type can be any valid C type that occupies the same amount of space as a pointer. This includes character, integers, floating point numbers, and pointers to any C type.

Variables may also be initialized in their declarations, except for arrays and structures. This restriction simplifies the implementation of the event language and may be lifted in the future. The initializer is preceded by "=", and consists of an expression which evaluates to a constant. The expression may involve constants, and previously declared variables and functions. Variables which are not initialized are guaranteed to start off as zero. The variable section can also be omitted if no local variables are used in the event handler. Some examples of variable declarations with initial values are:

```
var
 int state = 0 ;
 int i = 1 ;
 int p = i ;
```

Each instance of the event handler has its own set of local variables that are not visible to other instances. These local variables are used to retain the state of the dialogue. This enables the system to handle multi-threaded dialogues.

4.2.3. Event Declaration

The third section of an event handler declaration contains the event declarations. An event declaration starts with the keyword *event* followed by the name of the event, its type, and one or more C statements. The type declaration here does not define any new data type, but only provides a convenient way of identifying an event type. The type declaration is optional. The main body of the event declaration is composed of C statements that are executed when an instance of the event handler receives this event. These statements can perform arithmetic operations, control the flow of execution, create or destroy instances, generate new events, and send tokens to the presentation component and the application interface model. They can also reference both the instance's local variables, and global variables that are declared in the other parts of the program. For example, to specify that a box is drawn at the position selected by a user on the terminal screen:

```
event position_select: point {
   if ( state == draw )
      send_token( PRESENTATION, OUTPUT, drawbox, event_value );
}
```

When the dialogue control component receives the event "position_select", the output token "drawbox" is sent to the presentation component which then handles the display. In this example, the event "position_select" is explicitly identified as a "point" type whose "event_value" contains the coordinates of the position selected by the user.

There is a special event, called INIT, that can be used to initialize variables or perform some computation when an instance of the event handler is created. This is useful especially for prompting a user for sign on messages. If the INIT event is declared by the designer, the INIT event must be the first event following the variable declarations. This restriction turns out to have an advantage. It encourages the concept of structured programming since it is a good idea to group all initialization statements at the beginning of a program instead of being scattered all over the program. It is similar to imposing the declaration of variables at the beginning of a function in C. It also makes the compiler easier to write.

4.3. Multi-Threaded Dialogues

The main difference between C and the event language is that the event language supports multi-threaded dialogues. The illusion of multi-processing is achieved by providing each active instance of an event handler with its own array of local variables. When an event handler declaration is compiled, each variable declared in the event handler is converted into an element of the array. For example, the declaration

```
var
  int i = 1 ;
  int state ;
  char ch ;
  node a = NULL ;
```

is converted into C statements that make up the body of the INIT event, in EBIF:

```
case INIT : {
  var[0] = 1 ;
  var[3] = NULL ;
}
```

Since the variables "state" and "ch" are not explicitly initialized, their corresponding entries, var[1] and var[2], in the system created array do not appear in the compiled output of the INIT event. The mapping of the variables into elements of the array is stored in the system's variable table. All the variables used in the declarations of event handlers are converted to their corresponding elements of the array using this mapping.

The event compiler does not allocate actual memory to the array at compile time. The storage allocation is performed when an instance of the event handler is created at run time by calling the create_instance routine. After an entry in the instance table of the scheduler has been set up for the newly created instance, the create_instance routine will send the INIT event to initialize the elements of the array.

The relationship between an event handler and its instances is similar to that between a type and variables of that type. Therefore, there may be multiple instances of an event handler at the same time. In addition, several instances of different event handlers can be active at any state of a dialogue. The local array associated with an instance cannot be accessed by any other instances. The array exists until the instance is destroyed. Hence, the state of the dialogue at any point in time is retained in these arrays. This allows the execution of different instances of the event handlers to be interleaved. The user can switch to different spots in a dialogue without explicitly saving the state of the dialogue. Thus, the user interface can process help, cancel, and other special commands that must always be available.

4.4. Summary

In this section we discuss several factors that have been considered in the design of the event language. The strengths and weaknesses of the language are also identified during the discussion.

The language supports multi-threaded dialogues using the concept of events. The illusion of multi-processing is maintained by creating an array of local variables for each active instance of the event handlers, and allowing several instances to be active at the same time. The state of the dialogue at any point in time is retained in these arrays, which exist until their associated instances are destroyed. This allow their execution to be interleaved.

The time required to learn the event language should be minimized. This is a very important factor that often determines the success or failure of a language. The strategy used to reach this goal is to incorporate a large number of features that are already familiar to interaction designers into the event language. This will not only reduce the period of re-training for the designers, but also make them feel more comfortable with the new tool.

On the other hand we would like the event language to be flexible in order to specify various types of interaction dialogues. It should be possible to specify new interaction techniques without affecting the existing specification. Therefore, dialogue control is divided into different event handlers which are independent software modules. This modularization makes it possible to add new interaction techniques by adding new events, or new event handlers.

Programs written in the event language are also portable. This is because the event language is an extension of C which has proven to be highly machine independent [Kernighan and Ritchie 78, Stroustrup 84]. C does not have special facilities for handling input and output. The approach taken in C is to provide functions like printf() and scanf() in a "standard" library, but not in the language itself.

The main disadvantage of an event handler declaration written in the event language is that it resembles a program. The body of an event handler is similar to the switch statement in C. As in C, the event language is not strongly typed, consequently, violations of strong typing and other potential errors, such as using uninitialized variables, cannot always be detected.

In this thesis we have described the structures of the event language and pointed out some of its features that are different from the C programming language. Although the event language is based on C, no attempt has been made to describe the basic features of C. It is assumed that the interaction designers already have some programming experience in C. This assumption is based on the fact that C has been extensively used in the graphics community. The complete description of C can be found in [Kernighan and Ritchie 78].

Chapter 5

Implementation

In this chapter we discuss the implementation of the event language. This implementation includes the construction of the compiler, and the scheduler routines. An overview of the scheduling process in the University of Alberta UIMS is also presented.

5.1. The Compiler

This section discusses the implementation of the event compiler. The implementation can be divided into two parts. The first part is to specify the syntax of the event language and to build a parser. The second part is to construct the lexical analyzer used for translating a source program into a sequence of tokens that are used as input to the parser. The supporting tools used for automatic generation of the parser and the lexical analyzer are also briefly described.

5.1.1. Syntactic Specification

The *syntax* of a language is the way that words and symbols are combined to form the statements and expressions. It is a set of rules which determines whether the statements are well-formed or not. The syntactic specification of the event language is based on a context free grammar notation, which is also sometimes called BNF (Backus Naur Form). As noted in [Aho and Ullman 77], this notation has a number of significant advantages as a method of specifying the syntax of a language:

1. A grammar gives a precise syntactic specification for the programs of a particular programming language.

2. A grammar imparts a structure to a program that is useful for its translation into object code and for the detection of errors.

3. An efficient parser can be constructed *automatically* from a properly designed grammar.

One such well known parser generator is called YACC (Yet Another Compiler-Compiler) [Johnson 83]. YACC accepts a very general LALR (lookahead-LR) grammar with disambiguating rules and produces a bottom-up parser that can detect syntactic errors as soon as possible on a left-to-right scan of the input [Aho and Ullman 77]. Hence, the BNF description of the event language is used as input to YACC in order to generate the event compiler, which is also in C code.

To begin, let us look at a BNF description of the event declaration in the event language, as shown in Figure 5.1 (using YACC notation):

```
ev_body   : ev_decl
          |ev_body  ev_decl
          ;
ev_decl   : EVENT IDENTIFIER cmpd_stat
          |EVENT IDENTIFIER ':' IDENTIFIER cmpd_stat
          ;
cmpd_stat : '{'  stat_list  '}'
          ;
```

Figure 5.1 BNF Description of Event Declaration

The symbols :, ;, and | are not part of the event language which is being defined, but are part of the mechanism for describing the language. Using YACC convention, all capital letters denote token names, and all lower case letters denote non-terminals which are used to represent sequences of symbols. The vertical bars are used to separate alternatives given in a production. The algorithm used by the YACC parser encourages so called *left-recursion* grammar rules, such as the second alternative of the first production in Figure 5.1.

An important and difficult area in the design of a programming language is error handling. What should the compiler do in case an error is found in a parsing process? It is seldom acceptable to stop all processing when an error is encountered; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of restarting the parser after an error. YACC provides a special token "error" to solve the problem. The "error" token can be placed in the grammar rule where errors are expected. Usually a message is printed when a syntax error is found.

5.1.2. Lexical Analysis

The second part in the implementation of the event language is to construct the lexical analyzer that reads the input stream, one character at a time, and translates it into a sequence of primitive units called tokens. Keywords, identifiers, and constants are examples of tokens. The lexical analyzer passes these tokens to the parser. Fortunately, a language, LEX (A Lexical Analyzer Generator) [Lesk and Schmidt 83], has been provided to interface with YACC in just this way. The output of LEX is a lexical analyzer program that is used to partition the input stream.

A LEX source program is a specification of a lexical analyzer, consisting of a set of regular expressions together with an action for each regular expression. The action is a piece of code, written by the user, which is to be executed whenever a token specified by the corresponding regular expression is recognized. Typically, an action will pass an indication of the token found to the parser, perhaps with side effects such as entering an identifier in the variable table, or counting the occurrences of some tokens. A regular expression specifies a set of strings to be matched. It contains text characters and operator characters. Thus the regular expression

```
[A-Za-z][A-Za-z0-9_]*
```

indicates all alphanumeric strings and underscore with a leading alphabetic character. The operator * means "zero or more instances" of the expression enclosed by the operator pair []. The output of LEX is the lexical analyzer program constructed from the LEX source specification.

5.1.3. Compilation Process

There are three tables, in the compiler, which are used in generating the event based internal form (EBIF) from an event handler declaration. They are the token table, the variable table, and the event table. The token table contains the mapping between tokens and events defined in the token section of the event handler declaration. In the token declaration, each pair of this mapping is unique. A semantic error is reported by the compiler if a token is mapped onto more than one event name.

The variable table is used to map each variable declared in the event handler into an entry in the array associated with each instance of the event handler. In EBIF, each variable used in the body of an event declaration is converted into an element of the array using this mapping. The event compiler does not allocate actual storage to this array. The storage allocation is performed when an instance of the event handler is created at run time by calling the create_instance routine.

The event table stores all the events that can be processed by the event handler. Each event declaration is translated into a case label in a switch statement in the EBIF. It is a semantic error to define two events having the same name since all the cases in the switch statement must be unique. The C statements that make up the body of the event declaration are passed to the parser and then to the assembler. The output of the assembler is used as input to the C compiler. Any syntax errors found in the C statements will be reported by the C compiler. However, after the errors have been corrected, the event handler declaration has to be recompiled by the event compiler, reassembled and then recompiled by the C compiler. Therefore, in order to reduce the recompilation process, when the C statements are passed to the event compiler, all syntax errors are reported so that they can be corrected as soon as possible. Now, the recompilation process only involves the event compiler. When the C statements are passed to the event compiler, the data associated with an event is also assigned to the event name.

When an event handler declaration is compiled, the number of variables, tokens and events used in each event handler are recorded. This information is used by the assembler to construct the scheduler tables.

There is always an INIT event in the compiled event handler file. The INIT event is added to the output by the compiler in order to initialize the array associated with each instance of the event handler. The INIT event is sent by the create_instance routine after the instance has been created. If an interaction designer also declares an INIT event, the statements given by the designer in the body of the declaration are appended to the compiler generated statements that are used to initialize the local variables in the event handler.

5.1.4. Keywords

There are certain words that can not be used for names or identifiers. These are called reserved words or keywords. A list of these keywords follows.

break	case	char	continue	default
do	double	else	end	entry
event	eventhandler	float	for	goto
if	int	is	long	return
short	sizeof	struct	switch	token
typedef	union	unsigned	var	while

Note that the following storage class specifiers used in C have been omitted in the event language:

auto   extern   register   static

The reason for omitting them from the variable section of an event handler is that when an instance of the event handler is created, each variable is converted into an element of a local array, and the array exists until the instance is destroyed. Therefore, all the variables declared are equivalent to static variables in C. As a result, these keywords do not make sense in this context, thus, they have been excluded from the event language.

5.1.5. Summary

The implementation of the event compiler using YACC and LEX has at least two advantages. The first advantage is that the implementation is automated. Therefore, the time taken to implement the compiler has been greatly reduced and the resulting parser is also more reliable than hand coded parser. Using these software tools, one does not have to be a compiler expert in order to construct a compiler. The second advantage is that separating the design of the compiler into lexical and syntactic modules not only increase the ease of design, but also increase the ease of modification.

One problem with programs written in the event language is that there is no easy way for the event compiler to check for errors in statements calling the application programs. The errors, such as undefined identifiers, can only be detected by the C compiler when all the programs for the UIMS are linked together. Therefore, when these errors are found, the designers will need to make all necessary corrections in the event program, recompile it using the event compiler, and then link it with the other programs again. The event language enables the recompilation time to be reduced by allowing different parts of a single program to be compiled separately. In this case, when there are errors found at linkage time, only those modules containing the errors need to be recompiled. A separate compilation facility also enables different parts of the dialogue control component to be constructed by a few designers working independently. The object code produced by one designer can be used by other designers while constructing other parts of the component. The lack of a separate compilation facility could mean long recompilation for big program when a single change is made to the program.

5.2. Scheduler Routines

The implementation of the scheduler routines that are called by the event handlers is described in this section. The create_instance routine is used to create a new instance of an event handler. It creates an entry in the instance table for each active instance and allocat memory for the local array associated with the instance. A pointer is then set to point to this array, and an integer is set to the index of the corresponding event handler in the event handler table. After this, it sends an "INIT" event to initialize the variables declared in the variable section of the event handler. These variables are initialized in the order listed in the declaration. The parameters to this routine are the name of the event handler, the number of local variables to be initialized, and their initial values. Create_instance returns an integer which is used as the name of the new instance. The header of this routine is

```
int create_instance( event_handler, nvar, local_var )
int event_handler ;
int nvar ;
int *local_var ;
```

The destroy_instance routine destroys the event handler instance that is given as its parameter by removing its entry from the instance table. The syntax of this routine is

```
destroy_instance( instance_name )
int instance_name ;
```

The send_event routine is used to send an event to an active instance of an event handler. The parameters to this routine are the name of the instance, the name of the event, and the value of the event. When the send_event routine is called, the event is added to the end of an event queue. The event scheduler is responsible for removing these events and sending them to the appropriate instances of the event handlers. The header of this routine is

```
send_event( instance_name, event_name, event_value )
int instance_name ;
int event_name ;
int event_value ;
```

The send_token routine is used to send a token to another component of the user interface. The token provides a mechanism for communication between the presentation component, the dialogue control component, and the application interface model. The parameters to send_token are the name of the component receiving the token (PRESENTATION, DIALOGUE, or APPLICATION), the direction of the token (INPUT, or OUTPUT), the name of the token and its value. The token is added to the end of the token queue that belongs to the destination component. Each of the three components has its own token queue. The header of this routine is

```
send_token( destn, dir, token_name, token_value )
int destn ;
int dir ;
int token_name ;
int token_value ;
```

5.3. Process Scheduling

This section describes the programming environment of the University of Alberta UIMS. The scheduling of events and tokens in the UIMS at run time is also presented.

5.3.1. Environment

The University of Alberta UIMS is written in the C programming language for the UNIX operating system. The implementation is on a VAX 11/780. The initial devices supported by the UIMS consist of ASCII terminals, Jupitor graphics terminals, and VT125 graphics terminals. Device interactions are handled by the WINDLIB graphics package [Green 85b] that is capable of handling multiple input and output devices.

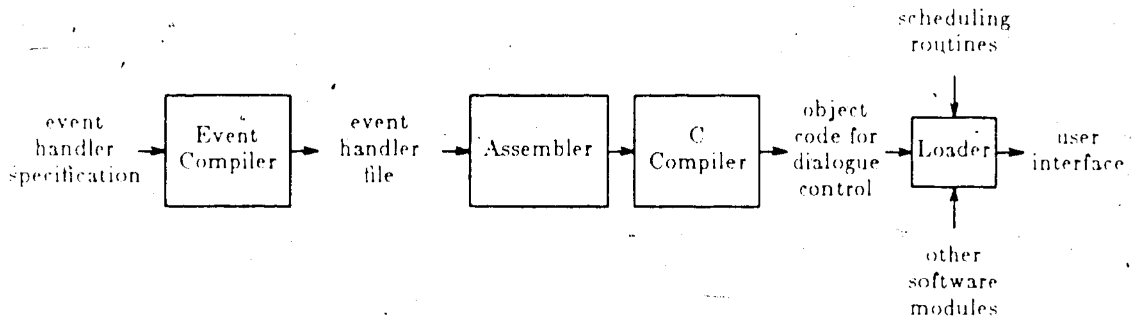


Figure 5.2 User Interface Generation Process

The process of generating a user interface using the event language is shown in Figure 5.2. The event handler specification for the dialogue control component, is compiled into an event handler file which is in turn passed to the assembler. The assembler will use the information in the event handler file to construct the token table and the event handler table. It will check that each of the tokens used in the UIMS has a definition. The assembler also produces a constant for each event handler. This constant is used to reference the event handler in calls to create_instance. The result of the assembly process will be one file that is used as input to the C compiler. The resulting object file for the dialogue control component is combined with the object code of the scheduling routines, other software modules, such as the applications routines, and the programs from the presentation component and the application interface model, to form a complete user interface.

At the start of execution, an instance of one of the event handlers is created to serve as the main event handler in the user interface. This instance may create other instances in the user interface.

5.3.2. Scheduling

In the design of the University of Alberta UIMS it has been assumed that the presentation component, the dialogue control component, and the application interface model are separate processes. The only communication mechanism among the three components is through tokens using the send_token routine. The simulation of concurrency, which is asynchronous, is maintained by a scheduler that allocates processing time to the individual components. The unit of scheduling at the component level is the token.

Each of the components in the UIMS has a scheduling queue associated with it. When one component sends a token to another component, the token is added to the scheduling queue associated with the receiving component. The scheduler will examine the scheduling queues according to some pre-defined priority. Highest priority is placed on the presentation component's queue so that lexical feedback to the user can be handled as fast as possible. Lowest priority is placed on the application interface model's queue, therefore, semantic feedback is slower. The scheduler will select tokens from these queues for execution and then invoke the appropriate routine in the receiving component to process the tokens.

The dialogue control component has two levels of scheduling. The high level scheduler removes the tokens from the dialogue control component's queue and then converts them into events. These events are sent to all the active instances of the event handlers associated with them via the send_event routine. The events are placed on an event queue, and the scheduler invokes a *lower level* event scheduler to remove the events. The event scheduler then calls the corresponding event handlers to process the events. These event handlers can be found from the scheduler tables constructed by the assembler. When an event is received by an event handler, the statements associated with the event are executed. These statements may perform some computation, or call the application and the scheduler routines.

Chapter 6

Examples

In this chapter we present some examples of the use of the event language for automatic generation of the dialogue control component. The first example is to construct a simple editor for managing *fixed* size geometrical objects on a terminal screen. The next example is to modify the first example to provide a more flexible editor that can handle *variable* size objects and has help messages which can be activated and deactivated by the user.

6.1. A Simple Geometrical Object Editor

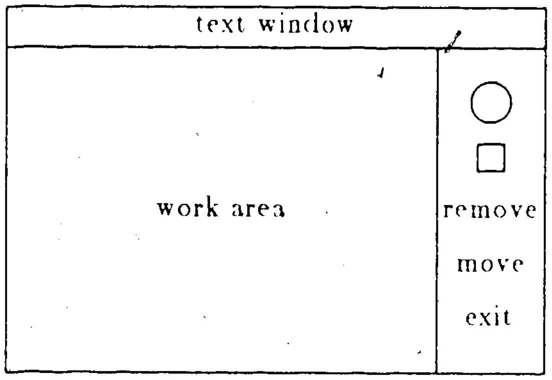


Figure 6.1 Screen Layout for A Simple Geometrical Object Editor

In this section we present the specification of a simple editor, using the event language, that aids the user in arranging a number of simple fixed size geometrical objects on a display screen. The geometrical objects are circles, and squares. There is a menu on the right side of the screen containing the two objects and also three commands as shown in Figure 6.1. The left side of the screen is the work area. There is a text window at the top of the screen for displaying help messages generated by the system.

A digitizing tablet is used to select one of the objects by pointing at the object with the tracking cross and pressing a button on the tablet. The selected object can be placed at any position in the work area by pressing the button on the tablet again. The user can also remove any object in the work area. First, the user selects the "remove" command in the menu and then points at the object to be removed. Objects in the work area can also be moved around. The user selects the "move" command and then points at the object to be moved in the work area. Next the user moves the tracking cross to a new position and presses the button to deposit the object at that position. The final command "exit" terminates the interaction.

Starting with the presentation component, the screen layout is produced through the use of the composite program [Singh 85]. Each of the three screen areas is assigned a different window. The next step is to construct a list of the tokens that flow between the presentation component and the dialogue control component, as shown below:

Input Tokens	Output Tokens
circle	drawcirc
square	drawsq
remove	erasecirc
move	erasesq
exit	
point	

Each of the first five input token names corresponds to an item in the menu. These tokens are generated when the user selects items from the menu. The last input token "point" is generated when a position is picked in the work area. This token has a value that contains the x and y coordinates of the position. All the input tokens will be processed by an event handler, which is yet to be written. The output tokens are sent by the dialogue control component to the presentation component which handles the actual displays.

The next step in the design is to construct an event handler for the dialogue control component. The event handler "control", shown in Figure 6.2, is responsible for processing all the commands for the user by routing appropriate tokens to either the presentation component or the application interface model.

```
eventhandler control is
 token
  point pointE ;
  circle circleE ;
  square squareE ;
  remove removeE ;
  move moveE ;
  exit exitE ;
  erase eraseE ;
 var
  int state = 0 ;
  int object ;
  int shape ;

 event circleE {
  state = 1;
 }
 event squareE {
  state = 2;
 }
 event removeE {
  state = 3;
 }
 event moveE {
  state = 4;
 }
 event exitE {
  stop( );
 }
 event pointE {
  switch(state) {
   case 1  : send_token(PRESENTATION,1,drawcirc,event_value);
           send_token(APPLICATION,1,drawcirc,event_value);
           break;
   case 2  : send_token(PRESENTATION,1,drawsq,event_value);
           send_token(APPLICATION,1,drawsq,event_value);
           break;
   case 3 :  send_token(APPLICATION,1,erase,event_value);
           break;
   case 4 :  send_token(APPLICATION,1,erase,event_value);
           break;
   case 5 :  shape = object->type ;
```

```
                if(shape == 1) {
                  send_token(PRESENTATION,1,erasecirc,object);
                } else if(shape == 2) {
                  send_token(PRESENTATION,1,erasesq,object);
                }
                object = event_value ;
                if(shape == 1) {
                  send_token(PRESENTATION,1,drawcirc,object);
                } else if(shape == 2) {
                  send_token(PRESENTATION,1,drawsq,object);
                }
                state = 4 ;
                break;
        }
      }
      event eraseE {
        if(state == 3) {
          shape = event_value->type ;
          if(shape == 1) {
            send_token(PRESENTATION,1,erasecirc,event_value);
          } else if(shape == 2) {
            send_token(PRESENTATION,1,erasesq,event_value);
          }
        if ( state == 4 ) {
          object = event_value ;
          state = 5 ;
        }
      }

    end control;
```

Figure 6.2 Event Handler for the Simple Geometrical Object Editor

In the token declaration section, each of the input tokens is mapped onto an event name. In the variable section, the variable "state" is used to record which command has been chosen by the user during the interaction. The variable "object" is used as a temporary buffer to store the location of an object. This variable may be sent to the presentation component together with some output tokens. The variable "shape" is used to identify an object type. The "event_value" contains information about an object type and its position on the screen.

To illustrate how the event handler "control" works, let us examine the dialogue sequence when the user selects the circle in the menu using the tablet. In this case the token "circle" is sent to the dialogue control component by the presentation component. This token maps onto the event "circleE" which sets the value of the variable "state" to 1. Next, when the user selects a position in the work area, the token "point" is sent to the dialogue control component. This token maps onto the event "pointE" and causes the output token and its value to be sent back to the presentation component, which then draws a circle at the position indicated by the user. The same token and value are also sent to the application interface model to update the data base. This data base is used when the user wants to move or remove an object in the work area. The dialogue sequence is similar to this when the user selects the square in the menu.

When the user selects an object to be removed, the dialogue control component sends the "erase" token to the application interface model to remove the object from the data base. In return, the application interface model also sends the token "erase" and the location of the object to the dialogue control component which redirects this information to the presentation component where some actions are finally taken to remove the object.

6.2. An Advance Geometrical Object Editor

In this section we modify the example in the previous section in order to produce a more flexible editor that can handle variable size objects. The editor also provides help messages that can be activated and deactivated by the user at any time. The screen layout for the editor is shown in Figure 6.3. In the menu, there is two addition commands that can be used to activate and deactivate the help messages.

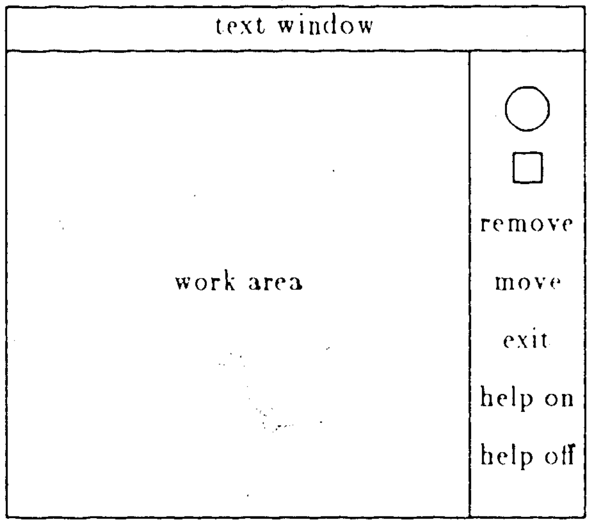


Figure 6.3 Screen Layout for An Advance Geometrical Object Editor

As in the previous example, a digitizing tablet is used to select an item in the menu. However, now to draw a circle in the work area, the user has to specify the center of the circle and a point on its circumference. To draw a square, the user has to specify two diagonal points in the work area to represent the opposite corners of the square. The information on how to draw a circle can be obtained by selecting the "help on" command following by the "circle" command. Similarly, the user can also obtain information on how to draw a square, and to move or remove an object in the work area. The "help off" command is used to exit from the help mode.

A list of input and output tokens that flow between the presentation component and the dialogue control component is shown below:

Input Tokens	Output Tokens
circle	drawcirc
square	drawsq
remove	erasecirc
move	erasesq
exit	
point	
helpon	
helpoff	

The description of the dialogue control component for this user interface is divided into two event handlers. One event handler is responsible for all command processing, and the other event handler displays the help messages. The new "control" event handler, shown in Figure 6.4, is responsible for all the commands for the user, and activating the other event handler.

```
eventhandler control is
 token
  point  pointE ;
  circle circleE ;
  square squareE ;
  remove removeE ;
  move moveE ;
  exit exitE ;
  erase eraseE ;
  helpon helponE ;
  helpoff helpoffE ;
 var
  int state = 0 ;
  int object ;
  int shape ;
  int pos ;
  int point  = 1 ;  /* used to ensure that two points are entered
                       on the work area to draw an object */
  int hh = -1 ;    /* index of an instance of the "help" event
                       handler in the Instance Table */

 event circleE {
  if ( hh = = -1 )
   state = 1;
```

```
}
event squareE {
 if ( hh == -1 )
   state = 2;
}
event removeE {
 if ( hh == -1 )
   state = 3;
}
event moveE {
 if ( hh == -1 )
   state = 4;
}
event exitE {
 if ( hh == -1 )
   stop();
}
event helponE {
 if ( hh == -1 ) {

   /* create an instance of the "help" event handler */
   hh = create_instance( help, 0, NULL ) ;  /* hh >= 0 */
   send_event( hh, helponE, 0 ) ;
 }
}
event helpoffE {
 if ( hh != -1 ) {
    destroy_instance( hh ) ;
    hh = -1 ;
 }
}
event pointE {
 switch(state) {
   case 1  : if ( point == 1 ) {
               pos.x1 = event_value->x ;
               pos.y1 = event_value->y ;
               point = 2 ;
            else {
               pos.x2 = event_value->x ;
               pos.y2 = event_value->y ;
               send_token(PRESENTATION,1,drawcirc,pos);
               send_token(APPLICATION,1,drawcirc,pos);
               point = 1 ;
            }
            break;
   case 2  : if ( point == 1 ) {
               pos.x1 = event_value->x ;
               pos.y1 = event_value->y ;
               point = 2 ;
            else {
               pos.x2 = event_value->x ;
               pos.y2 = event_value->y ;
```

```
                send_token(PRESENTATION,1,drawsq,pos);
                send_token(APPLICATION,1,drawsq,pos);
                point = 1;
              }
              break;
        case 3 : send_token(APPLICATION,1,erase,event_value);
              break;
        case 4 : send_token(APPLICATION,1,erase,event_value);
              break;
        case 5 : shape = object->type ;
              if(shape == 1) {
                send_token(PRESENTATION,1,erasecirc,object);
              } else if(shape == 2) {
                send_token(PRESENTATION,1,erasesq,object);
              }
              object = event_value ;
              if(shape == 1) {
                send_token(PRESENTATION,1,drawcirc,object);
              } else if(shape == 2) {
                send_token(PRESENTATION,1,drawsq,object);
              }
              state = 4 ;
              break;
    }
  }
  event eraseE {
    if(state == 3) {
      shape = event_value->type ;
      if(shape == 1) {
        send_token(PRESENTATION,1,erasecirc,event_value);
      } else if(shape == 2) {
        send_token(PRESENTATION,1,erasesq,event_value);
      }
      if ( state == 4 ) {
        object = event_value ;
        state = 5 ;
    }
  }

end control;
```

Figure 6.4. Event Handler for the Advance Geometrical Object Editor

The "help" event handler, shown in Figure 6.5, is responsible for displaying the help messages for each command in the menu. This event handler is deactivated when the user selects the "help off" command.

```
eventhandler help is
 token
  circle circleE ;
  square squareE ;
  remove removeE ;
  move moveE ;
  exit exitE ;
  helpon helponE ;

 event circleE {
  display_mesg( circle_cmd ) ;
 }
 event squareE {
  display_mesg( square_cmd ) ;
 }
 event removeE {
  display_mesg( remove_cmd ) ;
 }
 event moveE {
  display_mesg( move_cmd ) ;
 }
 event exitE {
  display_mesg( exit_cmd ) ;
 }
 event helponE {
  display_mesg( help_cmd ) ;
 }

end help ;
```

Figure 6.5 Event Handler for Processing Help Messages

In this example the user interface is capable of handling multi-threaded dialogues. The user interface allows an instance of both the "control" and "help" event handlers to be active at the same time. This enables the user to give incomplete commands and switch to different spots in the dialogue. For example, if the user has selected the "square" command in the menu and has forgotten how to draw a square in the work area, he can select the "help on" command and then the "square" command again to display the required help messages. Once in the "help command mode", the user can also display messages on the other commands. To get out of the "help command mode", the user only needs to select the "help off" command. After this, the user can immediately draws the square *without* having to select the "square" command again. The state of the dialogue prior to entering the "help command mode" is saved by the user interface. This explains why the user can give an incomplete draw square command and return to the same state later on. This is the most important feature of the event language and it cannot be achieved by the transition network notation and the context free grammar notation.

6.3. Comments

In the above examples, after the editors have been constructed, they can be modified easily to suit different users. For example, the menu window can be moved to the left side of the screen to accommodate left-handed users by modifying the presentation component only. The description of the dialogue control component and the definition of the input and output tokens need not be changed.

In the second example, if the user controlled help messages are not required, the screen layout for the first example can be used without any modification. Only the description for the dialogue control component needs to be modified. Thus, the separation of a user interface into separate components simplifies the modification of the user interface in the future.

The above simple examples illustrate how to use the event language to give a high level description of the dialogue control component. It has also been shown that the event language is capable of supporting multi-threaded dialogues. This feature is very important in modern interactive systems.

Chapter 7

Conclusions

7.1. Summary of Thesis Contributions

In this thesis a discussion of the University of Alberta UIMS along with the design and implementation of the event language have been presented. One of the main difficulties in developing better techniques for evaluating the performance of user interfaces is our lack of understanding of human-computer interaction. This thesis contributes in a small way to achieving a better understanding of how people respond to user interfaces. The event language serves as a vehicle for the experimentation with the interfaces.

The event language can also be used as a tool in the design and implementation of user interfaces for solving real application problems. Many concepts of modern programming language have been incorporated into the event language. The language supports concurrency, modularity, and allows a natural top-down structured design. These features of the language will help a designer to do his works systematically.

7.2. Evaluation

In this section we discuss the usefulness of the UIMS and the event language, and look at what has been learned from them.

The separation of the UIMS into three components allows an interaction designer to concentrate on the form of the computer dialogue, and the human factors of the interaction. The UIMS removes the burden of physical interaction handling from the designer, so that he can put more effort into the design of easy to use rather than easy to implement dialogue sequences. A prototype dialogue can be developed quickly to test his design at an early stage. The separation also provides increased consistency across applications.

An important feature of the event language is that its expressive power is greater than the recursive transition networks [Newman 68, Lau 85] and the context free grammars [Edmonds and Guest 78, Olsen and Dempsey 83a, 83b]. The event language is capable of supporting multi-threaded dialogues. The user can give incomplete commands, and is free to switch to different spots in a dialogue without explicitly saving the state of the dialogue. This enables the user interface to process help, cancel, undo, escape, and other special commands. On the other hand, a recursive transition network editor can only associate a single function with an action, and offers no direct method of attaching semantic functions to groups of actions, such as conditional statements. In addition, the event language can specify a complete dialogue sequence between the user and the program, whereas both the recursive transition network and context free grammars can only describe half the sequence. They just describe the actions (input tokens) performed by the user, but say nothing about the actions generated by the user interface. In the case of context free grammars, another grammar is needed to describe the output tokens passed from the application interface model to the dialogue control. This means that the dialogue control component must be able to accept two grammars and generate correct response for user's command.

Another advantage of the event language is that it is easy to learn and use. The structure of the event language is based on the programming language C. This is because C is widely used in the graphics community, and a designer usually feels more comfortable with a tool that is close to his domain of expertise. Therefore, most of the designers will require a minimum amount of learning to use the language effectively. The event language allows the designers to give a high level specification of user interfaces. This makes it easy for the designer to modify the specification and change the flow of the dialogue sequences to suit the demands of different users. It is very convenient for him to test different sets of interaction dialogues by invoking different interaction techniques.

A possible disadvantage to the event notation is that the structure of event handler declarations resemble a program. This is due to the procedural nature of the event handlers. The body of an event handler is similar to the switch statement in C. In this case the designer may find it more natural to use the recursive transition network editor. However, for a large application the designer may have to use multiple subdiagrams to decrease the size of the transition diagrams. By doing so, he may have to traverse a hierarchy of transition diagrams to modify the dialogue specification, and may get lost in the process.

In summary, the design and implementation of the University of Alberta UIMS, in particular, the event based dialogue component, would be helpful in reducing the time and cost of user interface construction and increase the quality of the user interfaces to suit different users. The event language gives the interaction designer the ability to modify human-computer dialogues effectively as the characteristics of the users change.

7.3. Extensions and Further Work

A possible extension to the implementation of the event language is to include the initialization of arrays and structures in the variable section of an event handler declaration. Unfortunately it is non-trivial to convert the contents of an array or a structure into elements of the system created array associated with each instance of the event handler in a "clean" way that does not slow down the compiler significantly.

Another possible extension of this work is to design the event language based on C++, a superset of C, that provides facilities for data abstraction [Stroustrup 84]. Data abstraction is supported by enabling the programmer to define new data types, called "classes". Each object of a class has its own copy of the data members of a class. The members of a class can only be accessed by an explicitly declared set of functions. Thus, a class definition is similar to an event handler declaration which also provides each instance of an event handler with its own set of local variables that can only be manipulated by its associated C procedure. Some other facilities provided by C++ include operator overloading, and guaranteed initialization of data structures. This extension may help the event language in structuring large systems and increase its expressive power.

There is very little hard evidence to indicate that the event language is the best specification language for the dialogue control component. The only way of obtaining hard evidence on the value of a new language is to use it in "real world" application. Experience gained from these applications will indicate ways in which the language can be improved.

References

[Aho and Ullman 77] Aho A.V. and Ullman J.D., "Principles of Compiler Design", Addison-Wesley, Reading Mass., 1977.

[Benbasat and Wand 84] Benbasat I. and Wand Y., "A Structured Approach To Designing Human-Computer Dialogues", Int. J. Man-Machine Studies, vol. 21, no. 2, p.105-126, 1984.

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

[Date 81] Date C.J., "An Introduction to Database System", Addison-Wesley, Reading Mass., 1981.

[Edmonds and Guests 78] Edmonds E.A. and Guests S.P., "SYNICS - a FORTRAN subroutine package for translation". Report N[illegible]8, Man-Computer Interaction Research Group, Leicester Polytechnic, 1978.

[Edmonds 81] Edmonds E.A., "Adaptive Man-Computer Interfaces". In Computing Skills and the User Interface (eds. Coombs and Alty), Academic Press, p.389-426, 1981.

[Edmonds 82a] Edmonds E.A., " The Man-Computer Interface: a note on concepts and design", Int. J. Man-Machine Studies, vol 16, no. 3, p.231-236, 1982.

[Edmonds 82b] Edmonds E.A., "Matching the User's Model of the Machine to the Machine", Proc. Man-Machine Systems, Manchester, p.72-75, 1982.

[Goldberg and Robson 83] Goldberg A. and Robson D., "Smalltalk-80: The Language and its Implementation", Addison-Wesley, Reading Mass., 1983.

[Green 79] Green M., "A Graphical Input Programming System", M.Sc. Thesis, Department of Computer Science, University of Toronto, 1979.

[Green 81a] Green M., "A Methodology for the Specification of Graphical User Interfaces", SIGGRAPH'81, p.99-108, 1981.

[Green 81b] Green M., "A Specification Language and Design Notation for Graphical User Interfaces", Comp[illegible]r Science Technical Report No. TR 81-CS-09, Unit for Computer Science, McMaster University, 1981.

[Green 82] Green M., "Towards a User Interface Prototyping System", Graphics Interface'82, p.37-45, 1982.

[Green 84a] Green M., "Design Notations and User Interface Management Systems", in Seeheim Workshop on User Interface Management Systems, ed. G. Pfaff and P.J.W. ten Hagen, Springer-Verlag, 1984.

[Green 84b] Green M., "Report on Dialogue Specification Tools", Computer Graphics Forum, vol. 3, p.305-313, 1984.

[Green 84c] 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.

[Green 84d] Green M., "The design of Graphical User Interfaces", Ph.D. Thesis, Department of Computer Science, University of Toronto, 1984.

[Green 85a] Green M., "The University of Alberta UIMS", SIGGRAPH Proceedings, 1985.

[Green 85b] Green M.; "WINDLIB: An Object Oriented Graphics Package", WINDLIB Programmer's Manual, Department of Computing Science, University of Alberta, 1985.

[Guest 82] Guest S.P., "The Use of Software Tools for Dialogue Design", Int. J. Man-Machine Studies, vol 16, no. 3, p.263-285, 1982.

[Guest and Edmonds 84] Guest S.P. and Edmonds E.A., "Graphical Support in a User Interface Management System", Human-Computer Interface Research Unit, Leicester Polytechnic, U.K., 1984.

[Horowitz 84] Horowitz E., "Fundamentals of Programming Languages", Computer Science Press, Maryland, 1984.

[Johnson 83] Johnson S.C., "YACC: Yet Another Compiler-Compiler", UNIX Programmer's Manual: Languages Support Tools, Bell Laboratories, seventh edition, Volume 2, 1983.

[Kamran and Feldman 83] Kamran A. and Feldman M.B., "Graphics Programming Independent of Interaction Techniques and Styles", Computer Graphics, p.58-66, 1983.

[Kasik 82] Kasik D.J., "A User Interface Management System", SIGGRAPH'82, p.99-106, 1982.

[Kernighan and Ritchie 78] Kernighan B.W. and Ritchie D.M., "The C Programming Language", Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Lau 85] Lau S.C., "The Use of Recursive Transition Networks for Dialogue Design in User Interfaces", M.Sc. Thesis, Department of Computing Science, University of Alberta, 1985.

[Lesk and Schmidt 83] Lesk M.E. and Schmidt E., "Lex - A Lexical Analyzer Generator", UNIX Programmer's Manual: Languages Support Tools, Bell Laboratories, seventh edition, Volume 2, 1983.

[Newman 68] Newman W.M., "A System for Interactive Graphical Programming", SJCC 1968, Thompson Books, Washington DC., p.47-54, 1968.

[Olsen 83] Olsen D.R., "Automatic Generation of Interactive Systems", Computer Graphics, vol. 17, no. 1, p.53-57, 1983.

[Olsen and Dempsey 83a] Olsen D.R. and Dempsey E.P., "SYNGRAPH: A Graphical User Interface Generator", SIGGRAPH'83, p.43-50, 1983.

[Olsen and Dempsey 83b] Olsen D.R. and Dempsey E.P., "Syntax Directed Graphical Interaction", ACM, p.112-117, 1983.

[Rogers and Feldman 81] Rogers G.T. and Feldman M.B., "An Intermediate Language and an Interpreter for Style-Independent Interactive Systems", Report GWU-IIST-81-21, Department of Electrical Engineering and Computer Science, George Washington University, Aug. 1981.

[Rosenthal 82] Rosenthal D.S.H., "Managing Graphical Resources", Computer Graphics 16, July 1982.

[Seattle 83] "Graphical Input Interaction Techniques (GIIT)", Workshop Summary, Computer Graphics, vol. 17, no. 1, Jan. 1983.

[Seeheim 84] Seeheim Workshop on User Interface Management Systems, EUROGRAPHICS-Springer Series, Springer-Verlag, 1984.

[Singh 85] Singh G., "Presentation Component for the University of Alberta UIMS", M.Sc. Thesis, Department of Computing Science, University of Alberta, 1985.

[Stroustrup 84] Stroustrup B., "Data Abstraction in C", AT&T Bell Laboratories Technical Journal, Oct. 1984.

[Tanner and Buxton 84] Tanner P.P. and Buxton W.A.S., "Some Issues in Future User Interaction Management System (UIMS) Development", 1984.

Appendix A1

YACC Specification of the Event Language

```
/*                         Event Language Compiler


Purpose:
    The purpose of this program is to compile programs written
in the event based language into executable C codes.

-----------------------------------------------------------------------

Flow of program:
    1. The 'main' routine opens the input file, creates a permanent
       output file which has the same name as the input file with ".e"
       added at the end, and a temporary output file "eetemp".
    2  The first event handler in the input file is parsed into the
       temporary file. The numbers of variables, input tokens and
       events, and a list of the events and a mapping of the tokens
       into events are recorded in the permanent output file.
    3. Contents in the temporary output file is then copied into the
       permanent output file.
    4. The temporary file is closed and then reopened for storing the
       compiled output of the next event handler from the parser.
    5. Step 2 to 4 are repeated until all the event handlers in the input
       file has been parsed. The temporary file is unlinked at the end.

-----------------------------------------------------------------------
How to use the Program:
    If the name of a program is "prog", the following command can be
used to compile it:

                    ec   prog

After the compilation, the output file is in the file "prog.e" which
is executable by the C compiler.

-----------------------------------------------------------------------
Input:
    A file which contains program written in the event based language.

-----------------------------------------------------------------------
Output:
    A file which has the same name as the input file with ".e" added at the
end.

-----------------------------------------------------------------------

*/

%{

#include "def.h"

extern struct eevr_list   *eevar_list[] ;
extern char yytext[] ;
extern int  eeflag ;
extern int  eetype ;
extern int  eeevent ;
extern int  eecomment ;
extern int   eestring ;
extern int   eeassign_stmt ;
extern int   eeline ;
extern int   eevar_cnt ;

int   eeevent_cnt   ;
int   eetoken_cnt   ;
char  *eetemp_buf,  *eetemp_fname, *eeevh_name ;
char  eein_fname[ EEFNAME_LEN ], eeout_fname[ EEFNAME_LEN ] ;

struct  eeev_list {
      char  *ev_name ;
      struct eeev_list  *next ;
} ;

struct  eetk_list {
      char  *token_name ;
      char  *ev_name ;
      struct eetk_list  *next ;
} ;

/*  storage for event and token lists  */
struct  eeev_list  *eeevent_list[ EEHASHSIZE ] ;
struct  eetk_list  *eetoken_list[ EEHASHSIZE ] ;

%}

%token    IDENTIFIER  NUMBER
%token    EVENTHANDLER  IS  TOKEN VAR EVENT END
%token    CHAR  SHORT  INT  DOUBLE  FLOAT  LONG  UNSIGNED
%token    STRUCT  UNION  TYPEDEF
%token    GOTO  RETURN  SIZEOF  BREAK  CONTINUE
%token    IF  ELSE  FOR  DO  WHILE  SWITCH  CASE  DEFAULT  ENTRY
%token    INCR_OP  BIN_OP  ASGN_OP
%token    RPOINTER  STRING

%start  evhndlr_file

%%
evhndlr_file : evhndlr
             | evhndlr_file  evhndlr
             ;

evhndlr      : head body tail
             | error ';'
             ;

head  : EVENTHANDLER  IDENTIFIER
              {  eestart( ) ; }
        IS
      ;
tail  : END IDENTIFIER
              {  eestatistics( ) ; }
      | ';'
              {  eewrap_up( ) ; }
      ;
body  : token_decl  var_decl  ev_body
      ;

/*------------  token declaration  -------------------*/

token_decl    :                  /* empty */
              | TOKEN  token_table
```

```
                   error ';'
                 ;
token_table    : token_list
                 token_table  token_list
                 ;
token_list     : IDENTIFIER
                       {  eetemp_buf = eestrsave( yytext ) ; }
                 IDENTIFIER
                       {  eestore_token( ) ; }
                 ';'
                 ;

/*------------  variable declaration  ------------*/

var_decl       :          /* empty */
                 VAR  decl
                 ;
decl           : var_list
                 decl  var_list
                 ;
var_list       : type  decl_list  ';'
                 TYPEDEF  var_list
                 error ';'
                 ;
type           : CHAR
                 SHORT
                 INT
                 LONG
                 UNSIGNED
                 FLOAT
                 DOUBLE
                 IDENTIFIER
                 struct
                 union
                 ;
decl_list      : declarator
                 decl_list  ','  declarator
                 ;
declarator     : [illegible]
                 var_name [illegible] initializer
                       {  fprintf( [illegible] \n" ) ;
                          eeflag = EESKIP ; }
                 ;
var_name       : IDENTIFIER
                       {  eetemp_buf = eestrsave( yytext ) ;
                          eestore_var( ) ;  }
                 '('  var_name  ')'
                 var_name  '['  ']'
                 var_name  '['  const_expr  ']'
                 '*'  var_name
                 var_name  '('  ')'
                 ;
struct         : STRUCT  '{'  sdecl_list  '}'
                 STRUCT  IDENTIFIER  '{'  sdecl_list  '}'
                 STRUCT  IDENTIFIER
                 ;
union          : UNION  '{'  sdecl_list  '}'
                 UNION  IDENTIFIER  '{'  sdecl_list  '}'
                 UNION  IDENTIFIER
                 ;
sdecl_list     : sdeclaration
                 sdecl_list  sdeclaration
                 ;
```

```
sdeclaration : type  sdecl
             ;
sdecl        : svar
               sdecl  ','  svar
             ;
svar         : var_name
               var_name  ':'  const_expr
               ':'  const_expr
             ;
initializer  :        { eecopy_initializer( ) ; }
               '='  expr
             ;
type_name    : type  abs_decl
             ;
abs_decl     :            /*  empty  */
               '('  abs_decl  ')'
               '*'  abs_decl
               abs_decl  '('  ')'
               abs_decl  '['  const_expr  ']'
             ;

/*------------------  event body  -------------------*/

ev_body      : ev_decl
               ev_body  ev_decl
             ;
ev_decl      : EVENT  IDENTIFIER cmpd_stat
                    {  fprintf( yyout, "%s",  "   break ;" ) ; }
               EVENT  IDENTIFIER  ':'  IDENTIFIER  cmpd_stat
                    {  fprintf( yyout, "%s",  "   break ;" ) ; }
             ;
cmpd_stat    : '{'  stat_list  '}'
             ;
stat_list    : statement
               stat_list  statement
             ;
statement    : cmpd_stat
               expr  ';'
               IF  '('  expr  ')'  statement
               IF  '('  expr  ')'  statement  ELSE  statement
               WHILE  '('  expr  ')'  statement
               DO  statement  WHILE  '('  expr  ')'  ';'
               FOR  '(' opt_expr  ';'  opt_expr  ';'  opt_expr  ')'  statement
               SWITCH  '('  expr  ')'  statement
               CASE  const_expr  ':'  statement
               DEFAULT  ':'  statement
               BREAK  ';'
               CONTINUE  ';'
               RETURN  ';'
               RETURN  expr  ';'
               GOTO  IDENTIFIER  ';'
               IDENTIFIER  ':'  statement
               ';'
               error  ';'
             ;
expr       : primary
             '*'  expr
             '&'  expr
             '-'  expr
             '!'  expr
             '~'  expr
             INCR_OP  lvalue
             lvalue  INCR_OP
```

```
                | SIZEOF  expr
                | '('  type_name  ')'  expr
                | expr  bin_op  expr
                | expr  '?'  expr  ':'  expr
                | lvalue  asgn_op
                        {  eeassign_stmt = EETRUE ;  }
                  expr
                | expr  ','  expr
                ;
primary         : NUMBER
                | '"'  string  '"'
                | '('  expr  ')'
                | primary  '('  expr_list  ')'
                | lvalue
                ;
string          : STRING
                | string  STRING
                ;
lvalue          : IDENTIFIER
                | '*'  expr
                | primary  '['  expr  ']'
                | primary  RPOINTER  IDENTIFIER
                | lvalue  '.'  IDENTIFIER
                | '('  lvalue  ')'
                ;
expr_list          :        /*  empty  */
                   | expr
                   | expr_list  ','  expr
                   ;
const_expr         : IDENTIFIER
                   | NUMBER
                   ;
opt_expr           :        /*  empty  */
                   | expr
                   ;
bin_op             : BIN_OP
                   | '*'
                   | '-'
                   | '&'
                   ;
asgn_op            : ASGN_OP
                   | '='
                   ;

%%

#include "lex.yy.c"
extern FILE *yyin, *yyout ;  /* yyout is a temporary file pointer */
FILE   *eeout ;

/*-----------------------------------------------------------------
                           Main Routine
   This function opens the input file and creates an output file which
 has the same name as the input file with ".e" added at the end.  A
 temporary file is also created to store the output from the parser.
-----------------------------------------------------------------*/
main(argc, argv)
int argc ;
char *argv[] ;
{  int i ;

   /* open input file */
```

```
    strcpy( eein_fname, *++argv ) ;
    if ( (yyin = fopen( eein_fname, "r" ) ) == NULL )
       {  fprintf( stderr, "%s%s\n", "*** Can't open ", eein_fname ) ;
          exit( 1 ) ;
       }

    /* create output file name */
    for ( i = 0 ; eein_fname[i] != '\0' ; i++ )
         eeout_fname[i] = eein_fname[i] ;
    eeout_fname[i++] = '.' ;
    eeout_fname[i++] = 'e' ;
    eeout_fname[i] = '\0' ;

    /* open output file */
    eeout = fopen( eeout_fname, "w" ) ;
    yyout = eeout ;
    eetemp_fname = "eetemp" ;

    /* start parsing */
    if ( yyparse() == 0 )
       {  unlink( eetemp_fname ) ;
          exit( 0 ) ;
       }
    else
       {  unlink( eetemp_fname ) ;
          exit( 1 ) ;
       }
}

/*------------------------------------------------------------------------------
     This function copies contents of the file pointed to by 'fp1' into
  the file pointed to by 'fp2'.
------------------------------------------------------------------------------*/
eefilecopy( fp1, fp2 )
FILE *fp1, *fp2 ;
{   int c ;

    while ( ( c= getc( fp1 ) ) != EOF )
         putc( c, fp2 ) ;
}

/*------------------------------------------------------------------------------
       This function prints the error message and the line number where the
  error occurred.
------------------------------------------------------------------------------*/
yyerror(s)
char *s ;
{
    fprintf( stderr, "%s%s%s%d%s%s\n",
                       "'", eein_fname, "', line " , eeline, " : ", s ) ;
}

/*------------------------------------------------------------------------------
        This function re-initializes all the counters and arrays before the
  next event handler is being parsed.
------------------------------------------------------------------------------*/
eestart( )
{  int i ;

    /* store event handler name */
    eeevh_name = eestrsave( yytext ) ;
```

```
      /*  re-initialize variables  */
      eevar_cnt = 0 ;
      eeevent_cnt = 0 ;
      eetoken_cnt = 0 ;

      /*  re-initialize arrays  */
      for ( i=0 ; i <= EEHASHSIZE  ; i++ )
         {  eeevent_list[ i ] = NULL ;
            eetoken_list[ i ] = NULL ;
            eevar_list[ i ] = NULL ;
         }

       yyout = fopen( eetemp_fname, "w" ) ;
       fprintf( yyout, "\n%s\n", "  case INIT :{" ) ;
}

/*----------------------------------------------------------------------------
       This function copies the values of the variables which have been
   initialized in the variable declaration of each event handler into the
   temporary output file.  The variable names themselves have been replaced
   by their corresponding elements in the array "var".
----------------------------------------------------------------------------*/

eecopy_initializer( )
{
     fprintf( yyout, "     %s%d%s", "var[", eevar_cnt-1, "] = " ) ;
     eeflag = EECOPY ;
     EEECHO ;
}

/*----------------------------------------------------------------------------
        This function prints the numbers of variables, input tokens and
   events, and a list of the events and a mapping of input tokens into
   events in an event handler.
----------------------------------------------------------------------------*/
eestatistics( )
{    int offset ;
     char *eeevh ;

     eeevh = eestrsave( yytext ) ;

     /* if the first character of an event handler is in lower case
        then it will be converted to upper case, and vice versa.      */
     offset = 'A' - 'a' ;
     if ( eeevh[0] >= 'a'  &&  eeevh[0] <= 'z' )
        eeevh[0] = eeevh[0] + offset ;
     else
        eeevh[0] = eeevh[0] - offset ;

    fprintf( eeout, "%s\n", eeevh ) ;
    fprintf( eeout, "%d %d %d\n", eevar_cnt, eetoken_cnt, eeevent_cnt ) ;
    eeprint_event( eeout ) ;
    eeprint_token( eeout ) ;
    eeprint_fn_heading( eeout ) ;
}

/*----------------------------------------------------------------------------
      This function uses 'eelookup_ev' to determine whether the event being
   installed is already present.  If there is any duplicate event names, an
   error message will be given.
----------------------------------------------------------------------------*/
eestore_event( s )
```

```
char *s ;
{
    struct eeev_list *np, *eelookup_ev( ) ;
    char  *eestrsave( ), *malloc( ) ;
    int  hashval ;

    if ( (np = eelookup_ev( s )) == NULL )          /* not found */
       { np =(struct eeev_list *) malloc( sizeof(*np)); /* create storage */
          if ( np == NULL )
             {  yyerror("run out of memory storage") ;
                exit( 1 ) ;
             }
          if ( (np->ev_name = eestrsave( s )) == NULL )
             {  yyerror("run out of memory storage") ;
                exit( 1 ) ;
             }

          /* insert new entry into the event list */
          hashval = eehash( np->ev_name ) ;
          np->next = eeevent_list[ hashval ] ;
          eeevent_list[ hashval ] = np ;
          eeevent_cnt++ ;

       }
    else
        yyerror("illegal duplication of event name") ;
}

/*-----------------------------------------------------------------------
        This function looks for string 's' in the event list.
-----------------------------------------------------------------------*/
struct eeev_list *eelookup_ev( s )
char *s ;
{
    struct eeev_list *np ;

    for ( np=eeevent_list[ eehash( s ) ] ; np != NULL ; np = np->next )
         if ( strcmp( s, np->ev_name ) == 0 )     /* found entry */
              return( np ) ;
    return( NULL ) ;                          /* entry not found */
}

/*-----------------------------------------------------------------------
     This function uses 'eelookup_token' to determine whether the token being
  installed is already present.  If there is any duplicate token names, an
  error message will be given.
-----------------------------------------------------------------------*/
eestore_token( )
{
    struct eetk_list *np, *eelookup_token( ) ;
    char  *eestrsave( ), *malloc( ) ;
    int  hashval ;

    if ( (np = eelookup_token( eetemp_buf )) == NULL )        /* not found */
       { np = (struct eetk_list *) malloc( sizeof(*np)); /* create storage */
          if ( np == NULL )
             {  yyerror("run out of memory storage") ;
                exit( 1 ) ;
             }
          if ( (np->token_name = eestrsave( eetemp_buf )) == NULL )
             {  yyerror("run out of memory storage") ;
                exit( 1 ) ;
             }
```

```
            if ( (np->ev_name = eestrsave( yytext )) == NULL )
               { yyerror("run out of memory storage") ;
                 exit( 1 ) ;
               }

            /* insert new entry into the token list */
            hashval = eehash( np->token_name ) ;
            np->next = eetoken_list[ hashval ] ;
            eetoken_list[ hashval ] = np ;
            eetoken_cnt++ ;
         }
    else
        yyerror("illegal duplication of token name") ;
}

/*-----------------------------------------------------------------------
      This function looks for string 's' in the token list.
-----------------------------------------------------------------------*/
struct eetk_list *eelookup_token( s )
char *s ;
{
    struct eetk_list *np ;

    for ( np=eetoken_list[ eehash( s ) ] ; np != NULL ; np = np->next )
        if ( strcmp( s, np->token_name ) == 0 )    /* found entry */
            return( np ) ;
    return( NULL ) ;                           /* entry not found */
}


/*-----------------------------------------------------------------------
     This function prints a list of events found in an event handler.
-----------------------------------------------------------------------*/
eeprint_event( fp )
FILE  *fp ;
{
     struct  eeev_list  *np ;
     int  index, n = 0 ;

     for ( index=0 ; index < EEHASHSIZE ; index++ )
        for ( np=eeevent_list[ index ] ;  np != NULL ; np=np->next )
            {  fprintf( fp, "%s  ", np->ev_name ) ;
               n++ ;
               if ( (n % 5) == 0 )  fprintf( fp, "\n" ) ;
            }
     if ( (n % 5) == 0 )
         fprintf( fp, "%s\n", "%" ) ;
     else fprintf( fp, "\n%s\n", "%" ) ;
}

/*-----------------------------------------------------------------------
    This function prints a mapping of the input tokens into events in an
 event handler.
-----------------------------------------------------------------------*/
eeprint_token( fp )
FILE  *fp ;
{
     struct  eetk_list  *np ;
     int  index, n = 0 ;

     for ( index=0 ; index < EEHASHSIZE ; index++ )
        for ( np=eetoken_list[ index ] ;  np != NULL ; np=np->next )
            {  fprintf( fp, "%s  ", np->token_name ) ;
            fprintf( fp, "%s\n", np->ev_name ) ;
        }
    fprintf( fp, "%s\n", "%" ) ;
}

/*-----------------------------------------------------------------------
      This function prints the heading for each event handler routine.
-----------------------------------------------------------------------*/
eeprint_fn_heading( fp )
FILE  *fp ;
{

    /* check for matching event handler name */
    if ( strcmp( yytext, eeevh_name ) != 0 )
       yyerror( "missing declaration for event handler name" ) ;

    fprintf( fp, "%s", yytext ) ;
    fprintf( fp, "%s\n",
             "(instance_name, event_name, event_value, var)") ;
    fprintf( fp, "%s\n","int  instance_name ;" ) ;
    fprintf( fp, "%s\n","int  event_name ;" ) ;
    fprintf( fp, "%s\n","int  event_value ;" ) ;
    fprintf( fp, "%s\n","int  var[] ;" ) ;
    fprintf( fp, "%s\n","{" ) ;
    fprintf( fp, "\n" ) ;
    fprintf( fp, "%s\n", " switch (event_name)  {" ) ;
}

/*-----------------------------------------------------------------------
     This function re-initializes the [illegible] variables and copies the
 contents of the temporary file into the permanent file.
-----------------------------------------------------------------------*/
eewrap_up( )
{
    /*  re-initialize control variables */
    eeflag = EECOPY ;
    eetype = EEOTHER ;
    eeevent = EEFALSE ;
    eeskip_brace = EEFALSE ;
    eecomment = EEFALSE ;
    eestring = EEFALSE ;
    eeassign_stmt = EEFALSE ;

    /*  print closing brackets at the end of a function  */
    fprintf( yyout, " }\n" ) ;
    fprintf( yyout, "}\n" ) ;
    fprintf( yyout, "%s\n", "%%" ) ;
    fclose( yyout ) ;

    /*  copy contents of temporary file pointed to by "yyout" into
        the permanent file pointed to by "eeout".  */
    yyout = fopen( eetemp_fname, "r" ) ;
    eefilecopy( yyout, eeout ) ;
    fclose( yyout ) ;
    yyout = eeout ;
}
```

Appendix A2

Lexical Analyzer for the Event Language

```
/*                    Lexical Analyzer for the Event Language

 Global variables:
    eeflag           -- a flag to signal whether the input data will be
                        echoed or skipped.
    eetype           -- a flag which indicates the keyword "event" has been
                        scanned. The keyword will be replaced by "case" in the
                        output file.
    eeevent          -- a boolean variable used to control the output routine
                        so that the INTERNAL DECLARATION of an event type will
                        not be echoed.  Echoing will resume after the first
                        bracket sign "{" is detected.
    eeskip_brace     -- a boolean variable used to control the output routine
                        to skip over the bracket sign "{" when the event
                        "INIT" is detected. The body of the "INIT" event will
                        be appended to the PREDEFINED "INIT" event of each
                        event handler.  Each predefined "INIT" event is used
                        to initialize all variables being declared [illegible]
                        event handler routine where it resides.
    eecomment        -- a boolean variable used to signal the be[illegible]
                        the ending of comments in the input file [illegible]
                        comments will be copied directly to the [illegible]
                        without being parsed.
    eeassign_stmt    -- a boolean variable used to control the [illegible]ment of
                        an event name that appears on the RHS of an assignment
                        statement in the input file by its value in the output
                        file.
    eevar_cnt        -- counts the number of variables in each event handler
    eeline           -- gives the number of the line where an error has been
                        found.

---------------------------------------------------------------------------- */
%{

#include "def.h"

#define EEECHO if ( (eeflag != EESKIP) || (eecomment == EETRUE) ) ECHO
#define EENOT_STRING  if ((eecomment == EEFALSE) && (eestring == EEFALSE))

/* The token STRING will be returned to the parser for each character in a
   string which is enclosed by the double quotation signs.  If the
   character does not belong to an array of characters, the character
   (token) itself is returned to the parser.  */
#define EESTRING if ( eestring == EETRUE ) return( STRING )

struct  eevr_list {
   char  *var ;
   int   index ;
   struct  eevr_list  *next ;        /* next entry in chaining */
} ;

/*  symbol table for variables  */
struct eevr_list  *eevar_list[ EEHASHSIZE ] ;
%}

                int eeflag = EECOPY ;
                int eetype = EEOTHER ;
                int eeevent = EEFALSE ;
                int eeskip_brace = EEFALSE ;
                int eecomment = EEFALSE ;
                int eestring = EEFALSE ;
                int eeassign_stmt = EEFALSE ;
                int eevar_cnt  ;
                int eeline = 1 ;
%%
eventhandler                { eeflag = EESKIP;
                              EENOT_STRING return ( EVENTHANDLER ) ;
                              EESTRING ; }
is                          { EENOT_STRING
                                return( IS ) ;
                              else  ECHO ;
                              EESTRING ; }
token                       { EENOT_STRING return( TOKEN ) ;
                              else  EEECHO ;
                              EESTRING ; }
var                         { EEECHO ; EENOT_STRING return( VAR ) ;
                              EESTRING ; }
event                       { EENOT_STRING { eeevent_decl( ) ;
                                           return( EVENT ) ; }
                              else EEECHO ;
                               EESTRING ; }
end                         {  EENOT_STRING { eeflag = EESKIP ;
                                            return( END ) ; }
                               else EEECHO ;
                               EESTRING ;   }
char                         { EEECHO ; EENOT_STRING return( CHAR )  ;
                               EESTRING ;   }
int                          { EEECHO ; EENOT_STRING return( INT )   ;
                               EESTRING ; }
short                        { EEECHO ; EENOT_STRING return( SHORT ) ;
                               EESTRING ;   }
float                        { EEECHO ; EENOT_STRING return( FLOAT ) ;
                               EESTRING ;   }
double                       { EEECHO ; EENOT_STRING return( DOUBLE ) ;
                               EESTRING ; }
long                         { EEECHO ; EENOT_STRING return( LONG )  ;
                               EESTRING ; }
unsigned                     { EEECHO ; EENOT_STRING return( UNSIGNED ) ;
                               EESTRING ; }
struct                       { EEECHO ; EENOT_STRING return( STRUCT ) ;
                               EESTRING ; }
union                        { EEECHO ; EENOT_STRING return( UNION )  ;
                               EESTRING ; }
typedef                      { EEECHO ; EENOT_STRING return( TYPEDEF )  ;
                               EESTRING ; }
goto                         { EEECHO ; EENOT_STRING return( GOTO )   ;
                               EESTRING ; }
return                       { EEECHO ; EENOT_STRING return( RETURN ) ;
                               EESTRING ; }
sizeof                       { EEECHO ; EENOT_STRING return( SIZEOF ) ;
                               EESTRING ; }
break                        { EEECHO ; EENOT_STRING return( BREAK )  ;
                               EESTRING ; }
continue                     { EEECHO ; EENOT_STRING return( CONTINUE ) ;
                               EESTRING ; }
```

```
if                          { EEECHO ; EENOT_STRING return( IF ) ;
                              EESTRING ; }
else                        { EEECHO ; EENOT_STRING return( ELSE ) ;
                              EESTRING ; }
for                         { EEECHO ; EENOT_STRING return( FOR ) ;
                              EESTRING ; }
do                          { EEECHO ; EENOT_STRING return( DO ) ;
                              EESTRING ; }
while                       { EEECHO ; EENOT_STRING return( WHILE ) ;
                              EESTRING ; }
switch                      { EEECHO ; EENOT_STRING return( SWITCH ) ;
                              EESTRING ; }
case                        { EEECHO ; EENOT_STRING return( CASE ) ;
                              EESTRING ; }
default                     { EEECHO ; EENOT_STRING return( DEFAULT ) ;
                              EESTRING ; }
entry                       { EEECHO ; EENOT_STRING return( ENTRY ) ;
                              EESTRING ; }

[A-Za-z][A-Za-z0-9_]*       {  eeconvert( ) ;
                               EENOT_STRING return( IDENTIFIER ) ;
                               EESTRING ; }
-?[0-9]+"."?[0-9]*          {  EEECHO ; EENOT_STRING return( NUMBER ) ;
                               EESTRING ; }
[ \t]                       EEECHO ;
\n                          {  EEECHO ; eeline++ ; }
("++" "--")                 {  EEECHO ; EENOT_STRING return( INCR_OP ) ;
                               EESTRING ; }
"->"                        {  EEECHO ; EENOT_STRING return( RPOINTER ) ;
                               EESTRING ; }

("/" "%" "+" ">>")          { EEECHO ; EENOT_STRING return( BIN_OP ) ;
                              EESTRING ; }
("<<" ">" "<" "<=")         { EEECHO ; EENOT_STRING return( BIN_OP ) ;
                            EESTRING ; }
(">=" "==" "!=" "|")        { EEECHO ; EENOT_STRING return( BIN_OP ) ;
                              EESTRING ; }
("^" "&&" "||" "?:")        { EEECHO ; EENOT_STRING return( BIN_OP ) ;
                              EESTRING ; }

("+=" "-=" "*=")            { EEECHO ; EENOT_STRING return( ASGN_OP ) ;
                              EESTRING ; }
("/=" "%=" ">>=")           { EEECHO ; EENOT_STRING return( ASGN_OP ) ;
                              EESTRING ; }
("<<=" "&=" "|=" "^=")        { EEECHO ; EENOT_STRING return( ASGN_OP ) ;
                                EESTRING ; }

"/*"                        { ECHO ; eecomment = EETRUE ;
                              EESTRING ; }
"*/"                        { ECHO ; eecomment = EEFALSE ;
                              EESTRING ; }

","                         { EEECHO ; EENOT_STRING return( ',' ) ;
                              EESTRING ; }
"."                         { EEECHO ; EENOT_STRING return( '.' ) ;
                              EESTRING ; }
":"                         { EEECHO ; EENOT_STRING return( ':' ) ;
                              EESTRING ; }
";"                         { EEECHO ; EENOT_STRING return( ';' ) ;
                              EESTRING ; }
"!"                         { EEECHO ; EENOT_STRING return( '!' ) ;
                              EESTRING ; }
"?"                         { EEECHO ; EENOT_STRING return( '?' ) ;
```

```
                                      EESTRING ; }
"*"                                 { EEECHO ; EENOT_STRING return( '*' ) ;
                                      EESTRING ; }
"-"                                 { EEECHO ; EENOT_STRING return( '-' ) ;
                                      EESTRING ; }
"&"                                 { EEECHO ; EENOT_STRING return( '&' ) ;
                                      EESTRING ; }
"·"                                 { EEECHO ; EENOT_STRING return( '·' ) ;
                                      EESTRING ; }
"="                                 { EEECHO ; EENOT_STRING return( '=' ) ;
                                      EESTRING ; }
"{"                                 { eebrace( ) ;
                                      EENOT_STRING return( '{' ) ;
                                      EESTRING ; }
"}"                                 { EEECHO ; EENOT_STRING return( '}' ) ;
                                      EESTRING ; }
"["                                 { EEECHO ; EENOT_STRING return( '[' ) ;
                                      EESTRING ; }
"]"                                 { EEECHO ; EENOT_STRING return( ']' ) ;
                                      EESTRING ; }
"("                                 { EEECHO ; EENOT_STRING return( '(' ) ;
                                      EESTRING ; }
")"                                 { EEECHO ; EENOT_STRING return( ')' ) ;
                                      EESTRING ; }
\"                                  { ECHO ;
                                      if (eecomment == EEFALSE)
                                         {  if ( eestring == EEFALSE )
                                               eestring = EETRUE ;
                                            else
                                               eestring = EEFALSE ;
                                            return( '"' ) ;
                                         } ; }

%%

/*--------------------------------------------------------------------------
    This function sets the appropriate flags when a new EVENT is found.
--------------------------------------------------------------------------*/
eeevent_decl( )
{
   eeflag = EECOPY ;
   eetype = EEEVENT ;
   eeevent = EETRUE ;
}

/*--------------------------------------------------------------------------
    This function does one of the following:
      1.   if the very first event found is not "INIT", then it adds the
           closing bracket for the default "INIT" event and converts the
           keyword "event" to "case", otherwise it appends the body of
           the "INIT" event written by the user to the body of the
           default "INIT" event.  Error message is given if the "INIT"
           event specified by the user is not the first event in the
           event handler. The new event found is also stored in the
           event list.
      2.   if the identifier found is a variable, it is replaced by its
           corresponding element in the array "var".
      3.   if the identifier is an event name and it appears at the RHS
           of an assignment statemnet, the event name is replaced by its
           value.
        4.  if none of the above is true, the identifier is echoed.
-----------------------------------------------------------------------*/
eeconvert( )
{
   struct eevr_list *np, *eelookup_var( ) ;
   struct eeev_list *eelookup_ev( ) ;

   if ( eetype == EEEVENT )
      { if ( strcmp(yytext, "INIT") != 0 )
           {  if (eeevent_cnt == 0)
                 {  fprintf( yyout, " }   break ;\n" ) ;
                    eestore_event( "INIT" ) ;
                 }
              fprintf( yyout, "\n%s", "   case " ) ;
              EEECHO ;
              fprintf( yyout,"%s", " : " ) ;
           }
        else if ( eeevent_cnt != 0 )
          yyerror("'INIT' must be the 1st event following 'var' declaration');
        else
           /* "INIT" is the first event after 'var' decl */
           eeskip_brace = EETRUE ;

        eetype = EEOTHER ;
        eeflag = EESKIP ;
        eestore_event( yytext ) ;
      }

   else if ( (np = eelookup_var(yytext)) != NULL )
      /*  convert variable name to an element of the array "var"   */
      fprintf( yyout, "%s%d%s", "var[", np->index, "]" ) ;


   else if ( (eeassign_stmt == EETRUE) &&
             (eelookup_ev(yytext) != NULL) )
      /*  convert event name to its value   */
      fprintf( yyout, "event_value" ) ;

   else
      EEECHO ;

   eeassign_stmt = EEFALSE ;

}


/*-------------------------------------------------------------------
    This function resets the appropriate flags when the first bracket
  sign "{" after an event declaration has been detected, so that the
  event body will be echoed.
---------------------------------------------------------------------*/
eebrace( )
{
    if ( eeskip_brace == EETRUE )
       {
         /*  reset flags  */
         eeskip_brace = EEFALSE ;
         eeflag = EECOPY ;
       }
    else if ( eeevent == EETRUE )
       { eeevent = EEFALSE ;
         eeflag = EECOPY ;
```

```
          /* echo the bracket sign for all events except "INIT"  */
          EEECHO ;
        }
      else  EEECHO ;
}

/*-----------------------------------------------------------------------
        This function stores the variable into the variable list.
-----------------------------------------------------------------------*/
eestore_var( )
{
  struct eevr_list *np, *eeinstall_var( ) ;

  if ( (np = eeinstall_var( yytext, eevar_cnt )) == NULL )
     {  yyerror( "symbol table overflow" ) ;
        exit( 1 ) ;                              /* terminate execution */
     }
  else
     eevar_cnt ++ ;
}

/*-----------------------------------------------------------------------
      This function uses 'eelookup_var' to determine whether the variable to
  be installed is already present : if so, the new 'index' will supersede
  the old one.
-----------------------------------------------------------------------*/
struct eevr_list  *eeinstall_var( var, ix )
char  *var ;
int    ix ;
{
    struct eevr_list *np, *eelookup_var( ) ;
    char  *eestrsave( ), *malloc( ) ;
    int  hashval ;

    if ( (np = eelookup_var( var )) == NULL )         /* not found */
       { np = (struct eevr_list *) malloc(sizeof(*np)); /* create storage */
          if ( np == NULL )
             return( NULL ) ;
          if ( (np->var = eestrsave( var )) == NULL )
             return( NULL ) ;

          /* insert new entry into the symbol table */
          hashval = eehash( np->var ) ;
          np->next = eevar_list[ hashval ] ;
          eevar_list[ hashval ] = np ;
       }

    np->index = ix ;
    return( np ) ;
}

/*-----------------------------------------------------------------------
       This function looks for string 's' in the variable list.
-----------------------------------------------------------------------*/
struct eevr_list *eelookup_var( s )
char *s ;
{
    struct eevr_list *np ;

    for ( np=eevar_list[ eehash( s ) ] ;  np != NULL  ; np = np->next )
          if ( strcmp( s, np->var ) == 0 )       /* found entry */
                return( np ) ;
    return( NULL ) ;                              /* entry not found */
```

```
}

/*-----------------------------------------------------------------------
        This function forms the hash value for the string 's'
-----------------------------------------------------------------------
eehash( s )
char *s ;
{
     int hashval ;

     for ( hashval = 0 ;  *s != '\0' ; )
        hashval += *s++ ;
     return( hashval % EEHASHSIZE ) ;
}

/*-----------------------------------------------------------------------
        This function saves the string 's' in location 'p'
-----------------------------------------------------------------------
char *eestrsave( s )
char *s ;
{
     char  *p, *malloc( ) ;

     if ( (p = malloc( strlen(s)+1 ) ) != NULL )
         strcpy( p, s ) ;
     return( p ) ;
}

/*---------------------- end ------------------------------------------
```