National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL-339 (r. 88/04) c

Canada

The University of Alberta

Automating the Lexical and Syntactic Design of Graphical User Interfaces

by

Gurminder Singh

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

Department of Computing Science

Edmonton, Alberta
Spring 1989

National Library of Canada

Bibliothèque nationale du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-52938-5

Canada

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Gurminder Singh

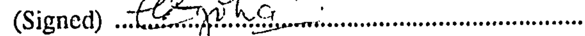
TITLE OF THESIS: Automating the Lexical and Syntactic Design of Graphical User Interfaces

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Doctor of Philosophy

YEAR THIS DEGREE GRANTED: 1989

Permission is hereby granted to The University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) 

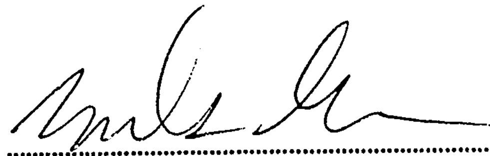
Permanent Address:
CC/40A, G-8 Rajouri Garden
New Delhi - 110 064
India

Dated

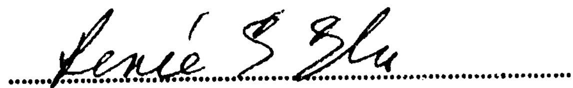
THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

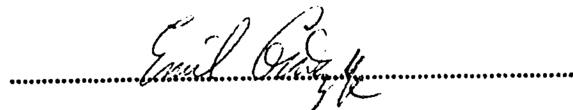
The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Automating the Lexical and Syntactic Design of Graphical User Interfaces** submitted by **Gurminder Singh** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy.**



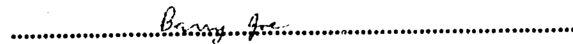
Mark Green - Supervisor



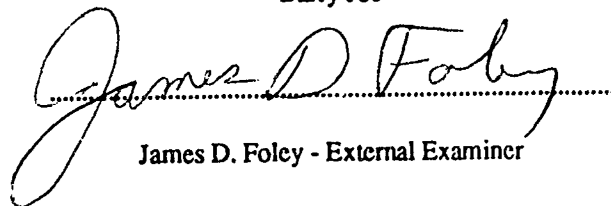
Renee Elio



Emil Girczyc



Barry Joe



James D. Foley - External Examiner

Date

Abstract

The primary goal of this thesis is to address a key problem with UIMSs: their inability to help in the initial design of user interfaces. Because of this inability, existing UIMSs require the interface designer to work at very low levels of syntactic and lexical details, which can be very time-consuming and expensive in terms of effort required. Also, the detailed design produced by the designer must be provided to the UIMS in a notation that it can process, which makes the UIMS difficult to use and increases the chance of error as the interface descriptions tend to be quite large.

The approach followed in this thesis to tackle this problem is to automatically produce the initial design of the user interface and implement it, and then enable the interface designer to improve its appearance and effectiveness by refining it. The interface designer, in this approach, works at the conceptual level of the user interface and produces a high level description of the commands the interface is to support. Based on this description the syntactic and lexical levels of the interface are automatically designed and implemented. This interface can be refined by the designer to improve the resulting interaction with the user. A UIMS based on this approach has been developed, and the research leading to its design and implementation is described in this thesis.

The two important contributions of this research are:

- The research shows that it is possible to automatically produce the initial lexical and syntactic design of graphical user interfaces. This initial design can then be refined very easily and rapidly by the designer, and automatically implemented by the UIMS. To the best of our knowledge, the UIMS presented in this thesis is amongst the first to follow such an approach, and takes the state-of-the-art beyond the capabilities of a "conventional" UIMS.

- This research makes a significant step forward in the direction of increasing the ease of use of UIMSs. The interface designer is no longer required to deal with detailed interface specifications, which are often cryptic, too time-consuming to produce, and error-prone. In our UIMS, a high level description of the commands supported by the application is directly transformed into the interface design and implemented. In addition, when editing interfaces, the changes are immediately visible and executable. This makes it much easier to develop interfaces.

The UIMS has been used to develop interfaces for a number of applications, including a three-dimensional skeleton editor and a distributed network editor. Experience with these examples has shown that the UIMS cuts down tremendously on the time and effort required for developing interfaces. Also, since it does not take much time and effort to produce an interface, a number of variants of an interface can be built and tested till the "right" look and feel is achieved.

Acknowledgements

I am indebted to my thesis supervisor, Mark Green, for his guidance and encouragement. I thank him for suffering through many drafts of this thesis and improving its readability. With Mark, I have worked for a looong time (Mark likes to tell people that I have been in the department longer than him) and enjoyed much of it. For all he has done for me, I owe him a lot of beer (I could mention alcohol-inspired research here, but I won't).

I would also like to thank the other members of my committee: Professors Renee Elio, Emil Girczyc, and Barry Joe for their helpful comments, and Professor James Foley of the George Washington University for being the external reviewer and taking the time to read and comment on my thesis.

I would like to thank Chris Shaw and Scott Goodwin for reading parts of this thesis. Chris Shaw sometimes got carried away with his red pen, but thanks anyways. Thanks are due to Ajit Singh for writing the introduction to the FrameWorks model (section 1 of Appendix 2), and to Glenn Klettke for writing major parts of Appendix 3. Thanks are also due to Glenn Klettke, Carolyn Morris, Ramesh Sankar, Ajit Singh, and Haiying Wong for using the UIMS and providing their feedback. Thanks to Franco Carlacci for providing the software for producing screen dumps, and to George Ferguson for help with troff and other utilities. Thanks to Ken Barker, Surinder Dhanjal, and Hanqui Sun for moral support. Alynn Klassen, Rene Leiva, and Steve Sutphen helped by keeping the systems running and provided help whenever asked.

This research would not have been possible without the financial support from the Department of Computing Science, Faculty of Grad Studies and Research of the University of Alberta, and Natural Sciences and Engineering Research Council (NSERC) of Canada.

Finally, I would like to thank my parents and siblings for making my education possible, and my in-laws for encouragement and support. Special thanks are due to my wife, Navneet, and twin kidlings, Zubin and Rubin, for their love and for giving me a reason to finish this work.

Table of Contents

Chapter	Page
Chapter 1: Introduction	1
1.1. The Problem	1
1.2. Automating the Design of Graphical User Interfaces	2
1.2.1. Designing the Presentation Component	3
1.2.2. Designing the Dialogue Control Component	4
1.2.3. Refining the User Interface	5
1.3. Contributions	6
1.4. Non-Contributions	9
1.5. Thesis Outline	10
Chapter 2: Basic Concepts and Related Research	11
2.1. Definition of Terms	11
2.2. Motivation for UIMSs	12
2.2.1. Advantages of UIMSs	13
2.3. A Bit of History	14
2.4. Structure of UIMSs	16
2.5. A Survey of Existing UIMSs	18
2.5.1. Libraries or Toolkits	19
2.5.2. Object-Oriented UIMSs	19
2.5.3. Grammar Based Systems	20
2.5.4. Transition Network Based Systems	20
2.5.5. Graphical Environments	23
2.5.6. Others	24
2.6. Some Existing UIMSs	25
2.6.1. The UofA UIMS	25
2.6.2. MIKE	29
2.6.3. UIDE	36
2.7. Shortcomings of Existing UIMSs	40
2.8. Chapter Summary	48
Chapter 3: Overview and Extended Example	49
3.1. The Approach	49
3.2. The System	50
3.2.1. The Designer's Interface	50
3.2.2. Design Principles	52
3.2.3. Overview of the UIMS	53
3.2.3.1. Diction	54

3.2.3.2. Chisel 54
3.2.3.3. vu 54
3.2.3.4. Run-Time Support 55
3.2.4. Implementation 55
3.3. Detailed Example 56
3.3.1. Producing Other Variants 67
3.4. Chapter Summary 69
Chapter 4: Designing the Dialogue Control Component 70
4.1. Diction 70
4.1.1. Design Principles 70
4.1.2. Input of Diction 72
4.1.3. Output of Diction 73
4.2. The Dialogue Control Component 73
4.2.1. The Event Model 74
4.2.2. Structure of Event Handlers 75
4.3. Producing HOUSE_KEEPER 79
4.4. Producing Event Handlers for Commands 81
4.5. Producing HELPER 86
4.6. Caveats 86
4.7. An Example 87
4.8. The Input for Chisel 90
4.9. Chapter Summary 91
Chapter 5: Designing the Presentation Component 92
5.1. Chisel 93
5.1.1. Design Principles 93
5.1.2. Chisel's Input 94
5.1.2.1. Dialogue Requirements 94
5.1.2.2. Device Description 95
5.1.2.3. User's Preferences 96
5.1.3. Chisel's Output 98
5.2. Designing Presentation Components 98
5.2.1. Interaction Technique Selection 98
5.2.2. Attribute Determination 101
5.2.3. Placing Interaction Techniques 103
5.2.4. Interface Designer's Control 105
5.3. An Example 106
5.3.1. Producing Other Variants 108
5.4. Structure of Chisel 110
5.5. Chapter Summary 113

Chapter 6: Refining the Presentation Component 114
6.1. vu 114
6.1.1. Design Principles 115
6.1.2. Vu's Interface 116
6.1.2.1. The Workshop Subsystem 117
6.1.2.2. The Objects Subsystem 119
6.1.2.3. The Simulation Subsystem 120
6.1.2.4. The Help Subsystem 120
6.1.2.5. The Menu Subsystem 121
6.2. Defining Output Tokens 121
6.3. vu in Action 122
6.4. Output of vu 128
6.5. Implementation and Structure of vu 129
6.6. Comparison with Existing Systems 131
6.7. Run-Time Support 134
6.8. Chapter Summary 134
Chapter 7: Run-Time Support 135
7.1. Managing Communication 135
7.2. Creating and Destroying Event Handlers 138
7.3. Chapter Summary 139
Chapter 8: Summary and Conclusions 140
8.1. Range of the UIMS 140
8.2. Experience 143
8.2.1. Experience with Developing the UIMS 143
8.2.2. Experience with Using the UIMS 144
8.3. Implications of the Approach 147
8.4. Relation to Existing Approaches/UIMSs 148
8.4.1. Low-Level UIMSs 149
8.4.2. High-Level UIMSs 150
8.5. Review of Contributions 153
8.6. Future Work 154
8.6.1. Additional Features 154
8.6.1.1. Chisel Reading in Refined Presentation Components 154
8.6.1.2. Chisel Producing Presentation Database 155
8.6.1.3. Interactive Entry of the Input Syntax 155
8.6.2. Areas for Future Research 155
8.6.2.1. Run-Time Structures 156
8.6.2.2. Automatic Analysis 156
8.6.2.3. Application Output 156

8.6.2.4. Higher Level Design Tools 157
8.6.2.5. User Interface Design Guidelines 157
References 159
Appendix A1: BNF Specifiction for Diction's Input 173
Appendix A2: Developing an Interface for FrameWorks 175
Appendix A3: Experience with Using the UIMS 183
Appendix A4: Event Handlers for the Skeleton Editor (section 3.3) 185
Appendix A5: Event Handlers for the Example Dialogue of Figure 4.7 198
Appendix A6: Rules Used in Chisel 203

List of Tables

Table	Page
3.1 Output Tokens	63
5.1 Measures of Specificity	103
8.1 Summary of Users' background and Systems	145
A2.1 Output Tokens	179
A6.1 Selecting Interaction Techniques	204
A6.2 Measures of Specificity	205

List of Figures

Figure	Page
1.1 Seeheim Model of User Interfaces	2
2.1 User Interface	11
2.2 Model of a UIMS (from [Tanner85])	16
2.3 Model of a Comprehensive UIMS	17
2.4 Transition Networks for a Simple Desk Calculator (from [Jacob85.])	21
2.5 An ipcs Screen (from [Singh85])	25
2.6 Structure of an Event Handler (from [Green85a])	28
2.7 Sequence for Developing Interfaces using the UofA UIMS	29
2.8 An Example Specification for MIKE (from [Olsen86])	30
2.9 MIKE architecture (from [Olsen86])	30
2.10 Renamings of Commands (from [Olsen86])	33
2.11 New Structure of Menus (from [Olsen86])	34
2.12 Overall organization of UIDE (from [Foley89.])	36
2.13 IDL Representation for the Square-And-Triangles Example(from [Foley89])	36
2.14 The Transformed Specification (from [Foley89])	37
2.15 SUIMS in Action (from [Foley89])	40
2.16 Event Based Specification of the Dialogue Control Component	47
3.1 Flow of Information within the UIMS	51
3.2 Communication at Run-Time	55
3.3 Commands in the Skeleton Editing Program	57
3.4 Command Description	58
3.5 Device Description for AED-767	58
3.6 Revised Command Description (skeleton)	60
3.7 Producing the Example Interface	61
3.8 Vu Containing the Default Design	61
3.9 Vu Containing the Refined Design	64
3.10 Completed Interface	64
3.11 Modified Change_Length	67
4.1 Structure of an Event Handler	75
4.2 Run-Time Control	76
4.3 Event Handler for Add_Object Command	78
4.4 Producing Code for PREFIX commands	81
4.5 Producing Code for POSTFIX commands	83
4.6 Producing Code for NOFIX commands	84
4.7 Command Description for the Example System	87

5.1 General Structure of the Dialogue Requirements File ... 95
5.2 Device Description for AED-767 ... 95
5.3 An Example of User's Preferences ... 97
5.4 Example Output of Chisel ... 98
5.5 Forcing the Selection of Interaction Techniques ... 108
5.6 Command Menu Placed Along the Left Edge ... 108
5.7 Menus with At Most Nine Items ... 108
5.8 Overlayed Menus ... 109
5.9 Structure of Chisel ... 110
6.1 A Typical vu Screen ... 116
6.2 Vu Screen with Default Presentation Component ... 122
6.3 Defining Output Tokens ... 123
6.4 Changing Default Token Names ... 123
6.5 Resizing Interaction Techniques ... 123
6.6 Resizing and Repositioning Interaction Techniques ... 124
6.7 The Refined Presentation Component ... 124
6.8 Creating New Interaction Techniques ... 125
6.9 Changing Name of Interaction Techniques ... 125
6.10 Default Attribute Values of Interaction Techniques ... 126
6.11 Changing Attribute Values of Interaction Techniques ... 127
6.12 Logical Organization of vu ... 129
6.13 Flow of Tokens ... 133
7.1 Run-Time Structure of a Typical Interactive System ... 135
7.2 Token Passing Amongst Various Components ... 136
7.3 Steps in the Scheduling Process ... 137
8.1 External Control (adapted from [Thomas83]) ... 141
8.2 Internal Control (adapted from [Thomas83]) ... 141
8.3 Mixed Initiative ... 142
A2.1 Commands Supported by the Distributed Network Editor ... 176
A2.2 Command Description for the UIMS ... 176
A2.3 User's Preferences ... 176
A2.4 Presentation Component Generated by Chisel ... 180
A2.5 Refined Presentation Component ... 181
A2.6 Completed Editor in Action ... 182

Chapter 1

Introduction

1.1. The Problem

The importance of user interfaces for interactive systems is widely recognized. The user interface has become one of the most important bases for marketing systems. As a result, a considerable effort is invested in producing good interfaces. Despite its recognized importance, systems with bad interfaces are still being produced. The reason being that the development of good interfaces is time consuming and expensive in terms of manpower requirements. To help improve this situation researchers have come up with the notion of a User Interface Management System (UIMS) [Thomas83], [Olsen84], [Pfaff85], [Olsen87]. A UIMS is a set of software tools designed to facilitate all aspects of user interface development and management. Over the past few years a number of UIMSs have been developed. Most existing UIMS, however, handle only a part of the user interface development.

The cost of developing an interface can be divided into two parts. The first part is the result of the time and effort expended in creating the initial design of the interface. And, the second part relates to the implementation of the initial design and its successive refinements. The majority of the existing UIMSs have concentrated on reducing the second part of the cost. These UIMSs help in reducing the cost of developing user interfaces by reducing the time and effort required for programming the interface. The user interface designer first designs the interface and then uses a UIMS for implementing it. The main emphasis in the UIMS is on facilitating the implementation of the design.

The interface designers spend a great deal of time and effort in creating the initial design. It is well documented in the literature how expensive this process can be [Morgan83,Smith82]. Typically, while

---

Previous papers about this research are [Singh87], [Singh88b], [Singh88a], [Singh89b], and [Singh89a].

creating the initial design, interface designers are concerned with macro level decisions, such as the choice of interaction techniques and parsing sequence for commands; and micro level decisions, such as the size, location, and colour of each interaction technique, and default and initial values of command arguments. The cost incurred in the initial design can be significantly reduced by automatically creating the initial design, or by helping the interface designer in doing so. To date, little work has been done in this direction. The goal of this work is to develop an approach and a means to automatically design and implement user interfaces. We restrict attention to graphical user interfaces (GUIs) only. This eliminates command language type interfaces from consideration. The coverage, however, remains wide enough to be interesting and of practical use.

1.2. Automating the Design of Graphical User Interfaces

It is useful to divide the user interface into various parts. This helps us organize the discussion better and concentrate on one part of the system at a time. Based on the Seeheim model [Green85b], a user interface is divided into three components; the presentation component, the dialogue control component, and the application interface model (see figure 1.1). The presentation component is responsible for accepting user input and passing it to the other components of the interface, and presenting application output to the user. The dialogue control component is responsible for managing the dialogue between the user and the application. The application interface model is a representation of the functionality of the application from the user interface's view.

In this work we devote attention to the issues involved in the design and implementation of the presentation and the dialogue control components. The presentation component for a GUI presents a graphical front-end to the user of the system. To provide input to the system, the user interacts with techniques which have graphical appearance. And, the system provides application output to the user in terms of the images displayed on the screen. The design of the presentation component for a GUI involves determining graphical techniques for entering user input, and determining how to represent the application output in a graphical manner. For this thesis, we will ignore the issue of automatically determining the presentation of

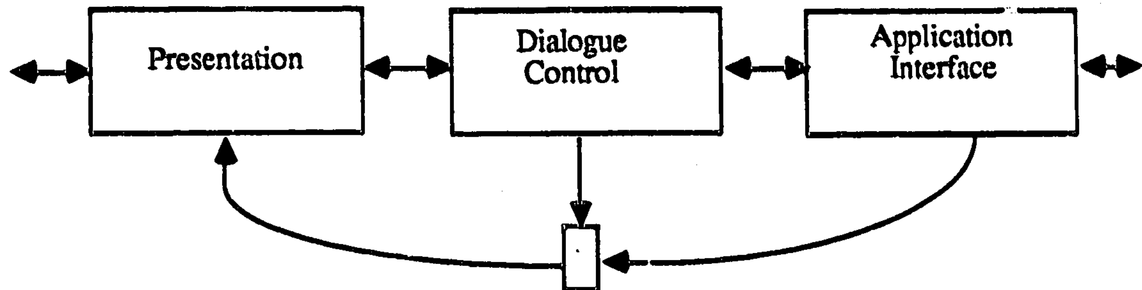


Figure 1.1 Seeheim Model of User Interfaces

the application output, and assume that this part of the design is handled by the interface designer.

The dialogue control component for a GUI is responsible for checking the syntactic correctness of the user's interactions with the presentation component of the interface. Therefore, designing the dialogue control component involves designing a system to do so.

1.2.1. Designing the Presentation Component

The presentation component for a GUI enables the user to enter input through interaction with graphical interaction techniques, and see application output in terms of images on the display screen. As stated earlier, we will be concerned with the input side of the presentation only. Designing the presentation component involves determining interaction techniques (ITs) that make up the user interface, determining IT parameters, and placing ITs on the display screen. The decision as to which ITs can be used depends upon the dialogue requirements, the availability of ITs, the user's preferences, and the hardware for implementing the interface. Each of these concerns limits the set of ITs that can be used.

After selecting the ITs used in the user interface, the next step is to decide on the attributes of the ITs. Each IT has a set of attributes, such as location, size, and colour. Suitable values for these attributes have to be found. The ITs may have constraints on the values the attributes can take. For example, an IT may

constrain its location on the display screen.

After the selections are made and attribute values are determined, the ITs are placed on the display screen. The placing of ITs on the screen depends upon the IT requirements, user's preferences, and the physical dimensions of the display screen.

1.2.2. Designing the Dialogue Control Component

The main responsibility of the dialogue control component is to check the syntactic correctness of the user's actions. Designing the dialogue control component involves designing a system to match the sequence of user's actions with the acceptable sequence(s). In GUIs, a command often has multiple correct sequences. That is, the user is allowed some flexibility in the way he can sequence his actions. The system must be able to recognize such cases and handle them correctly.

An important requirement for the dialogue control component is to support a variety of syntaxes. User interface designers often use prefix, postfix, and nofix (or free-form) syntaxes for parsing commands. For a command of the form

command (arg1, arg2)

the prefix syntax means selecting the command before selecting the arguments. The arguments could be selected in the sequence of their specification or in an arbitrary sequence. The postfix syntax for the command would mean selecting the arguments followed by the command. The nofix syntax means following the prefix, postfix, or any arbitrary sequence for selecting the arguments and the command.

Another issue related to command syntax is the open-ended selection of commands. Open-ended commands accept an arbitrary number of complete arguments. For example, the interface could support an Add-Object command which, once selected, allows multiple objects to be added, one after the other, until another command is selected. A typical input sequence for the above command would be

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

Another syntactic consideration is the use of default and initial values for command arguments. There is no apparent difference in default and initial values in graphical user interfaces. These terms will be defined later and used where helpful.

1.2.3. Refining the User Interface

In the current state of the art in user interface development the initial design undergoes a number of refinement cycles before it is finalized. The refinements are necessary to fine-tune the interface to take into account the users' feedback [Buxton80], [Swartout82], [Mason83], [Boies85]. In this approach the user interface is developed based on the initial design, and evaluated by examining the performance of users using the system. The results of the evaluation are used to refine the initial design. This cycle is repeated until a "satisfactory" interface is produced. Several systems with highly rated user interfaces were developed using this approach. Examples include the Xerox Star [Smith82], the Apple Lisa [Morgan83], the electronic mail system described in [Good84], and the Olympic Messaging System [Boies85].

It is useful to follow a similar approach here, and allow the designer opportunities for "creative doodling" and fine-tuning the interface. Both, the presentation as well as the dialogue control components may need refining by the interface designer. Since the presentation component is inherently graphical, it is best to support its refining through a "what-you-see-is-what-you-get" (WYSIWYG) mode of editing. It is however difficult to come up with an easy visual way of refining the dialogue control component. The problem lies with determining a suitable graphical representation, or visual metaphor, for the dialogue control component which is easy to understand and use for the interface designer. State Transition Diagrams have been

used for a long time to handle this [Newman68], [Parnas69], [Jacob83], [Wasserman85]. But, as will become clear later this notation is not suitable for all types of dialogue. Therefore, the alternative is to change either the input or the output of the system that designs the dialogue control component. In our system it is easier to change the input.

1.3. Contributions

The primary goal of this thesis is to address a key problem with UIMSs: their inability to help reduce the cost of initial design of user interfaces. Because of this inability, existing UIMSs require the interface designer to work at very low levels of syntactic and lexical details, which can be very time-consuming and expensive in terms of effort required. Also, the detailed design produced by the designer must be provided to the UIMS in a notation that it can process, which makes the UIMS difficult to use and increases the chances of errors as the interface descriptions tend to be quite large.

The approach followed in this thesis to tackle this problem is to automatically produce the initial design and implement it, and then enable the interface designer to improve its appearance and/or effectiveness by refining it. The interface designer, in this approach, works at the conceptual level of the user interface and produces a high level description of the commands the interface is to support. Based on this description the syntactic and lexical levels of the interface are automatically designed and implemented. This interface can be refined by the designer to improve the resulting interaction with the user.

This approach is a novel contribution to the field of UIMSs, and takes the state-of-the-art beyond the capabilities of a "conventional" UIMS. It forms the basis of a powerful UIMS that can reduce the cost of developing user interfaces tremendously. Ours is amongest the first UIMSs to follow such an approach.

Three key elements of a UIMS that supports this approach have been developed. These are:

- A system, called **Chisel** (Creating highly interactive screen layouts) for designing graphical presentation components. The presentation components designed by Chisel consist of a collection of interaction techniques.

- A system, called **Diction** (**Dialogue Control Generation**) for designing dialogue control components. Diction is capable of producing dialogue control components suitable for implementing prefix, postfix, and nofix command syntaxes.
- A system, called **vu** (**visual user-interface design workshop**) for refining graphical presentation components designed by Chisel. Vu is a highly interactive visual programming environment which enables User Interface designers to refine presentation components without having to deal with conventional programming.

Both Chisel and Diction demonstrate that a part of the designer's work can be automated, leaving the designer with more resources to devote to other parts of the interface. The designs produced by Chisel and Diction are automatically implemented, which eliminates the need to program or to write the design in some special notation for the UIMS.

Other contributions of this thesis include the following:

- As far as we know, Chisel is the first system to handle the design of graphical presentation components at a detailed level. It is concerned with high level decisions, such as the selection of interaction techniques, as well as low level decisions, such as the size, location, and colour of interaction techniques.
- Chisel demonstrates that it is possible to generate presentation components which are sensitive to:
  - user's prefences
  - user interface designer's guidelines, and
  - hardware devices.
- As far as we know, Diction is amongst the first systems to demonstrate that a variety of syntax types (prefix, postfix, and nofix) can be automatically designed and implemented.
- Diction demonstrates that prefix, postfix, and nofix type syntaxes can co-exist in the same interface, and can be successfully parsed.

- Vu demonstrates that it is possible to refine highly interactive and graphical presentation components without using conventional programming.
- The run-time environment allows parameters to be passed to interaction techniques at run-time. This greatly increases the flexibility of the system and the range of interfaces that can be supported.
- By restricting the range of data produced by the interaction techniques, the presentation component does a part of the job of the dialogue control component. This results in increased efficiency of the generated interface.
- Chisel and Diction promote structured design and produce well structured code. The presentation component is organized as a collection of interaction techniques, and each command in the interface is implemented as a separate event handler. An interaction technique or event handler is implemented as a separate program module and may reside in a separate file.
- The systems Chisel, Diction, and vu demonstrate that it is possible to provide extremely rapid prototyping. It takes very little time to create interfaces from scratch. For the three-dimensional skeleton editing system discussed in section 3.3 of chapter 3, it took me nearly two hours. This is around a factor of 28 faster than using a "conventional" UIMS (see section 8.2.2). The interface designer can quickly create different prototypes by modifying system parameters. The ease and efficiency of creating prototypes encourages experimentation, and may therefore result in better user interfaces.
- Significant portions of the presentation component can be changed without affecting the dialogue control component, and vice-versa. This greatly increases the ease of exploring different designs to choose the most appropriate one.
- This research makes a significant step forward in the direction of increasing the ease-of-use of UIMSs. The interface designer is no longer required to deal with detailed user interface descriptions. A high level specification is directly transformed into the interface design and implemented.
- The interfaces designed by our system are consistent in many respects. The consistency in user inter-

faces helps users in learning new systems.

- The significant reduction in the cost of producing the initial design makes it possible to devote more time and resources to refining the interface. This moves the state of the art in user interface development closer to the point where the halting condition for the development becomes the creation of a satisfactory interface, instead of resource exhaustion. The overall result should be an improvement in the quality of interfaces.
- The implementation of the user interface can be physically separated from the implementation of the application routines, yet these two components can communicate at run-time. This separation is an implicit goal of most, if not all UIMSs, but few existing systems achieve it to the extent it is achieved in our system.

1.4. Non-Contributions

This thesis does not attempt to develop a comprehensive UIMS. This is a complex task and needs more resources than we have at our disposal. In particular the parts which are ignored are support for designing the presentation of the application output and the application interface model. The systems developed are prototypes, built to show the key concepts behind the research.

The research reported here concentrates on GUIs only, therefore the issues dealt with are specific to a special class of interfaces. There are multiple reasons for this restriction. The first being that by considering a restricted class of interfaces we can focus our efforts better and be more thorough in addressing the issues involved. Second, there are few similarities between the issues in GUIs and other types of interfaces. This indicates that it is better to address the problems separately.

1.5. Thesis Outline

Chapter 2 describes basic concepts and discusses background material necessary for understanding the rest of the thesis. In particular, the background material covers a brief history of UIMSs, the general structure of UIMSs, some existing UIMSs, and shortcomings of existing UIMSs. A model for a comprehensive UIMS is also discussed in this chapter.

Chapter 3 contains an overview of the system. It begins by describing the overall approach followed, and introduces Chisel, vu, and Diction. How these systems fit together is also explained in this chapter. In order to demonstrate how to create interfaces using the UIMS, this chapter provides a detailed example of exactly how an interface for a three-dimensional skeleton editing system is created.

Chapter 4 is concerned with the design of dialogue control components. The requirements for the design system are discussed, followed by a description of Diction. A number of example are used to show how interface designers use Diction to produce dialogue control components.

Chapter 5 discusses the design of graphical presentation components. The Chisel system is used for designing presentation components. Chisel selects interaction techniques, determines their attribute values, and places them on the screen of the display device. This chapter discusses the working and design of Chisel.

After Chisel has produced the input part of the presentation component, the interface designer refines it and adds the information regarding the output part to it. This is done by using the Chisel's companion system vu discussed in chapter 6.

Chapter 7 discusses the issues involved in, and the structure of the run-time environment. A description of its implementation is also provided.

Chapter 8 summarizes the thesis, and provides suggestions for future work.

Chapter 2

Basic Concepts and Related Research

2.1. Definition of Terms

In the literature related to user interfaces there is often a conflicting use of terms. To remove the confusion arising because of this, we will start by defining the terminology used in this thesis.

We define the *user interface* as the software part of an interactive system that handles the interaction between the user and the application. In order to communicate with the user the system accepts inputs and presents outputs through the user interface (figure 2.1). Other terms commonly used in the literature for a user interface include human-computer interface, man-machine interface, user-system interface, and attention processor.

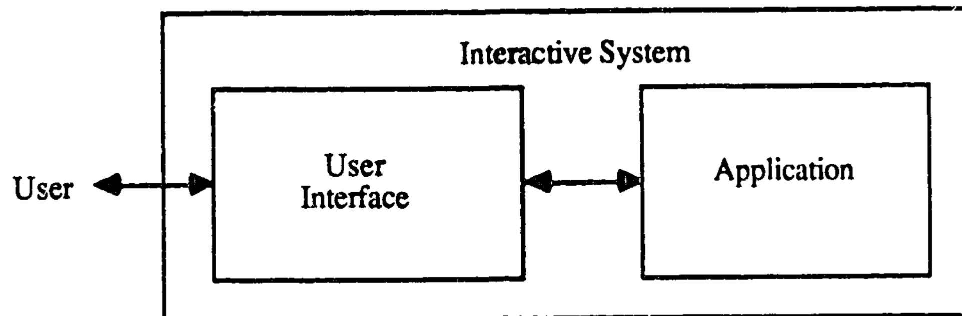


Figure 2.1 User Interface

An *interaction technique* (IT) is defined as a way of using a physical input device to enter a certain type of word (command, value, location, etc.), coupled with the simplest form of feedback from the system to the user [Foley84b]. It manipulates the raw data generated by the interactions to produce more

meaningful information.

A *User Interface Management System* (UIMS) [Thomas83], [Olsen84], [Pfaff85], [Olsen87] is a set of software tools designed to facilitate all aspects of user interface development and management. Most existing UIMSs, however, handle only a part of the user interface development. Other terms used in the literature for a UIMS include Dialogue Management System [Roach82], Abstract Interaction Handler [Feldman82], and User Interface Development System [Hill87a].

There are a number of specialists involved in user interface development. The person who designs the UIMS (e.g., me) is called the *UIMS designer*. The person who designs the user interface is called the *user interface designer*, or simply the *interface designer*. An interface designer, also known as Interaction Designer [Thomas83], Interaction Programmer [Tanner85], or User Interface Architect [Foley84a], uses a UIMS to develop quality user interfaces. An interface designer has software development/design skills, knowledge of human factors, and experience in the area of human-computer interaction [Betts87]. The term *application designer* refers to the person who designs applications. The *application programmer* and application designer will refer to the same person, even though in practice they may be different. This person creates the application which has as its front end the user interface created by the interface designer. The term *user* refers to the end user of the system, the person for whom the user interface is designed.

2.2. Motivation for UIMSs

The realization of the importance of the user interface has lead to the development of some of the most elaborate interfaces. The creation of these interfaces is characterized by large programming and design efforts. A study by Sutton and Sprague [Sutton78] indicated that for interactive business applications the size of the interface in terms of lines of code averaged 59% (varied from 29% to 88%) of the total system. An expert system project [Mittal86] reported that the user interface contributed nearly 40% of the total code. Much effort is also involved in designing good interfaces. Interface designers devoted about 30 work-years to the design of the Xerox Star's user interface [Smith82]. The design of the Apple Lisa's inter-

face also required a great deal of effort [Morgan83].

The cost of developing a quality interface is further compounded by the need to build a number of prototypes leading to the final product. The most often recommended approach to building good user interfaces involves creating and testing prototypes with end users, and modifying the design based on the results of the tests. This approach is called the "iterative design approach" and is suggested in [Buxton80], [Swartout82], [Mason83], [Sheil83], and [Boies85]. The iterative design approach has been used in the creation of several highly rated interfaces: the Xerox Star [Smith82], the Apple Lisa [William83], the electronic mail system described in [Good84], and the Olympic Messaging System [Boies85].

In the absence of appropriate design and implementation tools, the creation of interfaces consumes large amounts of money and time, and leads to the development of software which is hard to debug and modify. This leads to the resources becoming the critical factor, limiting the number of iterations the design can go through, thus forcing the creation of less than satisfactory interfaces.

A UIMS facilitates the creation of good user interfaces by alleviating some of the problems discussed above. It supports rapid prototyping and also makes the creation of the final user interface easier and cheaper. A UIMS can be viewed as a productivity tool which reduces the cost of developing user interfaces tremendously. The Apple MacApp has been reported to reduce the development time by a factor of four or five [Schmucker86]. For some examples Peridot can reduced the development time by a factor of 50 (cf. [Myers87a] page 12).

2.2.1. Advantages of UIMSs

The advantages of using UIMSs include the following.

- The use of a UIMS reduces the cost of developing the user interface by providing tools for increasing the productivity of the people involved in user interface development.
- UIMSs encourage experimentation by facilitating rapid prototyping. This may result in improvements in the quality of the interface.

- The use of a UIMS provides a more consistent interface both within and across applications. This may expedite the learning of new systems.
- Once the UIMS is debugged, the software it generates should be more reliable than the hand-coded software.
- The UIMS provides a suitable environment for exploiting the skills of various specialists involved in user interface development. It does so by representing and supporting the proper roles of various people involved in the development.
- It facilitates the distribution of functionality across systems and processors.
- The separation of the management of user dialogues from the application and the graphics package should allow a better exploitation of various physical devices and interaction techniques.
- There should be cost savings due to the reduced construction cost and the increased usability of the product.

2.3. A Bit of History

The earliest system which had a structure and supported the facilities expected of a UIMS appears to be Reaction Handler developed by Newman [Newman68]. It provided a special language, called the Network Definition Language in which the transfer of control within the program, in response to the user's actions, was specified. The language was based on a finite state machine model. Other parts of the system were written in procedural languages. The structure of Reaction Handlers is remarkably similar to a number of UIMSs developed many years later, and the basic techniques used in it (state transition networks) are still being used. The Reaction Handler is the ultimate ancestor of the UIMSs based on state transition networks.

The next major development after the Reaction Handler appears to be LANG-PAK developed by Heindel and Roberto [Heindel75]. As opposed to the Reaction Handler which dealt with graphical interaction, LANG-PAK aimed at keyboard based interaction. LANG-PAK accepts a BNF-like specification of

the interaction language with embedded calls to application routines. The specification can be tested during development, without the need to supply the application routines. The interface of LANG-PAK was developed using LANG-PAK itself. LANG-PAK appears to be the ultimate ancestor of the UIMSs based on BNF.

There was a remarkable increase in the research and development of tools for user interface development after the mid 1970's. This is evident from the number of workshops and conferences, and the increasing number of systems being reported. The first important event appears to be the Workshop on Methodology of Interaction, also called Seillac II, held in May 1979 in Seillac, France [Guedj80]. The next important workshop, called the Workshop on Graphical Input Interaction Technique, was held in Battelle, Seattle in June 1982 [Thomas83]. The term "User Interface Management System" was coined at this workshop. The next workshop, called the Workshop on User Interface Management Systems was held in Seeheim, FRG in November 1983 [Pfaff85]. The Seeheim Model of user interfaces [Green85b] was developed at this workshop. The next workshop devoted to software tools for user interface development and management was named the Workshop on Software Tools for User Interface Management and was held in November 1986 in Battele, Seattle [Olsen87]. In addition to these workshops a number of conferences focusing on human-computer interaction were also organized. Annual conferences are now sponsored by Eurographics and the ACM Special Interest Group on Computer-Human Interaction (SIGCHI). The ACM Special Interest Group on Graphics (SIGGRAPH) annual conference also holds a special session on the research related to UIMSs, and some of the best work in this area has been reported in this conference.

The most notable systems reported in the literature include Hanau and Leronovitz's prototyping tools [Hanau80], a BNF-based system [Reisner81], TIGER [Kasik82], UofT UIMS [Buxton83], SYNGRAPH [Olsen83], a state-transition UIMS [Jacob83-Jacob85], COUSIN [Hayes83], GRINS [Olsen85], UofA UIMS [Green85a, Singh86], Trillium [Henderson86], GWUIMS [Sibert86], Peridot [Myers86-Myers87b], a state-transition UIMS for direct-manipulation interfaces [Jacob86], MIKE [Olsen86], Sassafras [Hill86-Hill87b], and UIDE [Foley87a, Foley87b, Foley88b, Foley89].

2.4. Structure of UIMSs

Section 2.1 defines a UIMS as a tool or tool set that facilitates all aspects of user interface development and management. But in practice this term is used for any tool that facilitates the creation of any part of the user interface. Figure 2.2 shows a model of a UIMS by Tanner and Buxton [Tanner85]. This model identifies three main phases of a UIMS, namely pre-processing or preparation, run-time support, and post-processing or follow-up. The preparation phase involves the creation of the user interface definition in terms of interaction techniques. The role of the UIMS in this phase is to provide access to the library of interaction techniques and provide the administrative support required to bind these techniques to each other and to the application. The Interaction Technique Builder is a program that creates interaction techniques. Examples of systems which help in creating interaction techniques include Peridot [Myers86], Panther [Helfman87], and Squeak [Cardelli85]. Icon Builder is somewhat similar to a paint or sketch system, and is used for creating icons. Examples of icon builders include [Furuya87] and [Mussio87]. It should be noted that in this model the selection of interaction techniques and their attribute value determination is the responsibility of the interface designer.

The run-time support component provides the mechanism for executing the user interface. It handles the communication with the user and within the system. The run-time support component also records information (such as, time and location) about user interactions and user errors, which is used during the follow-up to discover errors in the interface and to fine-tune the interface to increase the efficiency of interaction. Virtually none of the existing UIMSs provide this component.

There are a number of UIMSs that fit this model very well. But this model is somewhat outdated as it assumes that the interface designer is solely responsible for interface design, and that the UIMS can neither design the interface nor provide any assistance to the interface designer in doing so. Our research shows that it is possible to include this activity as a part of the UIMS. The model of a comprehensive UIMS (figure 2.3) should, therefore, include the following four phases:

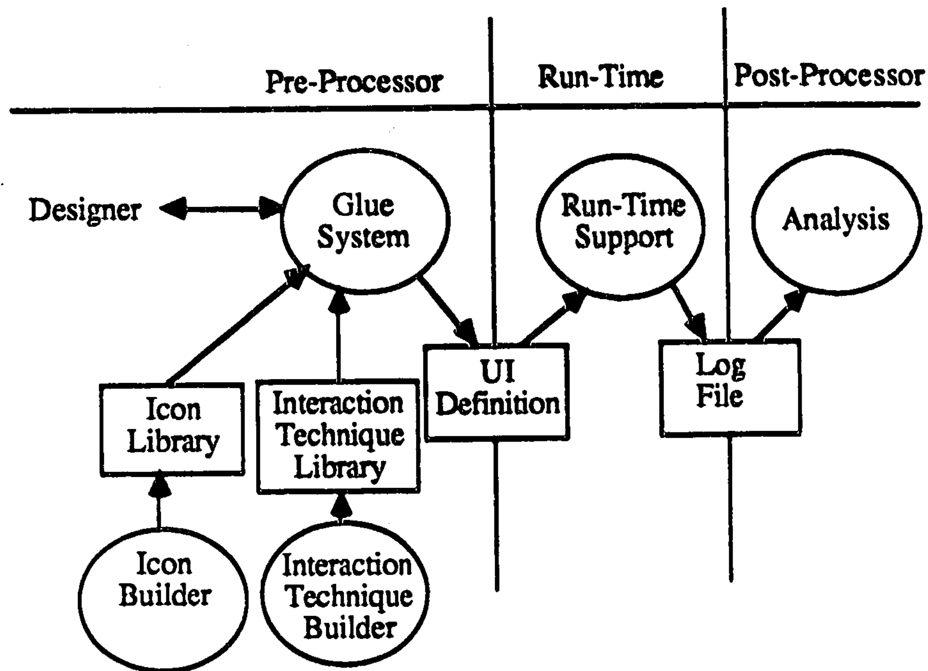


Figure 2.2 Model of a UIMS (from [Tanner85])

- Design
- Generation
- Run-Time Support
- Follow-up

The design phase supports tools for designing various components of the interface, or tools which help the designer in designing the interface.

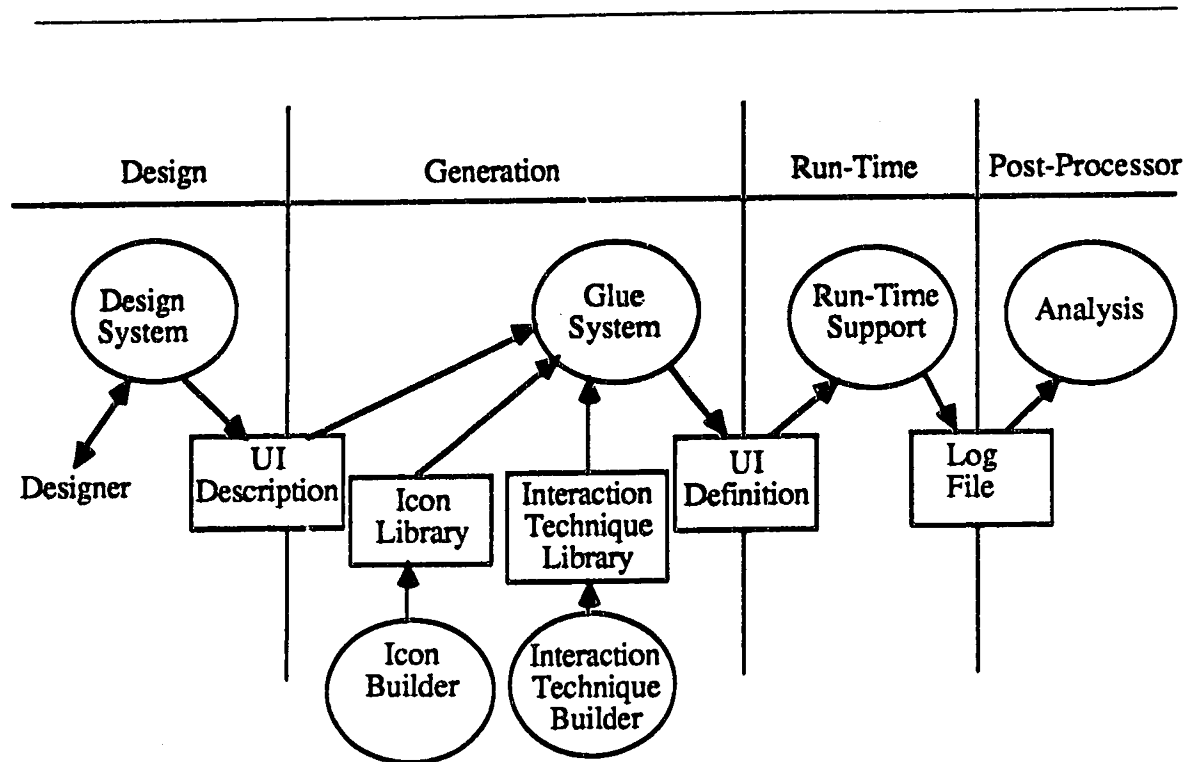


Figure 2.3 Model of a Comprehensive UIMS

2.5. A Survey of Existing UIMSs

Most UIMSs concentrate on supporting only a part of user interface development, some concentrate on supporting screen layout while others concentrate on supporting dialogue control. These systems can be classified into various groups depending on the main technique used for specifying or developing the interface.

2.5.1. Libraries or Toolkits

There are a number of systems that provide libraries of routines that can be used while developing the interface. These systems cannot be classified as UIMS because they lack the structure of a UIMS and do not provide the support expected of a UIMS. But these systems do reduce the cost of developing user interfaces. Examples of systems in this category include SunView and Macintosh Toolbox.

There are a number of limitation of this approach. First, to use libraries a knowledge of programming is essential. This is a major limitation of this approach. Another limitation, related to the first one is that the person developing the interface must also be familiar with other tools, such as editors, compilers, and linkers, which he must use in order to build an interface. The next limitation of this approach is that to test or modify a prototype the code must be compiled and executed or interpreted, giving rise to a long feedback loop.

2.5.2. Object-Oriented UIMSs

UIMSs in this group provide a kernel of graphical objects that can be used directly or customized by the interface designer for specialized application. The objects are arranged in a taxonomic hierarchy in which procedures and data can be shared through inheritance. The saving in cost occurs because of the reuse of code. Often these systems are accessible only to programmers.

Examples of systems in this group include MacApp [Schmucker86], GWUIMS (George Washington User Interface Management System) [Sibert86], and GROW (GRaphical Object Workbench) [Barth86].

2.5.3. Grammar Based Systems

Since user-system interaction can be viewed as a dialogue between two parties, grammars, to many researchers appear as a good way of describing it. A number of user interface description techniques based on grammars have been developed. Many interesting techniques are based on BNF (Backus Naur Form).

When BNF is used to describe user interfaces the terminals correspond to the primitive actions the user can perform. These actions include pressing a button, moving the mouse, or typing a character string. The non-terminals are used to structure these primitive actions into phrases and commands. Reisner [Reisner81] made use of BNF to describe the user interface of two functionally equivalent slide layout programs. SYNGRAPH (SYNtax directed GRAPHics) [Olsen83] is another system which uses interface descriptions in extended-BNF to produce programs in Pascal.

One of the main advantages of BNF is that the specification of the dialogue can be automatically checked for inconsistencies, ambiguities, and incompleteness. Another advantage is that parser generators can be used to automatically produce a part of the user interface given a grammar for it.

The main disadvantage of BNF is that it provides facilities for describing only the user actions, which is only half of the dialogue. The grammar needs to be extended to provide facilities for describing the system actions as well. Ben Shneiderman extended the concept of BNF to multiparty grammars [Shneiderman82]. A multiparty grammar is essentially a BNF grammar where the non-terminals may be labeled by the party which generates them.

2.5.4. Transition Network Based Systems

The earliest use of transition networks to describe user interfaces is by William M. Newman [Newman68]. He used transition networks to specify the transfer of control within the program in response to the user's actions. Other parts of the user interface were written in procedural languages.

A transition network consists of a collection of states and transitions connecting one state to another. The states can be represented as circles and transitions between them as labeled arcs. In their simplest

form, the arc labels represent the actions that the user can perform. These actions are expressed in terms of the hardware devices used by the program. For example, an arc label could be pressing a particular button, or moving the light pen.

Each transition network has a start state and one or many termination states. The start state represents the state of the system before the user starts interacting with it. When the user performs a legal action the system moves to a new state. This current state depends upon the old state and the user action. One state may lead to many different states depending upon the user action. If an arc with a label corresponding to the user's action does not exist then the user has committed an error. The termination state represents a state that takes the user out of the transition network.

The transition networks for real user interfaces tend to be quite large and complex. As a result techniques for partitioning the network have been developed. A large transition network can be divided into a main network and a number of sub-networks. The sub-networks are separate networks describing one part of the system. There are two ways in which a sub-network can be referenced. One way is by using the name of the sub-network as an arc label. In order to traverse the arc, the sub-network must be traversed from its start to a terminal state. The other way of referring to a sub-network is by making it a state in the main network. When the main network enters one of these states the corresponding sub-network is invoked.

Many extensions have been proposed to add more power to the basic transition network notation. Jacob [Jacob83-Jacob85] uses output tokens to display messages and prompt users for input (see figure 2.4). He also uses conditionals to make arbitrary tests on external variables, which must be true for the transition to take place. Wasserman and Stinson [Wasserman79, Wasserman85] use arcs with no labels to catch user errors. In their notation, at most one arc leaving a state can have a blank label. This arc is traversed if the user action does not match any of the other arc labels. They also extend the transition network notation to include variables to store input information.

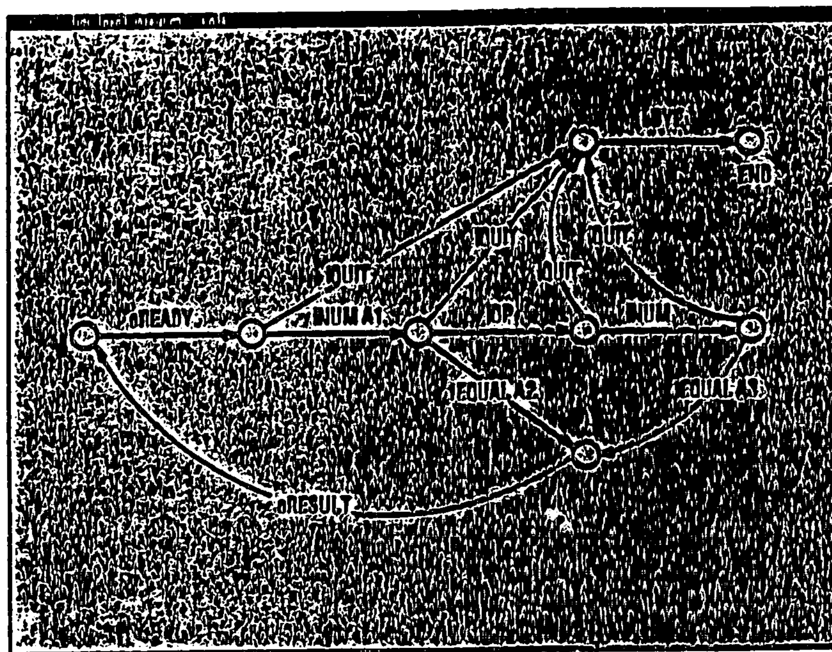


Figure 2.4 Transition Networks for a Simple Desk Calculator (from [Jacob85.])

One of the criticisms of transition network UIMSs has been that since transition networks are most useful when a large amount of syntactic parsing is necessary (or when the interface has a large number of modes, as each state can be viewed as mode), they are not suitable for creating systems, such as direct-manipulation systems, which attempt to be mode-free (c.f. page 21 in [Myers87a]). Jacob argues that direct-manipulation systems, though they appear to be mode-less, in fact have a highly moded structure with many distinct modes [Jacob86]. He treats a direct-manipulation interface as a collection of many relatively simple dialogues which relate to each other as coroutines, and presents a technique where state transition networks are used to specify direct-manipulation interfaces.

2.5.5. Graphical Environments

Many UIMSs provide direct-manipulation facilities for describing the user interface. These environments are quite similar to paint programs, and some of these systems can be used by non-programmers. The focus in these environment is on the screen layout or the presentation component of the interface, as opposed to grammar or state transition network based systems, which concentrate on the dialogue control component. Many UIMSs in this group enable the interface designer to test the interface under construction, whereas others require that the specification be compiled before execution.

The main advantage of systems in this group is that the interface designer is able to see and feel the end product while under construction, thus reducing the feedback time to almost nil. The effects of changes made in the specification are visible without delay. The disadvantage is that not all components of the interface can be described in a graphical fashion.

Menulay (front-end of the UofT UIMS [Buxton83]) enables the designer, using interactive graphic techniques, to define user interfaces which are made up of networks of menus. The specification made by using Menulay is compiled into the C programming language through the use of a companion program, called Makemenu. The semantic routines are written in a conventional programming language.

Ipcs (Interactive Presentation Component Specification) [Singh86], a part of the UofA UIMS [Green85a], provides graphical facilities for creating presentation components for graphical user interface. A presentation component created by ipcs is organized as a hierarchy of windows. With each window the interface designer can associate an interaction technique or a network of menus. The presentation components created by ipcs communicate with other parts of the interface by using input and output tokens. Ipcs is described in greater detail in section 2.6.1.

Peridot (Programming by Example for Real-time Interface Design Obviating Typing) [Myers86-Myers87b] is a User Interface Management System which uses programming by example and visual programming to allow the user interface designer to create interaction techniques. To specify an interaction technique the interface designer demonstrates how it should look and act by drawing the screen display that

the end user will see, and then manipulating the mouse and other input devices to show what the end user will do.

Other UIMSs in this group include GRINS (GRaphical INteraction System) [Olsen85], Trillium [Henderson86], and Cardelli's UIMS [Cardelli87].

2.5.6. Others

There are a number of UIMSs that clearly do not fit into any of the groups described so far. One such systems is TIGER (The Interactive Graphics Engineering Resource) [Kasik82], used for developing engineering applications. TIGER provides a special block-structured language, called TICCL (TIGER Interactive Command and Control Language) for defining and organizing interactive dialogue sequences. A preprocessor compiles TICCL specification into a formatted menu file that is read by the run-time interpreter.

The UofA (University of Alberta) UIMS [Green85a] provides multiple techniques for describing the interface. The presentation component is specified using Ipcs, described in section 2.5.5, and the dialogue control component can be specified in state transition networks (similar to Jacob's system [Jacob83]) or in an event-based notation. The designer is free to choose the technique he finds most appropriate for describing the dialogue control component. The UofA UIMS is described in greater detail in section 2.6.1.

Sassafras [Hill86-Hill87b] is a prototype UIMS based on ERL (Event-Response Language) and LEBM (Local Event Broadcast Method), and focuses on the dialogue control component of the user interface. ERL is a textual language for specifying the syntax of the dialogue and looks similar to programs in production systems. LEBM provides the run-time structure for the interface.

MIKE (Menu Interaction Kontrol Environment) [Olsen86] generates menu based interfaces from the definition of semantic commands that the interaction supports. The default syntax is refined using an interface editor that allows modification of the presentation of the interface. MIKE is described in greater detail in section 2.6.2.

2.6. Some Existing UIMSs

Three UIMSs, the UofA UIMS, MIKE, and UIDE are described in detail in this section. There are a number of reasons for selecting these UIMSs. First, these UIMSs provide a large part of the background required for this thesis, and are, therefore specially relevant. Second, these systems represent the state-of-the-art in the issues they address. Third, these systems are widely known.

2.6.1. The UofA UIMS

The UofA UIMS [Chia85, Green85a, Lau85, Singh85, Singh86] is based on the Seeheim Model of user interfaces [Green85b], which divides the interface into three components: the presentation component, the dialogue control component, and the application interface model. The UIMS provides design tools for each component and a run-time environment which binds the three components together and provides mechanisms for communication.

Designing the Presentation Component

The design of the presentation component can be divided into three activities, screen layout, interaction techniques, and display techniques. These activities are supported by an interactive layout program, called ipcs (interactive presentation component specification) [Singh85, Singh86].

Ipcs (see figure 2.5) enables the designer to divide the screen into a number of overlapping windows. The designer specifies the size and position of a window by pointing at two opposing corners. The designer can then specify the background colour of the window, its coordinate system, a name for the window, and an output token. The output token associated with a window is used to indicate when the window is to be displayed. When the presentation component receives this token the window is displayed on the screen. A network of menus can be associated with each of the windows. A menu can either be static (always displayed in the same position) or pop-up (the current cursor position is the upper left corner of the menu). Each menu is viewed as a collection of menu items. A menu item consists of an input token, and a text string or icon. When the menu item is selected its input token is sent to the dialogue control component.

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

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

Designing the Dialogue Control Component

The UofA UIMS supports two notations, recursive transition networks (RTNs) and event language, for the dialogue control component. This gives the user interface designer some flexibility in his approach to the design of this component. In order to provide this flexibility the UIMS supports a common format that the two notations can be translated into. This common format forms the basis for the run-time support of the dialogue control component. The common format, called EBIF (Event Based Internal Format) is based on the event notation.

Recursive Transition Networks

In the UofA UIMS an interactive approach is taken to the design of recursive transition networks. The interactive transition diagram editor [Lau85] is used to enter and edit RTNs. This editor is based on a graphical display of the transition network. The designer can use a tablet or mouse to enter and edit the nodes and arcs in a diagram. Each arc in the diagram has an input token, and optional output tokens to be sent to the presentation component and application interface model when the arc is traversed. One interesting feature of this editor is the ability to select and save a group of nodes and arcs. This group can then be added to another diagram in the user interface.

The transition diagrams are stored in a database. This database is used to store the diagrams between editing sessions and is used to generate the EBIF for the dialogue control component. A separate program is used to convert the transition diagrams to EBIF.

Select the window for which menu is to be defined using PF1. First menu header and then menu items are created. On header CTRL-a: next menu, CTRL-d: delete menu. On items: CTRL-i: insert item, CTRL-a: append item, CTRL-d: delete item.

Name: w9

Limit llx: 0.00

Limit lly: 0.00

Limit urx: 1.00

Limit ury: 1.00

Bg Color: 0

Draw Color: 1

Bndry Color: 1

Intrct Tech.: none

Output Token: w9

Name: w7

Menu Name: menu

Menu Type: fixed

Coord Choice: system

Orientation: vertical

Output Token: menu

Window Definition

Window Attributes

Menu Definition

Input Token Definition

Output Token Definition

Next Level

Previous Level

EXIT

Help

ON OFF

Window	Parent	Bg Color	Draw Color	Boundary	Int. Tech.	In Tokens	Out Tokens
jup7	jup7	0	1	1	none	0	0

Figure 2.5 An ipcs Screen (from [Singh85])

Event Language

The event language used in the UofA UIMS is based on the C programming language. A program in the event language consists of a number of event handlers. The text of the program contains one or more event handler definitions. When the program is executed instances of these event handlers are created. It is the instances that perform computations, not the event handlers themselves. There may be several instances of the same event handler, parameters can be used to establish the state of an instance when it is created.

```
Eventhandler event_handler_name Is
    Token
            token_name  event_name
                .
                .
                .
    Var
            type variable_name = initial_value;
                .
                .
                .
    Event event_name:type{
            statements
    }
                .
                .
                .
    event event_name:type{
            statements
    }
end event_handler_name;
```

Figure 2.6 Structure of an Event Handler (from [Green85a])

The structure of an event handler declaration is shown in figure 2.6. An event handler declaration is divided into three sections. The first section lists the tokens (either input or output) that the event handler can process. This information is used to map tokens into events for the event handler.

The second section of an event handler declaration contains the declarations of the event handler's local variables. Each instance of the event handler has its own set of local variables, there is no sharing of

storage between instances. A variable declaration consists of a type, a variable name, and an optional initial value.

The third section consists of event declarations. An event declaration starts with the keyword Event followed by the name of the event and its type. The body of the event declaration consists of one or more C statements. These statements are executed when an instance of the event handler receives this event. The statements can reference the instance's local variables and the global variables in the program. The data associated with the event is assigned to the event name before the execution of the statements in the event declaration. The implementation of the event compiler is described in detail in [Chia85].

Designing the Application Interface Model

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

In the UofA UIMS a written notation is used for describing the application interface model. This notation is converted into C code and tables which become part of the user interface at run-time.

The complete sequence for developing interfaces using the UofA UIMS is shown in figure 2.7.

2.6.2. MIKE

MIKE [Olsen86] generates a default syntax from the specification of the semantic commands that the interaction supports. Instead of using detailed syntactic specifications, as is generally the case with many UIMSs, MIKE uses a concise specification of commands the application supports. From this specification, MIKE generates a default interface which is refined by the designer by using an interface editor.

Olsen claims that the syntactic specification required by many UIMSs is the source of most of the

learning problems related to the use of UIMSs. Therefore, the syntactic specification in MIKE is replaced by a high-level specification of semantic commands in the user interface.

The components that make up a MIKE interface specification are (1) the command procedures and functions that define the application semantics, (2) bindings between these procedures and the specific procedures to be used for interactively expressing them, and (3) a set of viewport definitions that define how the screen will appear and how interactive behaviors are attached to visual objects.

Generating the Default Interface

In MIKE the basic definition of an interface is a set of types and a set of functions and procedures that can operate on data objects of those types. In addition to the object types defined for the interface, there are a number of types (integer, real, point, key, and instring) that are predefined and automatically supplied by MIKE. An example specification for MIKE is shown in figure 2.8. This information is entered using MIKE's interactive interface editor.

After the commands have been entered, a working user interface can be generated. Figure 2.9 shows the general architecture for generating interactive programs. First the semantic commands are defined, and then the presentation (screens, function buttons, icons) is added to the commands. The command definitions which are entered using the interface editor are stored in the interface profile, which contains all the information about the user interface. The editor also generates a piece of Pascal code containing definitions necessary for interfacing the application-independent user interface code to the application-specific code that implements the commands given in the interface definition. The generated code is compiled and linked to the application specific code and to the standard MIKE user-interface code to create the interactive application program.

In the default interaction created by MIKE, the functions and the types they return form the basic control mechanism. MIKE creates menus of all commands in the interface based on the type of result returned by the commands. All procedures or functions that return a result of a particular type are placed in

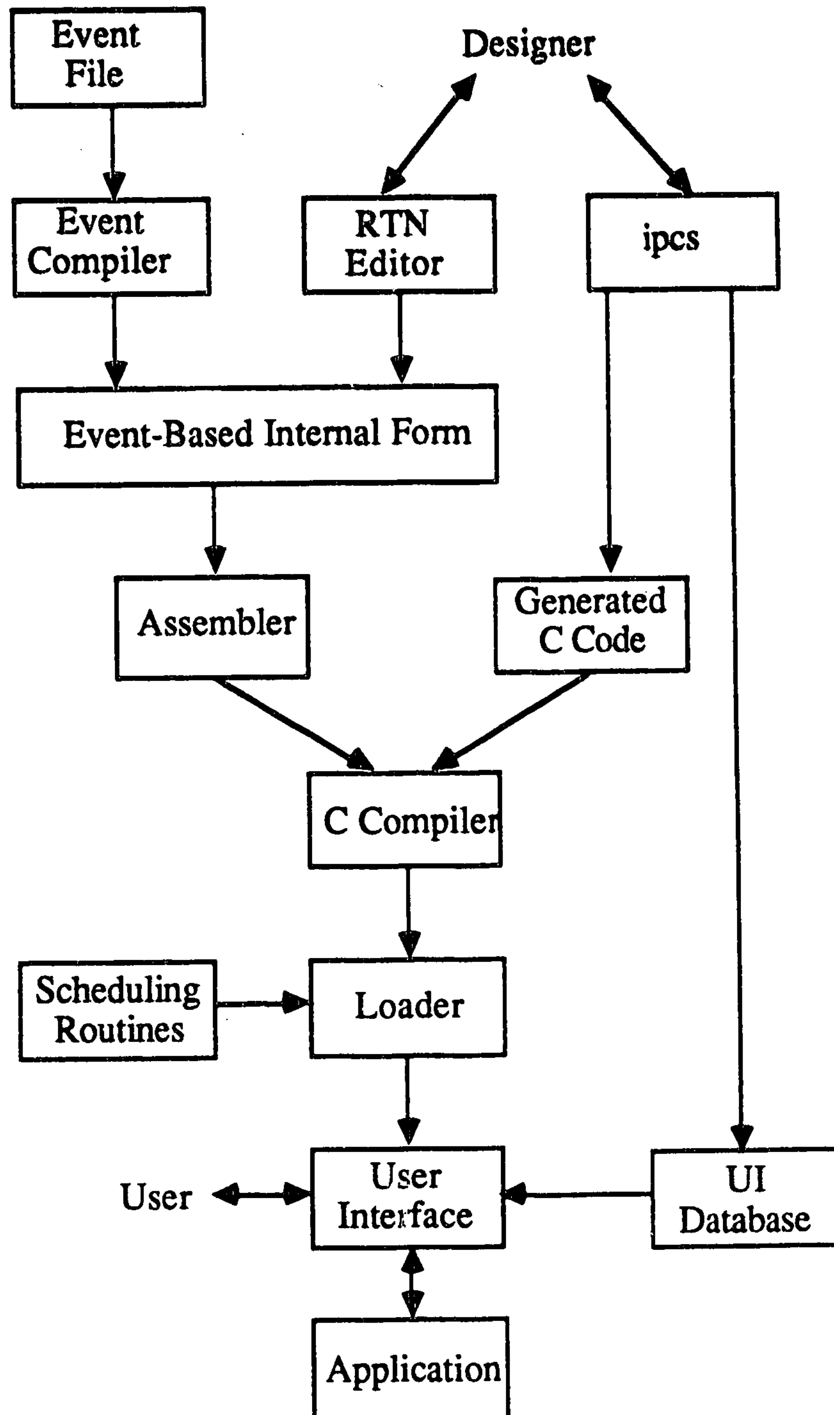


Figure 2.7 Sequence for Developing Interfaces using the UofA UIMS

CreateResistor(Ohms: Integer; C1: Connection; C2: Connection)
CreateCapacitor(Farads: Integer; C1: Connection; C2: Connection)
CreateWire(C1: Connection; C2: Connection)
PickConnection(Where: Point): Connection
If Where is over an existing connection, then that connection is returned.
Otherwise a new connection is generated at Where
PickResistor(Where: Point): Resistor
PickCapacitor(Where: Point): Capacitor
PickWire(Where: Point): Wire
DeleteResistor(R: Resistor)
DeleteCapacitor(C: Capacitor)
DeleteWire(W: Wire)
MoveResistor(R: Resistor; To: Point)
MoveCapacitor(C: Capacitor; To: Point)
MoveWire(W: Wire; To: Point)
ChangeResistance(Of: Resistor; Ohms: Integer)
ChangeCapacitance(Of: Capacitor; Farads: Integer)
ResistanceOf(R: Resistor): Integer;
CapacitanceOf(C: Capacitor): Integer;
SaveCircuit(FileName: InString)
DiscardCircuit
LoadCircuit(FileName: InString)

Figure 2.8 An Example Specification for MIKE (from [Olsen86])

one menu. Using this menu structuring, MIKE parses for command arguments in the specific order in which they are specified in the command definition.

The user can select a command from the current menu either by pointing at it with the locator or by typing a unique abbreviation of the command's name. Once the command is selected, MIKE then prompts for the first parameter of the selected command and fills the menu with all of the functions that return that parameter's type. Using the new menu of functions, the command selection proceeds as before. If the parameter's type is one of those that are predefined, then the appropriate interactive technique (such as, typing in an integer number, typing in a real number) is enabled, in addition to the menu. As the command is parsed in this manner, an echo of the partially constructed command is displayed, along with the prompts.

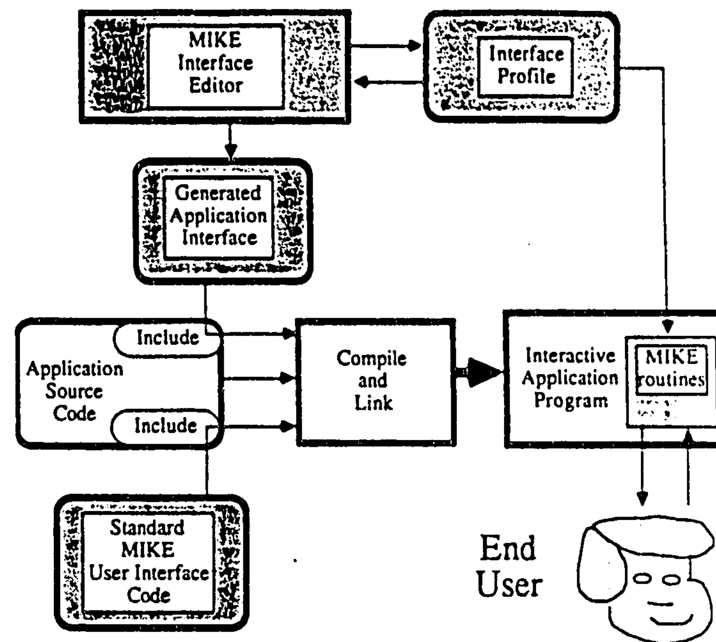


Figure 2.9 MIKE architecture (from [Olsen86])

Refining the Default Interface

The default interface created by MIKE is almost completely keyboard oriented and is not in any way tailored to the needs of the end user. It can be improved by the interface designer by editing the presentation information in the interface profile. The presentation can be improved by restructuring menus, providing help messages, echoes and prompt messages, and improving screen layout.

The default menus generated by MIKE can be too large to fit on the screen. One large menu can be divided into smaller menus organized in a tree structure. This is accomplished by associating external names with commands, which consist of a list of terms separated by periods. These terms form the basis of organizing menus. Figure 2.10 shows renamings for the commands shown in figure 2.8. This renaming is used to define the forest of menu trees shown in figure 2.11. Each menu tree in the forest is identified by

CreateResistor	**=> New.Resistor**
CreateCapacitor	**=> New.Capacitor**
CreateWire	**=> New.Wire**
PickConnection	**=> Pick**
PickResistor	**=> Pick**
PickCapacitor	**=> Pick**
PickWire	**=> Pick**
DeleteResistor	**=> Remove.Resistor**
DeleteCapacitor	**=> Remove.Capacitor**
DeleteWire	**=> Remove.Wire**
MoveResistor	**=> Move.Resistor**
MoveCapacitor	**=> Move.Capacitor**
MoveWire	**=> Move.Wire**
ChangeResistance	**=> Change.Resistance**
ChangeCapacitance	**=> Change.Capacitance**
SaveCircuit	**=> Save**
DiscardCircuit	**=> Remove.File**
LoadCircuit	**=> Load**

Figure 2.10 Renamings of Commands (from [Olsen86])

the type of result that its members return.

Using the interface editor the designer can associate textual prompts, echo and help messages with commands. Up to this point, the interface that has been created is highly textual in nature, and does not support quality graphical interaction. Active viewports are the mechanism that MIKE uses to map visual objects and behaviors on to its normal command parsing syntax. Viewport definitions consist of layouts and action expressions attached to viewports.

MIKE uses a hierarchical system of viewports. Each viewport is uniquely defined by a path name from the root of the viewport tree. Within the interface editor, layouts can be attached to certain viewport names. Layouts consist of graphical primitives and initial placement of subviewports, and can be drawn by using the interface editor.

Nil
New
Resistor, Capacitor, Wire
Remove
Resistor, Capacitor, Wire, File
Move
Resistor, Capacitor, Wire
Change
Resistance, Capacitance
Save
Load
Integer
ResistanceOf, CapacitanceOf
Resistor
Pick
Capacitor
Pick
Connection
Pick
Wire
Pick

Figure 2.11 New Structure of Menus (from [Olsen86])

In order to create a direct-manipulation behavior, action expressions can be attached to viewports. Each viewport definition is assumed to have attached to it a list of action expressions that defines how that viewport is to behave. Whenever the cursor is within a specified viewport and one of the specified events occurs, the action list for that viewport is searched for a command expression suitable for execution. Action lists can also be used in conjunction with layouts to create iconic menus. Each iconic menu items is subviewport of the menu viewport and has a layout(which is the icon) and an action expression attached to it.

2.6.3. UIDE

The development of UIDE [Foley87a-Foley89] took place at the George Washington University around the same time as that of our UIMS. UIDE is a high-level conceptual design tool in which the interface is described as a knowledge base consisting of

- the class hierarchy of objects which exist in the system,
- properties of the objects,
- actions which can be performed on the objects,
- units of information required by the actions,
- and pre- and post-conditions for the actions.

UIDE can algorithmically transform the knowledge base into a number of functionally equivalent interfaces, each of which is slightly different from the original interface. The transformed interface definition can then be input to SUIMS (Simple UIMS) which automatically implements the interface. Figure 2.12 shows the overall organization of UIDE.

The input to UIDE is provided interactively in a language called IDL (Interface Definition Language). The input design information can be automatically checked by UIDE for consistency and completeness. Figure 2.13 shows IDL representation of the knowledge base for a simple square-and-triangles application. Two types of objects - squares and triangles - are subclasses of shape. The attributes of each object are colour, angle, and position. The objects can be created, deleted, and rotated.

UIDE enables the designer to apply transformations on the input to produce designs which are slightly different from the original design. The following generic design paradigms are supported by UIDE.

- Factoring, special cases of which are creating a selected object, a selected command, and a selected attribute value.
- Establishing a selected set as a generalization of the selected-object concept.

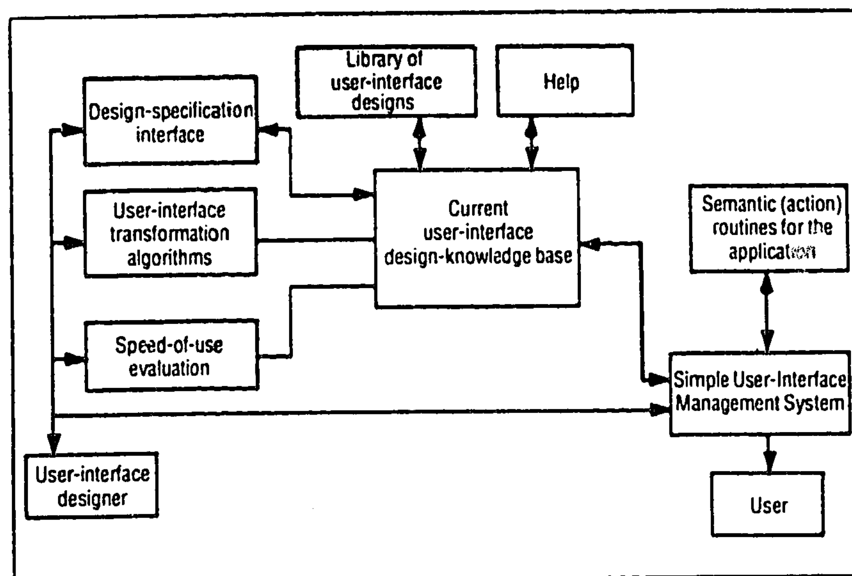


Figure 2.12 Overall organization of UIDE (from [Foley89.])

- Establishing initial values.
- Specializing and generalizing commands based on object and command hierarchies, and
- Modifying the scope of some types of commands.

A transformation is a way to precisely specify the alternatives by simply adding or modifying appropriate pre-conditions and post-conditions or by adding entirely new system-defined actions. Figure 2.14 shows the set of actions that result when the example of figure 2.13 is transformed to have a selected set.

The transformed design of figure 2.14 does not prescribe any presentation style, dialogue syntax, or set of interaction techniques. To define this information, the designer uses SUIMS. SUIMS lets the designer define these aspects of the interface and save them for reuse if further transformations are effected. Figure 2.15 shows SUIMS in action.

```
{object class hierarchy}

type shape
  superclasses : ()
  subclasses : (triangle, square)
  actions : (create_shape, delete_shape, rotate_shape)
  attributes : (position, angle, color)

triangle, square
  superclasses : (shape)
  subclasses : ()
  actions : (create_shape, delete_shape, rotate_shape)
  attributes : (position, angle, color)

{attributes types: Max_X and Max_Y are designer-defined constants. Set(N, M)
gives the cardinality for the choice from the set of enumerated values that fol-
lows. The attribute must have a minimum of N and a maximum of M values
chosen from the set.}

color    : set (1, 1) of (white, gray, black)
angle    : range [0..360] of integer
position : range [x:(0..Max_X),y:(0..Max_X)] of integer

{actions on the objects}

initial: number(shape) = 0

pre-condition: true
  create_shape(obj_type: shape_class, anchor_pt: position, color_value: color,
    orientation: angle)
post-condition: number(shape) = number(shape) + 1

pre-condition: number(shape) > 0
  rotate_shape(obj_type: shape, orientation: angle)
post-condition: none

pre-condition: number(shape) > 0
  delete_shape(obj_type: shape)
post-condition: number(shape) = number(shape) - 1
```

Figure 2.13 IDL Representation for the Square-And-Triangles Example(from [Foley89])

SUIMS uses both the conceptual knowledge base and an additional knowledge base that describes the run-time context for both SUIMS and the application. It also has a set of rules that operate on the knowledge base. The SUIMS window frames control the screen layout with both initial and updated settings. SUIMS provides initial settings, window sizes and positions, and the way new windows will be

```
initial: number(shape) = 0
initial: number(CSS_shape) = 0        {CSS initially empty.}
initial: number(NSS_shape) = 0        {NSS initially empty.}

+   pre-condition: number(NSS, shape) > 0
+     select_shape(obj_type: shape)
+   post-condition: number(NSS, shape) = number(NSS, shape) +
        number(CSS, shape)
+   post-condition: number(CSS, shape) = 1

+   pre-condition: number(NSS, shape) > 0
+     add_to_CSS_shape(obj_type: shape)
+   post-condition: number(CSS, shape) = number(CSS, shape) + 1
+   post-condition: number(NSS, shape) = number(NSS, shape) - 1

+   pre-condition: number(CSS, shape) > 0
+     remove_from_CSS_shape(obj_type: shape)
+   post-condition: number(CSS, shape) = number(CSS, shape) - 1
+   post-condition: number(NSS, shape) = number(NSS, shape) + 1

+   pre-condition: number(CSS, shape) > 0
+     clear_CSS_shape()
+   post-condition: number(NSS, shape) = number(NSS, shape) +
        number(CSS, shape)
    {All selected shapes become nonselected shapes.}
+   post-condition: number(CSS, shape) = 0
    {Size of the selected set of type shape becomes 0.}

    pre-condition: true
      create_shape(obj_type: shape_class, anchor_pt: position, color_value:
        color, orientation: angle)
    post-condition: number(shape) = number(shape) + 1
+   post-condition: number(NSS, shape) = number(NSS, shape) +
        number(CSS, shape)
    {When a shape is created, the selected set of type shape, if any, is deselected,
    thus adding members of that set to the nonselected set of type shape.}
+   post-condition: number(CSS, shape) = 1
    {Size of the selected set after a creation action is 1: the newly created shape.}

+   pre-condition: number(CSS, shape) > 0
    {To rotate a shape, you must first select it.}
    pre-condition: number(shape) > 0
+     rotate_shape(obj_type: shape implicit, orientation: angle)
      {The shape to rotate is implicit in the selected set.}
    post-condition: none

+   pre-condition: number(CSS, shape) > 0
    {To delete a shape, you must first select it.}
    pre-condition: number(shape) > 0
+     delete_shape(obj_type: shape implicit)
      {The shape to delete is implicit in the selected set.}
    post-condition: number(shape) = number(shape) - 1
+   post-condition: number(shape) = number(shape) - number(CSS, shape)
    {Deletion decreases the total by the number of shapes in the selected set.}
+   post-condition: number(CSS, shape) = 0
    {The size of the selected set of type shape becomes 0.}
```

Figure 2.14 The Transformed Specification (from [Foley89])

added when the user creates a window. Once the window is created, SUIMS maintains all the changes (size and position), even if the window is temporarily deleted or made invisible.

SUIMS provides a prefix type syntax in which the command is selected before providing its arguments values, except for arguments that have been factored out by a transformation algorithm and that thus can be set globally. The user can change argument values, as long as the command has not been performed or cancelled.

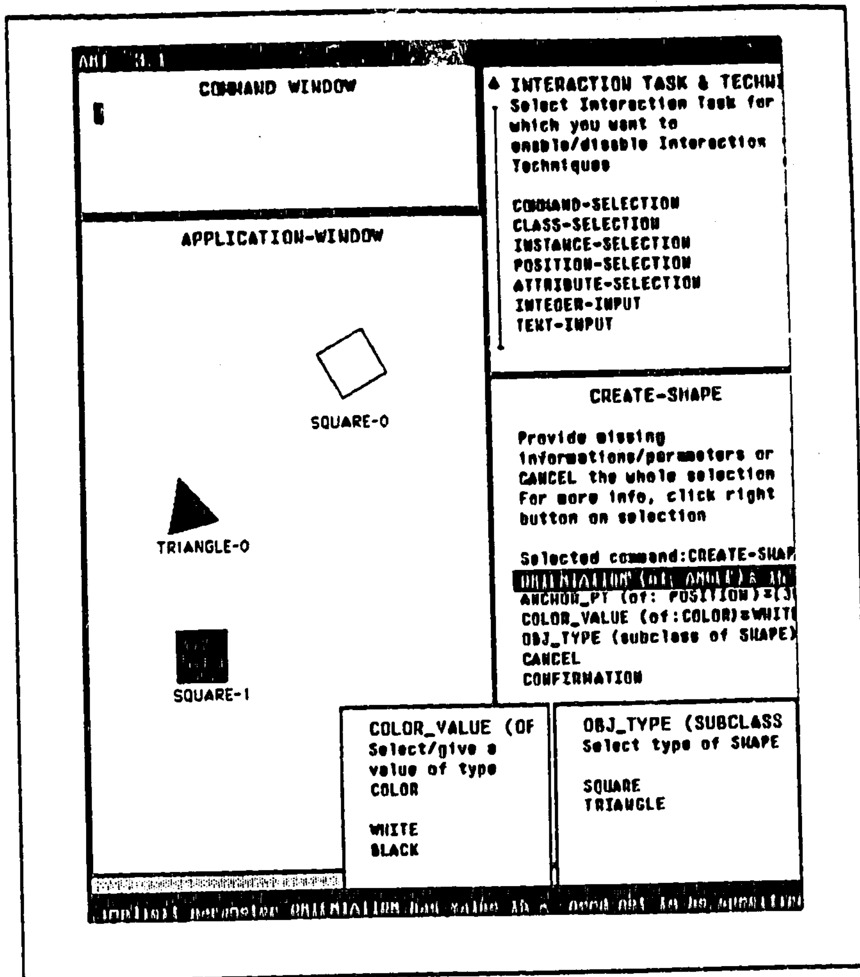
UIDE also implements a keystroke-analysis tool that automates the Card, Moran and Newell Keystroke-Level Model [Card80]. This model predicts how much time a skilled user who makes no mistakes will take to perform a short task using a given design and set of interaction techniques.

UIDE's help system uses the conceptual design and SUIMS run-time knowledge bases to generate context-sensitive help messages. Two kinds of help are provided: explanation of why a command is disabled and of command semantics. The system's explanations of why a command is disabled are based on the unsatisfied pre-conditions for that command. Explanations of what a command does are based on the command semantics as represented by the command's post-conditions.

2.7. Shortcomings of Existing UIMSs

As stated in Chapter 1, the major contribution of this thesis is the development of an approach and means to help in the design of user interfaces. It was demonstrated in Chapter 1 that it is an important problem to tackle. This section will show that existing UIMSs do not address this and related problems adequately.

Consider developing an interface for a geometrical object editor. This example is chosen as it is widely known, fairly easy to understand, and demonstrates the main points of the discussion. This editor enables the user to add, remove, and move two types of geometrical objects, circles and squares, on the display screen. The objects have variable sizes. The editor also supports a help facility, which when turned on, provides descriptive messages about the currently selected command.



SUIMS executing the squares-and-triangles application, with the angle attribute factored out. The application window contains object instances and their names. The command window is used for textual input. The selected action is create_shape; values for all its parameters are provided, as shown in the Create Shape menu window. A parameter Orientation (shown in reverse video) has been selected; additional information for it is at the bottom of the screen explaining that its value is provided by the system (because the attribute Angle is factored out). Anchor_pt was selected by pointing the mouse in the application window, thus there is no text in the command window. Color_value and obj_type are selected from the menus at the bottom of the screen. Confirmation is requested, although you could provide new values for parameters as long as neither Cancel nor Confirm has been selected. You can also rearrange the screen layout or change selected interaction techniques with the menu in the top right corner.

Figure 2.15 SUIMS in Action (from [Foley89])

The designer starts off by determining the commands the editor must support. One of the ways the editor could provide the above functionality is by supporting AddCircle, AddSquare, Remove, Move, HelpOn, HelpOff, and Exit commands. Now the parameters for each command must be determined. The AddSquare and AddCircle commands have two parameters each, indicating the location and size of the object to be placed. The Remove command needs a parameter to identify the object to remove. The Move command needs two parameters, first identifying the object to move, and the second indicating the new location of the object. The HelpOn, HelpOff, and Exit commands do not need any explicit parameters from the user.

The next step determines the flow of information between the application and the interface. This can be represented by input tokens generated by the interface and output tokens generated by the application. The following list of tokens can be used for this purpose:

Input Tokens	Output Tokens
circle	drawcircle
square	drawsquare
remove	erasecircle
move	erasesquare
helpon	
helpoff	
exit	
point	

At this point the designer determines how the information will be presented to the user and how the user will interact with the interface to convey his requirements. This can be divided into two parts, first dealing with the presentation and the second dealing with the dialogue control. In the presentation part, the designer determines how the user will select commands and provide parameters, and how the help information will be presented. Menus are one way of selecting commands. For this simple interface, the designer determines the area, and location on the screen where the menu will be located, which is affected by a number of factors, including the number of commands in the interface, characteristics of the display device, and user's preferences. Other issues to consider at this time include background and drawing colours, and

icons used in the menu.

One of the ways of presenting help information to the user is by using textual messages. The designer must determine where on the screen to present these messages and how much space should be reserved for this facility. The third set of considerations relates to the work area. Size, location, and colours must be determined for the work area.

While designing the dialogue control component for the editor the designer is concerned with the sequence of parameter entry, and sending the information to the application. A number of choices exist at this point, and the designer chooses the one he finds most appropriate. For the example editor, prefix is one possible syntax for parsing commands in the interface.

Now the design of the interface to the application remains to be completed. The designer must determine how to map the information generated by the interactions on to the application.

After all of the steps mentioned above have been completed by the designer, the implementation of the interface begins. First, the detailed design information must be written in a form acceptable to the UIMS. Often this information must be provided in a textual notation. The specification of presentation components includes location, size, and colour of various windows, menus, and other interaction techniques, and other relevant details about these objects. The textual specification of this information is usually clumsy, unnatural and difficult to modify (see, for example [Reisner81] and [Shneiderman82] for BNF-based specification languages for interactive systems). In some UIMSs this detail can be entered graphically, which is easier and faster than the textual specification. The dialogue control component also needs to be specified in a textual notation. This specification includes the syntax of the interaction and calls to the application routines, and for some UIMSs it looks very much like a program. Figure 2.16 shows the specification for the example editor in the event notation used in the UofA UIMS. In some UIMSs this can be specified in visual notations, such as transition networks. To complete the specification, the designer needs to specify the interface to the application. Often a program is written to implement this part. The complete specification of the interface is compiled and linked with run-time support routines from the

UIMS and the application routines to produce a complete system. This system is given to the users for testing and the results are used to modify the interface for a better interaction. Modifying the interface includes modifying the interface specification and regenerating the interface. This cycle may be repeated a number of times till a satisfactory interface is produced.

Two important observations can be made on the role and use of UIMS as an aid in developing interfaces. First, the UIMS provides little or no help while designing the interface. This part of the development is entirely the responsibility of the interface designer. The designer must determine, without any help from the UIMS, high level details, such as which interaction techniques to use, as well as low level details, such as size and location of interaction techniques. The second observation relates to the specification of the interface. The majority of the UIMSs require detailed specification of the interface to be produced. It takes a great deal of effort and time to provide this information, and is error-prone. Often the specification is in a cryptic textual notation, which increases the difficulty of use of the UIMS.

These are two key problems associated with existing UIMSs. To alleviate the first problem, the UIMS must be extended to help in the design of interfaces, which is the aim of this thesis. The second problem can be solved if the extended UIMS can produce the interface directly from the design it creates or helps in creating. This means that the designer does not have to write the detailed interface specifications required by the UIMS. In the UIMS described in this thesis, the design is produced directly in a UIMS-acceptable specification.

```
eventhandler ged is
   token
      point pointE;
      circle circleE;
      square squareE;
      remove removeE;
      move moveE;
      exit exitE;
      helpon helponE;
      helpoff helpoffE;
   var
      int state = 0, point = 1;
      int object, shape, pos;
      int hh = -1;  /*name of the "help" event handler instance*/
   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 ){
         hh = create_instance (help, 0, NULL);
         send_event (hh, helponE, 0);
      }
   }
   event helopoffE{
      if (hh != -1){
         destroy_instance (hh);
         hh = -1;
      }
   }
```

Figure 2.16 Event Based Specification of the Dialogue Control Component
(continued on next page)

```
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;
  }
}
```

Figure 2.16 Event Based Specification of the Dialogue Control Component
(continued from previous page, continued on next page)

```
event eraseE{
    if (shape == 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 ged;

eventhandler help is
  token
    circle circleE;
    square squareE;
    remove removeE;
    move moveE;
    exit exitE;
    helpon helponE;
  event circleE{
    display_message (circle_cmd);
  }
  event squareE{
    display_message (square_cmd);
  }
  event removeE{
    display_message (remove_cmd);
  }
  event moveE{
    display_message (move_cmd);
  }
  event exitE{
    display_message (exit_cmd);
  }
  event helponE{
    display_message (help_cmd);
  }
```

Figure 2.16 Event Based Specification of the Dialogue Control Component

2.8. Chapter Summary

To remove the confusion because of non-standard terminology used in the user interface literature, terms used in this thesis have been defined. The motivation and advantages of using UIMSs have been described. A brief history of UIMSs has been presented, followed by a descriptive structure of UIMSs emphasizing the design phase of user interface development. This structure provides a framework for organizing new UIMSs.

A survey of UIMSs has been presented. The discussion is divided into various groups based on the main description technique used by the UIMS. Two existing UIMSs, the UofA UIMS and MIKE, that represent the current state of the art in the issues they address were discussed. Finally, two key limitations of existing UIMSs were discussed: their inability to help in the design phase, and the need for detailed interface specifications.

The next chapter proposes a solution to the problems associated with existing UIMSs. An overview of the tools built to implement this approach is provided. The remainder of the thesis describes these tools in detail and shows how to use the tools to build interfaces.

Chapter 3

Overview and Extended Example

This chapter describes the basic approach followed to address the inability of existing UIMSs to assist in the initial design of graphical user interfaces. An overview of the UIMS based on this approach is provided, and how various pieces of this system fit together is discussed. These pieces are then discussed in detail in the following chapters.

In order to demonstrate how to create graphical interfaces using this UIMS, this chapter provides a detailed example of exactly how an interface for a three-dimensional skeleton editing system is created.

3.1. The Approach

To overcome the inability of existing UIMSs to help in the design of user interfaces, we have developed an approach in which the initial design of the interface is automatically produced and implemented. This initial interface can then be successively refined. Many existing UIMSs follow a similar approach in which the initial design produced by the interface designer is implemented by using the UIMS, and successively refined by the designer. Our approach is different in that the initial design is produced by the UIMS.

This approach enables the interface designer to work at a higher level of abstraction without worrying about how the interface will be organized and produced. The designer is mainly concerned with the functionality of the application, and produces a high-level description of the commands supported by the application, which includes the names and arguments of commands in the interface. From this description and some additional information about the devices and the end-user of the application, lexical and syntactic design of the interface is automatically produced and implemented. This initial interface can then be improved by the interface designer to increase its usability.

Two important observations can be made on this approach. First, by eliminating the need for the interface designer to produce the lexical and syntactic design, it enables him or her to concentrate on the conceptual and semantic design of the interface. The extra effort devoted to the conceptual and semantic design should result in better interfaces as this part of the design deals with the facilities provided to the user.

The second observation relates to the specification of the design for the UIMS. The majority of existing UIMSs require a detailed specification of the interface to be produced. This specification for a non-trivial user interface tends to be quite large, and hence difficult to produce and error-prone. In our approach, since the design is handled by the UIMS, the specification is produced directly in a UIMS-acceptable form. This eliminates the need for the interface designer to convert the design into a special specification for the UIMS, thus substantially reducing the time required and chance of error.

3.2. The System

A UIMS based on the above approach has been developed and an overview of it is presented in this section.

3.2.1. The Designer's Interface

The components that make up the input to the UIMS are:

1. high-level description of commands supported by the application,
2. characteristics of the device on which the interface is to be implemented, and
3. end-user's preferences.

The last part of the input, namely the end-user's preferences is optional and may be omitted from the input altogether. Based on these inputs the UIMS determines the details of the presentation and dialogue control components of the interface, and produces an implementation of it. The designer then adds the information regarding the application output to the presentation component and provides the routines

implementing the application interface and application semantics to complete the system.

The default interface produced by the UIMS can be refined by the interface designer. Often the refinement involves associating icons with command names, and moving and/or resizing objects on the display screen. The refinements are performed by using an interactive facility. This facility enables the designer to see the effects of refinements without delay and also enables him or her to interact with the interface under development.

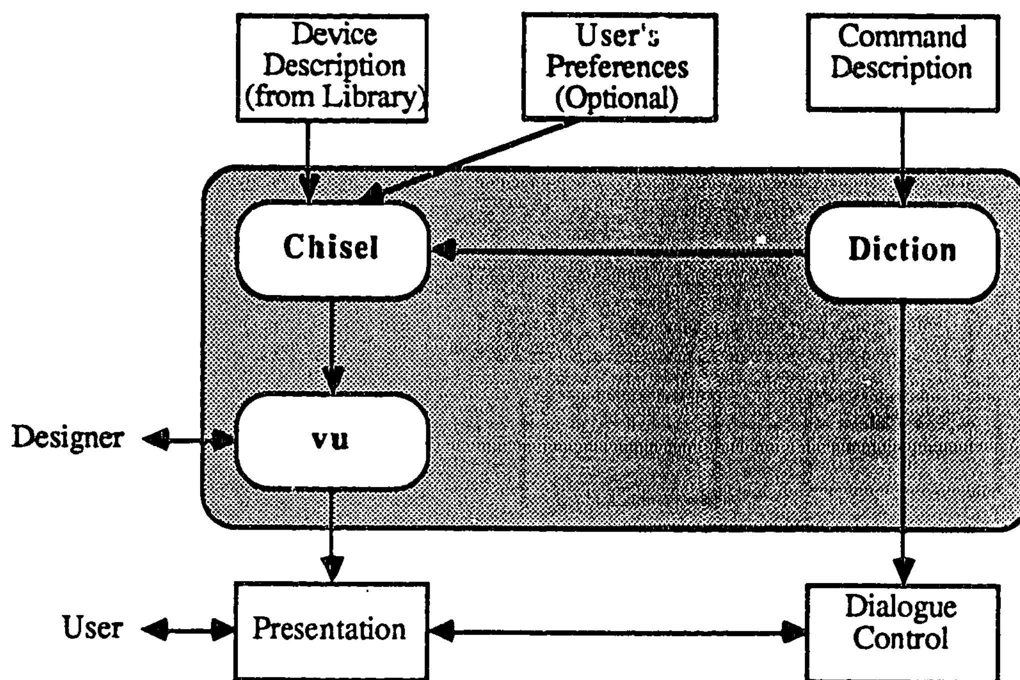


Figure 3.1 Flow of Information within the UIMS

The flow of information within the UIMS is shown in figure 3.1. An overview of various parts of the UIMS is provided later in this section.

3.2.2. Design Principles

One of the chief design goals of our UIMS was that it should enable the interface designer to produce a working prototype of the interface quickly, and enable the designer to refine it. From the interface designer's point of view this translates into being able to produce an implementation of the interface from a minimum amount of detail, and being able to add additional detail afterwards. To realize this goal, the UIMS expects the designer to provide a very high-level description of the commands supported by the application and a brief description of the device on which the interface is to be implemented. From this description the UIMS designs and implements the presentation and dialogue control components of the interface. It is useful to note that the device descriptions for a number of devices can be created once, and the relevant descriptions from this set provided to the UIMS. So the only input the designer must produce afresh is the high-level description of commands supported by the application. The UIMS enables the designer to interact with the interface without having to provide the routines implementing the semantics of the application. This facilitates the parallel development of the application and its interface.

Another design goal of the UIMS was to increase the ease of exploring different alternatives in interface design. The UIMS enables the interface designer to quickly produce different interfaces for the same application. This is achieved by changing the parameters which control the behavior of the UIMS. The designer can not only produce interfaces which look and behave different from each other, he can keep the look the same and change the behavior and vice-versa. This leads to two advantages. First, it greatly increases the number of prototypes that can be built. Second, it reduces the time required for creating prototypes.

One of the important design decisions involved in building the UIMS was the issue of interface designer control versus the UIMS designer control. Many UIMSs impose strict style rules on the interfaces they produce. This considerably restricts the designer's freedom and hence leads to the creation of unsatisfactory user interfaces. We have devoted special attention to allowing the designer to control macro as well micro level behavior of the UIMS. To achieve this the UIMS uses a number of parameters which

determine the behavior of the UIMS. The designer can change the values of these parameters and control the behavior of the UIMS. When the designer does not assign any value to these parameters, default values are used. The interface designer's control of the UIMS is not without its cost in terms of speed of the UIMS. By using parameters, instead of hard-coded options, the UIMS has to sacrifice some speed, but the benefits outweigh the cost in this case.

The next important design decision was the ease of use of the UIMS. Many UIMSs require detailed design specifications before the interface can be constructed. Producing this specification is often error-prone and time consuming. We have eliminated this requirement from our UIMS. The UIMS converts high-level description of the commands directly into the interface. After the initial interface is generated the refinements are performed by using a graphical facility. This facility enables the designer to use graphical techniques to refine graphical presentation components and see the effects of the refinements without delay. More details about how this is achieved are provided in chapter 6.

3.2.3. Overview of the UIMS

The complete UIMS can be divided into four distinct but inter-related subsystems. Each of these subsystems performs a well defined function and interfaces with other subsystems through a well defined interface. The subsystems that make up the UIMS are: Diction, Chisel, vu, and the Run-Time Support. Figure 3.1 shows how the first three of these subsystems relate to each other in the UIMS. The Run-Time Support subsystem provides support routines for the interfaces created by using the UIMS. An overview of the subsystems is provided in the following sections. Each of these subsystems is discussed in detail in the following chapters.

3.2.3.1. Diction

Diction is mainly responsible for producing the dialogue control component of user interfaces and a part of the input for Chisel. It accepts a high-level description of commands in the application, and produces a file which contains procedures (called event handlers) for implementing the dialogue control component. These procedures are produced in a special C-based language, called Event Language. A special assembler converts the programs in the Event Language into programs in the C programming language. The second part of Diction's output forms the input for Chisel. This is a lisp procedure and contains information for creating the presentation components.

3.2.3.2. Chisel

Chisel is responsible for designing graphical presentation components for user interfaces. It accepts as inputs a file of dialogue requirements produced by Diction, a file containing description of the device on which the interface will be implemented, and a file containing user's preferences. The last part of the input is optional, and may be omitted from the input altogether. From this input it produces design information (an ASCII file) for presentation components which can be refined by using its companion system vu.

3.2.3.3. vu

Vu is used for refining the presentation components designed by Chisel. It provides direct manipulation facilities for editing graphical information and shows the effect of changes made by the interface designer without delay. Vu also provides facilities for rehearsing or simulating the interface. The input of vu is the design information produced by Chisel and its output includes a database which is used at application run-time to generate the presentation component and two token definition files which provide the interface between the presentation component and the other parts of the system.

3.2.3.4. Run-Time Support

The run-time support subsystem is responsible for managing the communication between the user and the application, and among various components of the user interface. Figure 3.2 shows how this communication is managed. At run-time, the user of the application is not aware of the presence of this component.

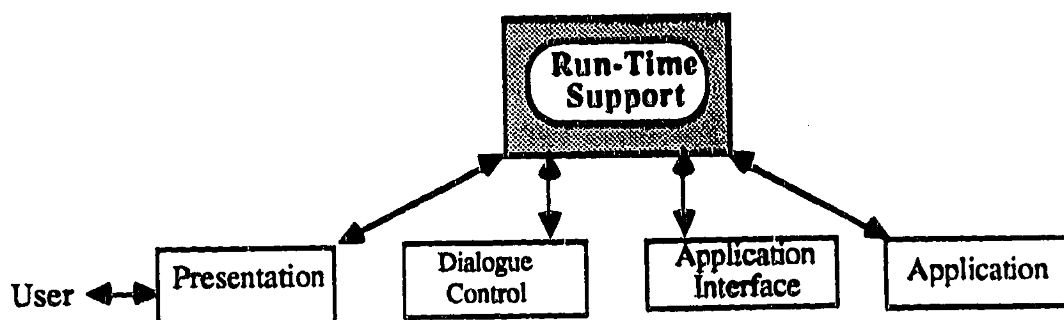


Figure 3.2 Communication at Run-Time

3.2.4. Implementation

The UIMS has been implemented on a VAX-11/780 running UNIX† 4.3BSD. The primary graphical device used for testing the UIMS has been AED-767 colour graphics terminal equipped with a keyboard and a four-button tablet. The subsystem Diction is written in Lex [Lesk75] and Yacc [Johnson75]. Chisel is written in Franz-Lisp, whereas Vu and the run-time support subsystems are written in the C programming language. Vu uses a window-based graphics package called WINDLIB [Green84a] and a graphical database package called FDB (Frame Database) [Green83].

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

3.3. Detailed Example

The best way to demonstrate how designers use our UIMS is through an example. This section presents the sequence of steps that a designer might perform to create an interface for a three-dimensional skeleton editing application. There are a number of reasons for selecting this example. The first being that it represents a non-trivial functionality. The second reason being that it is different from the "canonical" example of paint programs, which has been used for such a long time to demonstrate the capabilities of UIMSs (see e.g. [Green84b], [Olsen86], [Foley87a], and [Hill86]), that it has almost become a drag to read. The third reason being that it shows that the UIMS is being used to create interfaces for various projects in our department. The implementation of the skeleton editing system is used by the animation research group in the department.

The main purpose of this program is to enable the user to interactively enter and edit the skeleton of an arbitrary character. The skeletons created by this program are used to drive the dynamics routines developed by Armstrong and Green [Armstrong85a, Armstrong85b]. The dynamics software views the character to be animated as a tree of limbs. Each limb in the tree has length, mass, center of mass, rotation matrix, and other dynamics parameters. The skeleton editing program allows the user to enter the limbs in the skeleton. This includes identifying the root of the skeleton and how the limbs are connected. This program also allows the user to enter and edit the parameters associated with the limb.

The main function of the program can be divided into two sub-functions: entering skeleton information, and editing skeleton information. The enter skeleton sub-function can be divided into a number of smaller functions. The first function the user must perform when entering skeleton information is to choose the root for the skeleton. This is the three-dimensional position that the tree defining the skeleton is attached to. Once the root has been defined, the user can start entering the topology of the skeleton. The topology covers how the limbs are connected to each other and not the positions of the limbs. For each limb its length, mass, and center of mass must be specified. The length of a limb is a real number that indicates how long the limb is in some unit of measure. The mass of the limb, a real number, indicates how

heavy the limb is in some unit of measure. The center of mass is a fraction that indicates the distance along the limb where the center of mass is located. At this point the use: can specify the initial orientation of the limbs. This orientation is defined in terms of bend and rotate angles which give the orientation of the limb with respect to its proximal joint. If both bend and rotation angles are zero the limb is assumed to be oriented vertically with its distal link below its proximal link.

One of the important parameters associated with each limb is its torque function. The torque function determines the natural range of motion for the joint at the proximal end of the limb. This function gives the torque applied to the joint (by the animation system) as a function of the angle the limb makes with its parent. The user needs some mechanism for assigning these functions to limbs.

The other main sub-function is editing skeleton information. This sub-function involves changing any of the information that was entered in the enter skeleton sub-function. The functions in this sub-function are based on the functions in the enter skeleton sub-function. We need the ability to change the root position. At any point in time there must be one and only one root position. Therefore, we do not need functions to remove and add root positions. For the topology information we need the ability to re-arrange the limbs in the skeleton. This gives rise to add, remove, and move functions for limbs. When a limb is removed or moved all its children are similarly affected. The other parameters associated with the limb and its children remain unaffected. We also need the ability to change the values of length, mass, center of mass, and torque function for a limb. In addition to these functions, we will need a function to display the current values of length, mass, center of mass, torque function, bend angle, and rotation angle for a limb.

The two main sub-functions, enter skeleton and edit skeleton can now be combined to produce the complete task decomposition for the skeleton editing program. Figure 3.3 shows the list of functions that must be supported by this program.

A number of arguments used in the commands can be assigned maximum and minimum values. The center of mass is the simplest to handle, it ranges from 0.0 to 1.0. Typical maximum and minimum values

```
Change_Root (position)
Add_Limb (parent_limb, mass, length, cofm, bend, rotate, torque)
Remove_Limb (limb)
Move_Limb (limb, new_parent)
Show_Attributes (limb)
Change_Orientation (limb, new_rotation, new_bend)
Change_Length (limb, new_length)
Change_Mass (limb, new_mass)
Change_Cofm (limb, new_cofm)
Change_Torque (limb, new_torque)
Save()
Load()
Exit()
```

Figure 3.3 Commands in the Skeleton Editing Program

for length and mass can be used to limit the range of values for these arguments. For this program, assume that the length can vary from 1.0 to 5.0, and mass can vary from 1.0 to 20.0. We can also use a library of torque functions that can be assigned to limbs. The names of these functions can be provided to the user for selection. Since we are dealing with three-dimensional structures, we need some way of entering three-dimensional data and picking three-dimensional objects. We also need some place for showing attribute values for limbs. We can rewrite the command description for this system as shown in figure 3.4.

The device for implementing this system is AED-767. The device description for AED-767 can be selected from the library of device descriptions and provided to the UIMS. The description for this device is shown in figure 3.5. At this point we have sufficient information for producing the interface. The (default) interface produced by the UIMS will allow the user to select commands and provide argument values for the commands through graphical interaction techniques. This interface parses commands in the prefix mode in which the command is selected before entering argument values. The selected command in this interface remains active until another command is selected, and all argument values must be provided afresh each time a command is executed. Clearly, a number of improvements can be made in this interface.

```
/* declare global arguments. Center of Mass (COFM), length of the limb (LENGTH)
   and mass of the limb (MASS) are subranges of real numbers, whereas band
   (BEND) and rotation (ROTATE) angles are subranges of integers.
   TORQUE represents a list of torque functions that the user can assign to limbs.
   LIMB is an interaction technique which implements a 3-dimensional pick.
   The INFO window is used to display limb attributes. */

COFM = [0.0 : 1.0]
LENGTH = [1.0 : 5.0]
MASS = [1.0 : 20.0]
BEND = [0 : 360]
ROTATE = [0 : 360]
TORQUE = (TorqueFunc1 TorqueFunc2 TorqueFunc3 TorqueFunc4
          TorqueFunc5 TorqueFunc6)
LIMB = pick3d;
INFO = window;

/* The commands are declared as follows. Command name is optionaly
   followed by the syntax type, and the selection type of the command.
   After this come the argument declarations. For each argument its
   type, range or enumerations, if any, and default value, if any, is
   specified. In the following declarations none of the commands specifies
   syntax or selection type. */

Change_Root (position : LIMB)
Add_Limb (parent_limb : LIMB, mass : MASS,
          length : LENGTH, cofm : COFM, bend : BEND,
          rotate : ROTATE, torque : TORQUE )
Remove_Limb (limb : LIMB)
Move_Limb (limb : LIMB, new_parent : LIMB)
Show_Attributes (limb : LIMB)
Change_Orientation (limb : LIMB, new_rotation : ROTATE, new_bend : BEND)
Change_Length (limb : LIMB, new_length : LENGTH)
Change_Mass (limb : LIMB, new_mass : MASS)
Change_Cofm (limb : LIMB, new_cofm : COFM)
Change_Torque (limb : LIMB, new_torque : TORQUE)
Save()
Load()
Exit()
```

Figure 3.4 Command Description

```
(defun define-device ()
    (screen    768       ;x resolution
          565      ;y resolution
          8        ;size of a char in x
          12       ;size of a char in y
          64       ;pixels/inch
          'yes)    ;colour?

    (input-devices '(mouse))

    (colour 'red       255     0       0)
    (colour 'green     0       255     0)
    (colour 'blue      0       0       255)
    (colour 'white     255     255     255)
    (colour 'black     0       0       0)
    (colour 'grey      128     128     128)
     .
     .
     .
)
```

Figure 3.5 Device Description for AED-767

The first improvement can be made in entering argument values. For example, for the Add_Limb command we can use current values of length, mass, center of mass, and torque function. What this means is that to add a limb in the skeleton the user must provide only the parent limb value, the rest of the arguments assume the currently selected values from their respective interaction techniques. The second improvement can be made in the command selection. We do not want all commands to work in the close-ended mode. The Add_Limb, Remove_Limb, Move_Limb, Show_Attributes, Change_Orientation, Change_Length, Change_Mass, Change_Cofm, and Change_Torque commands should be open_ended, i.e. the selected command should be active till another command is selected. Figure 3.6 shows the description for this interface after making the above improvements. Let us name the file containing this description as "skeleton".

For this example we will skip the user's preferences part, and produce the interface with defaults (we will come back to this later). Figure 3.7 shows the steps performed in producing the complete system. The

```
COFM = [0.0 : 1.0]
LENGTH = [1.0 : 5.0]
MASS = [1.0 : 20.0]
BEND = [0 : 360]
ROTATE = [0 : 360]
TORQUE = (TorqueFunc1 TorqueFunc2 TorqueFunc3 TorqueFunc4
        TorqueFunc5 TorqueFunc6)
LIMB = pick3d;
INFO = window;

Change_Root (position : LIMB)
Add_Limb {OPEN_ENDED} (parent_limb : LIMB, mass : MASS {CSV},
        length : LENGTH {CSV}, cofm : COFM {CSV},
        bend : BEND {CSV}, rotate : ROTATE {CSV},
        torque : TORQUE)
Remove_Limb {OPEN_ENDED} (limb : LIMB)
Move_Limb {OPEN_ENDED} (limb : LIMB, new_parent : LIMB)
Show_Attributes {OPEN_ENDED} (limb : LIMB)
Change_Orientation {OPEN_ENDED} (limb : LIMB,
        new_rotation : ROTATE, new_bend : BEND)
Change_Length {OPEN_ENDED} (limb : LIMB, new_length : LENGTH)
Change_Mass {OPEN_ENDED} (limb : LIMB, new_mass : MASS)
Change_Cofm {OPEN_ENDED} (limb : LIMB, new_cofm : COFM)
Change_Torque {OPEN_ENDED} (limb : LIMB, new_torque : TORQUE)
Save ()
Load ()
Exit ()
```

Figure 3.6 Revised Command Description (skeleton)

first step is to use Diction to produce event handlers for the dialogue control component and the input for Chisel. The description shown in figure 3.6 is input to Diction, and Diction's output is two ASCII files. The first file implements the dialogue control component and the second file contains the input for Chisel.

In the next step Chisel is used to design the presentation component. To do so we need to provide it with two inputs: the file produced by Diction, and the device description for AED-767. The output of Chisel is an ASCII file containing design information (called "design_file") for the presentation component.

We now use vu to refine the design produced by Chisel and add the output token information to it.

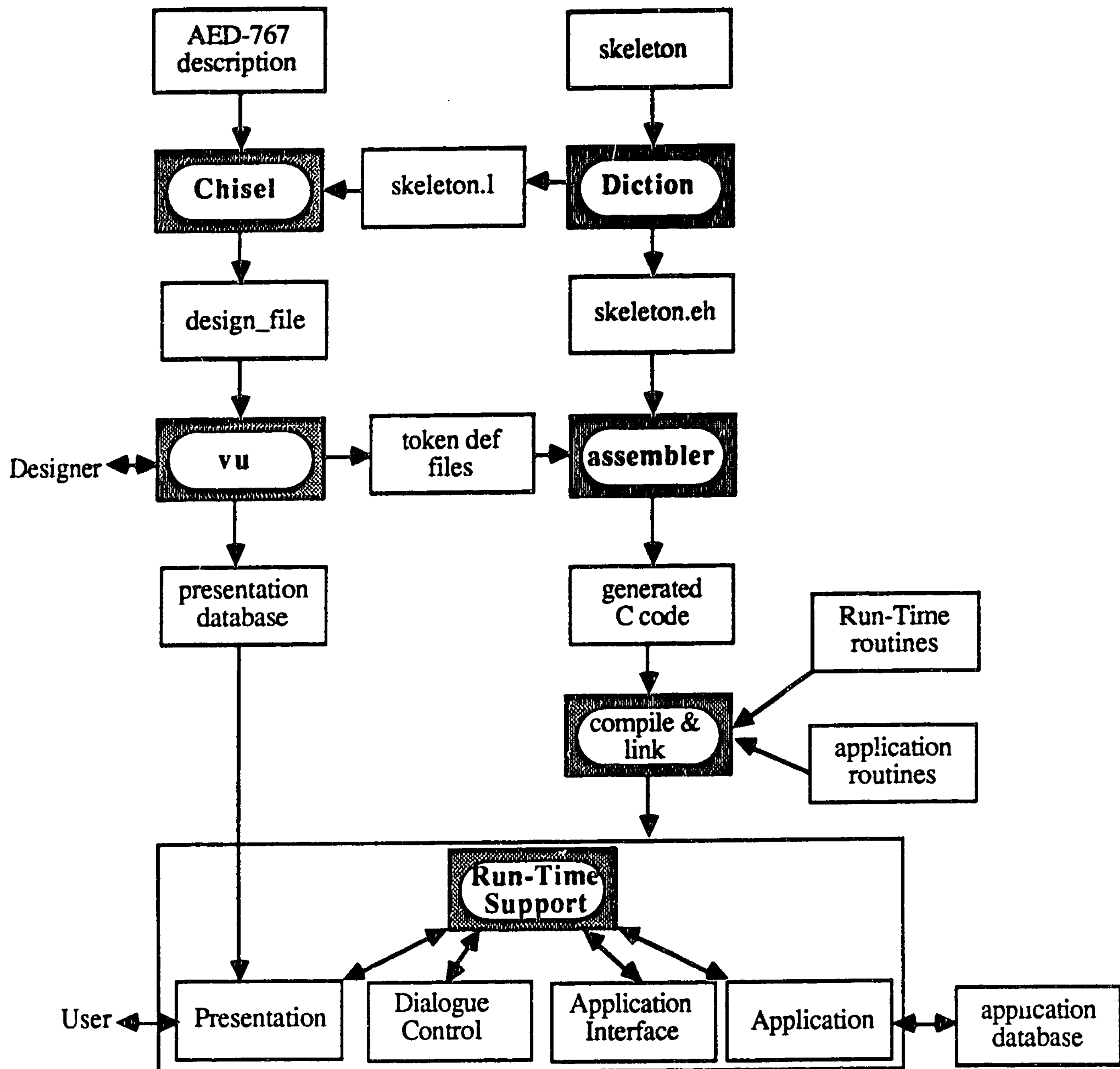


Figure 3.7 Producing the Example Interface

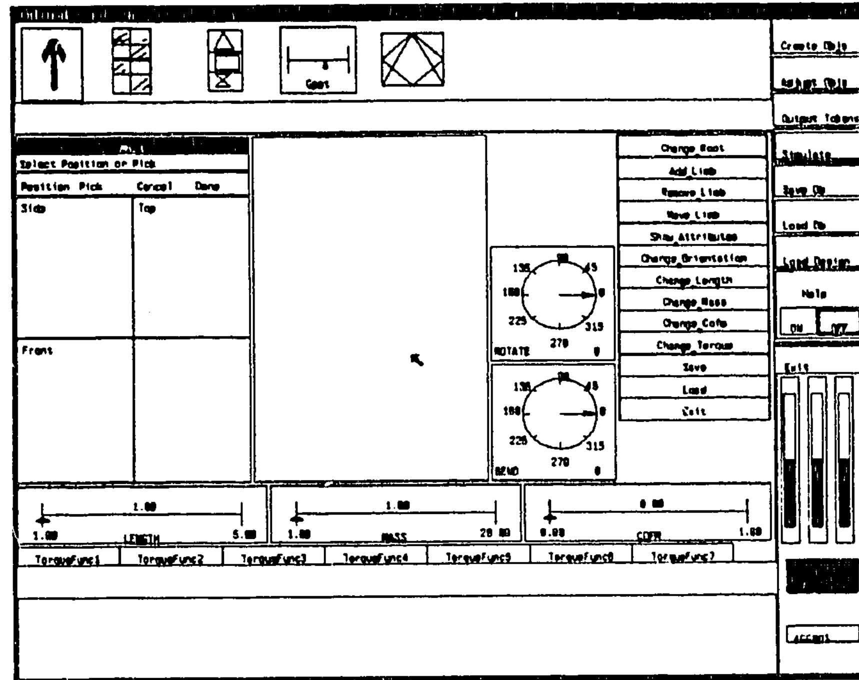


Figure 3.8 Vu Containing the Default Design

The display screen of vu containing the design before refining is shown in figure 3.8. Let us first add the output token information to the presentation component. Table 3.1 shows the token and display procedure names for the example. The window for the skeleton display (LIMB) will receive two output tokens. The first token (SKL) is used for adding new limbs to the skeleton and the second token (DEL) is used for deleting existing limbs from the skeleton. We also need display procedures which transform these tokens into images. We need one output token (INF) and a display procedure (Show) for the window (INFO) which displays limb attributes. The output tokens are associated by selecting the output tokens command from the vu menu and typing in the token and display procedure names.

Table 3.1 Output Tokens

Window	Token Name	Display Procedure
LIMB	SKL	Display
LIMB	DEL	Delete
INFO	INF	Show

We can refine the interface shown in figure 3.8 by moving and resizing objects on the screen. This is done by using direct-manipulation techniques supported by vu. The presentation component after refining is shown in figure 3.9. When we save this design, vu produces a database describing the presentation component and two ASCII files containing the input and output token definitions. The database is used at the application run-time to generate the presentation component, and the token definition files provide an interface between the presentation component and other parts of the system.

At this point we have completed the production of the presentation component. The dialogue control component is still in the event notation. This is converted into procedures in the C programming language by using an assembler. We can now compile the generated code, and produce the complete system by loading the application and run-time routines with it.

Figure 3.10 shows the completed system in action. The window across the bottom of the screen provides help messages about the commands in the interface. When the user selects a command the system prints messages explaining the command and its arguments. These help messages are automatically produced by the UIMS and serve the purpose of reminding the user about how the commands work and their argument requirements. The help facility can be turned off by selecting the Help Off button. It is possible to run the system without the help facility. By typing the -h flag when starting the system, the user can disable this facility for the complete session. In this case the system allocates the complete screen to the application.

Let us now see how this interface behaves as the user interacts with it. When the user selects the Change_Origin command from the command menu, the system highlights the command and prints a help

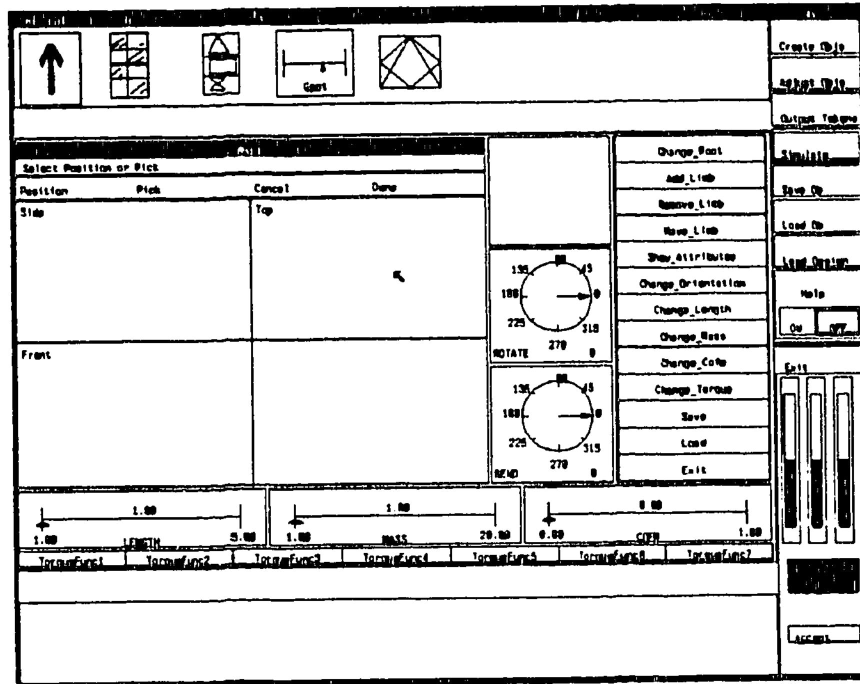


Figure 3.9 Vu Containing the Refined Design

message in the help window. The only argument needed for this command is the three-dimensional point denoting the root of the skeleton. To enter this argument, the user interacts with the interaction technique which implements three-dimensional pick. After the user has entered this argument, the run-time support subsystem informs the application to execute the command, and since the command selection is close-ended, the support system deactivates the command automatically. What actually happens in the run-time system is explained in the following chapters of this thesis. To add limbs in the skeleton the user selects the Add_Limb command. This command has several arguments, but the only argument value the user must provide is the parent_limb. Other arguments assume the current values of the interaction techniques they are tied to. Of course, the user can change these values as well as providing the value for the parent limb. The parent limb is the identifier of a limb in the skeleton. The interaction technique implementing three-

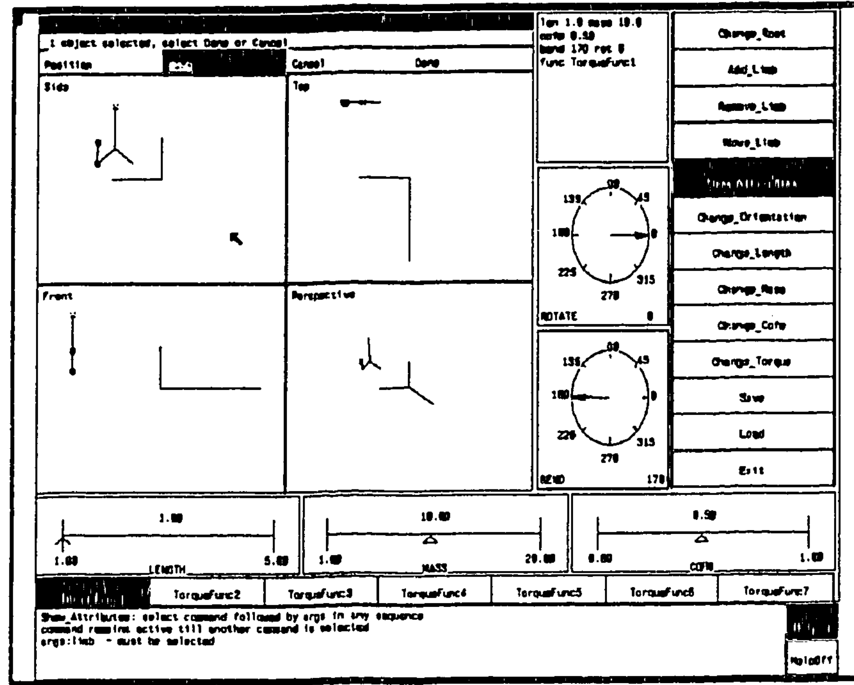


Figure 3.10 Completed Interface

dimensional pick makes it possible for the user to pick limbs by using a two dimensional mouse. It displays three orthogonal views (side, front, and top) of the skeleton. The user can pick limbs from these views, and send the value when the proper limb is selected. The interaction technique helps in performing the pick operation by highlighting currently picked objects. When the users sends the parent_limb value the run-time system asks the application to execute the Add_Limb command. The application updates its data structures and sends an output token to the presentation component to display it. The run-time system determines the appropriate window and display procedure for this token, and executes the display procedure. This display procedure updates the display in the skeleton display window. Since Add_Limb was specified as an open-ended command, it remains active, and the user can enter as many limbs as he or she wants without having to reselect the command.

The Delete_Limb command deletes limbs from the skeleton. This command needs the limb to be deleted as its argument, which is entered in the same way as the parent_limb argument for the Add_Limb command. The application in this case sends a different output token to the presentation component. The display procedure for this token erases the limb and all its children from the display. Since the command is close-ended, it is de-selected after execution. The Move_Limb command moves the selected command to a new parent. The children of the moved limb follow it.

The argument for the Show_Attributes command is entered in the same way as for the Delete_Limb. The difference is in the effect of the command on the display. The command in this case causes the attributes of the selected limb to be displayed in the window reserved for this purpose.

The Change_Length command needs the limb and new_length as arguments. Both the arguments must be entered. The Change_Mass and Change_Cofm commands also work in a similar way. The Save command does not require any arguments. When the user selects this command the dialogue control component asks the application to execute the command. The application saves the skeleton information in the application database. The command is de-selected after execution. The Load command loads the skeleton information from the application database. The Exit command causes the system to stop.

3.3.1. Producing Other Variants

In the previous section we created an interface for the three-dimensional skeleton editing system. With a little extra effort a number of variants of this interface can be produced. There are two ways of producing variants of this interface. First, we can keep one of the components the same and change the other. Second, we can produces new interfaces from scratch which differ from the one we created earlier in both the presentation and the dialogue.

Let us first keep the presentation the same and change the dialogue control. We can change the parsing sequence and the selection mode of commands, and the command argument specification. For example, let us change the Change_Length command to accept the currently selected value of Length. The com-

```
COFM = [0.0 : 1.0]
LENGTH = [1.0 : 5.0]
MASS = [1.0 : 20.0]
BEND = [0 : 360]
ROTATE = [0 : 360]
TORQUE = (TorqueFunc1 TorqueFunc2 TorqueFunc3 TorqueFunc4
        TorqueFunc5 TorqueFunc6)
LIMB = pick3d;
INFO = window;

Change_Root (position : LIMB)
Add_Limb {OPEN_ENDED} (parent_limb : LIMB, mass : MASS {CSV},
        length : LENGTH {CSV}, cofm : COFM {CSV},
        bend : BEND {CSV}, rotate : ROTATE {CSV},
        torque : TORQUE)
Remove_Limb {OPEN_ENDED} (limb : LIMB)
Move_Limb {OPEN_ENDED} (limb : LIMB, new_parent : LIMB)
Show_Attributes {OPEN_ENDED} (limb : LIMB)
Change_Orientation {OPEN_ENDED} (limb : LIMB,
        new_rotation : ROTATE, new_bend : BEND)
Change_Length {OPEN_ENDED} (limb : LIMB, new_length : LENGTH {CSV})
Change_Mass {OPEN_ENDED} (limb : LIMB, new_mass : MASS)
Change_Cofm {OPEN_ENDED} (limb : LIMB, new_cofm : COFM)
Change_Torque {OPEN_ENDED} (limb : LIMB, new_torque : TORQUE)
Save ()
Load ()
Exit ()
```

Figure 3.11 Modified Change_Length

mand specification for the modified interface is shown in figure 3.11. We can now produce the new dialogue control component by using Diction and then assembling and compiling the generated programs. The presentation component need not be generated again in this case. This saves us the time we devoted to creating and refining the presentation component. A number of variants of the interface can be produced and tested in this way, and the one which feels most natural can be kept.

The presentation component can be changed in two ways. The first way is to change the presentation component by using vu. The designer might want to move interaction techniques to different places on

screen, and change their sizes or colours. In this case the designer makes the changes by using vu, saves the database, and runs the system with the new database. There is no need to recompile the system again, as no additional code has been produced. If the new version of the presentation component contains new output tokens and display procedures, the system must be relinked to load new display procedures. The second way of producing new variants of the presentation components is by changing the defaults assumed by Chisel. For example, if the user prefers command menus along the left side of the screen, the designer can produce the user's preferences file and get Chisel to produce a new presentation component. This design can be refined by using vu, and a new version of the system produced without a major recompile. The dialogue control component of the system remains same, the only thing that has changed is the database which contains the presentation component description.

In cases where changes in the dialogue control component affect changes in the presentation component, the complete interface must be produced from the beginning. Cases where this may be necessary include addition of new commands in the interface and addition of new arguments in commands. Such a change is usually required when the conceptual and semantic design of the interface changes.

3.4. Chapter Summary

This chapter has described the approach followed in this thesis to overcome the inability of UIMSs to help in the design of user interfaces, and provided an overview of the UIMS based on this approach. A brief introduction to subsystems of this UIMS has been provided. These subsystems are explained in greater detail in the following chapters.

A detailed description of exactly how an interface for a three-dimensional skeleton editing application can be created has been provided. Methods of producing variants of this interface were also discussed.

Chapter 4

Designing the Dialogue Control Component

To create a user interface for an application using our UIMS, the interface designer first produces a high-level description of the commands supported by the application, and then uses Diction to generate the dialogue control component of the interface. The output of Diction consists of program modules, called event handlers, which implement the dialogue control. In addition to producing the event handlers for dialogue control, Diction produces a part of the input for Chisel, which is used for designing presentation components.

In this chapter, the design and implementation of Diction is presented. How event handlers implement the dialogue control is discussed. A number of examples are used to show how interface designers use Diction to produce dialogue control components.

4.1. Diction

The main responsibility of Diction is to produce dialogue control components for user interfaces. Its secondary responsibility is to produce input for Chisel, which is used for designing presentation components for user interfaces (see figure 3.1 in chapter 3). In this section an overview of Diction is presented.

4.1.1. Design Principles

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

command (arg1, arg2)

the prefix syntax means that the command must be selected before the arguments are selected. The

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

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

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

Diction also enables the interface designer to use default and initial values for command arguments. There is no apparent difference in default and initial values in graphical user interfaces. We put the following restrictions on the arguments. Only globally declared arguments can have initial values. Such arguments are set to their initial values when the interface is first started. The user can change the argument values through interaction with the interface. The arguments which are not global (hence, are command or local level) can have default values. The value of such an argument is set to its default value every time the command containing the argument is selected. This argument value can also be changed by the user. The difference between the use of default and initial values is that initial values are set just once, when the interface is initialized, whereas the default values are set every time the command containing the default arguments is selected.

The second major design goal of Diction was that it should facilitate rapid prototyping of interfaces.

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

The third design goal of Diction was that it should be easy to use. Unlike a number of existing UIMSs, Diction does not require detailed specifications to produce the dialogue control components. It accepts a high-level specification of commands and converts the specification into program modules which implement the dialogue control. Even though the command description is at a much higher level than in many UIMSs, it enables the designer to control the interface behavior to a very low level.

Another design goal of Diction was that it should expedite the production of user interfaces by automating many of the time-consuming activities. One of the ways Diction achieves this is by automatically producing help messages from the command descriptions provided by the designer. The help information produced by Diction explains the syntactic structure of the command and reminds users about the arguments of commands.

4.1.2. Input of Diction

The input of Diction is an ASCII file containing a high-level description of commands supported by the application. This description can be divided into three parts. The first part is optional, and may contain a declaration of parsing sequence and selection type to be applied to commands. The parsing sequence can be one of PREFIX, POSTFIX, and NOFIX. The selection type can be either OPEN_ENDED or CLOSE_ENDED. In the absence of this declaration, Diction assumes default values for parsing sequence, which is PREFIX, and for selection type, which is CLOSE_ENDED.

The second part of the description contains a declaration of all the global arguments used by the commands. For each argument its name, type, and if applicable, range or enumerations and initial value are specified.

The third, and the last part of the description contains a declaration of commands supported by the application. For each command, the description includes its name, optionally followed by its parsing sequence and selection type, followed by argument declarations. The parsing sequence and selection type may take the values described above. The declarations made here override the declarations made in the first part of the input. If a command does not declare its parsing sequence and selection type, the global (or default) values are used. For each argument in a command, its name followed by type, and if applicable, range or enumerations and default value are specified. A command may have a variable number of arguments.

A BNF (Backus-Naur Form) description of the input to Diction is provided in Appendix A1.

4.1.3. Output of Diction

The output of Diction consists of two parts. The first part contains program modules which implement the dialogue control component of the user interface, whereas the second part contains information which is used by Chisel for designing graphical presentation components. Details of the output of Diction are presented in the rest of this chapter.

4.2. The Dialogue Control Component

The dialogue control components produced by Diction consist of program modules called event handlers. The complete dialogue control component consists of a number of event handlers, and each event handler is responsible for one well defined function. For example, an event handler may be responsible for parsing a particular command or for generating help messages. Diction produces one event handler per command, which is responsible for handling default values for the command arguments, parsing the command, notifying the presentation component when errors occur, and notifying the application when the command is successfully parsed. In addition to producing event handlers for commands, Diction produces two event handlers responsible for house-keeping and producing help information for commands. The details of how event handlers implement the dialogue control are provided later in this chapter.

4.2.1. The Event Model

The event handlers are used in the event model for describing dialogue control components†. This model is based on the concept of input events that is found in a number of graphics packages. In these packages the input devices are viewed as a source of events. Each input device generates one or more events when the user interacts with it. An event has an identifier and a number of data values associated with it. The events generated by the devices are placed on a queue, and the application program removes the events one at a time from the queue and processes them.

The event model is an extension of this basic idea. In the event model, there is an arbitrary number of event types, which can be generated by the input devices or inside of the dialogue control component. When an event is generated, it is sent to one or more event handlers. An event handler is a process (defined by a procedure or module) that is capable of processing certain types of events. When an event handler receives one of the events it can process, it executes a procedure. This procedure can perform some computation, generate new events, call application procedures, create new event handlers, or destroy existing event handlers.

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

Once an event handler has been created, it is in the active state. It remains in this state until it is destroyed, either by itself or by another event handler. Only the active event handlers can respond to events. In the event model, a dialogue control component is described by the set of templates that define the event

† This section presents a summary of the event model discussed in greater detail in [Green86].

handlers it uses. At the start of execution an instance of one of these templates is created to serve as the main event handler in the dialogue control component. This event handler will then create (possibly indirectly) all the other event handlers. Conceptually, all the event handlers in the dialogue control component execute concurrently, processing events as they arrive.

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

4.2.2. Structure of Event Handlers

The structure of event handlers produced by Diction is shown in figure 4.1. The keywords in figure 4.1 are printed in bold. An event handler declaration is divided into four sections. The first section declares the name of the event handler. In the event handlers produced by Diction, the event handler name is the same as the name of the command parsed by the event handler; only the case is inverted.

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

The third section of an event handler declaration contains the declarations of the event handler's local variables. Each instance of the event handler has its own set of local variables, there is no sharing of storage between instances. A variable declaration consists of a type, a variable name, and an optional initial value. In the event handlers produced by Diction, this part of event handlers is rarely used.

The fourth section consists of event declarations. An event declaration starts with the keyword "event" followed by the name of the event. The body of the event declaration consists of a number of C statements. These statements are executed when an instance of the event handler receives this event. The statements can reference the instance's local variables and the global variables in the program. How event

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

Figure 4.1 Structure of an Event Handler

handlers are used to implement the dialogue control is explained in the rest of this section.

When the user interacts with the presentation component, input tokens are produced. These tokens are added at the end of a token queue reserved for the dialogue control component. The run-time control removes these tokens from the front of the queue, one at a time, and processes them. Processing a token involves sending the event corresponding to the token to the event handlers (see figure 4.2). An event handler receives only those events which it can process.

When the application is first started, the run-time control instantiates two event handlers. The first event handler is called the HOUSE_KEEPER. It remains active throughout the interactive session, and its main responsibilities include initializing interaction techniques, instantiating event handlers for commands, and maintaining data-structures used by the dialogue control component. The second event handler, called HELPER, generates help messages for commands selected by the user. The discussion on how help is

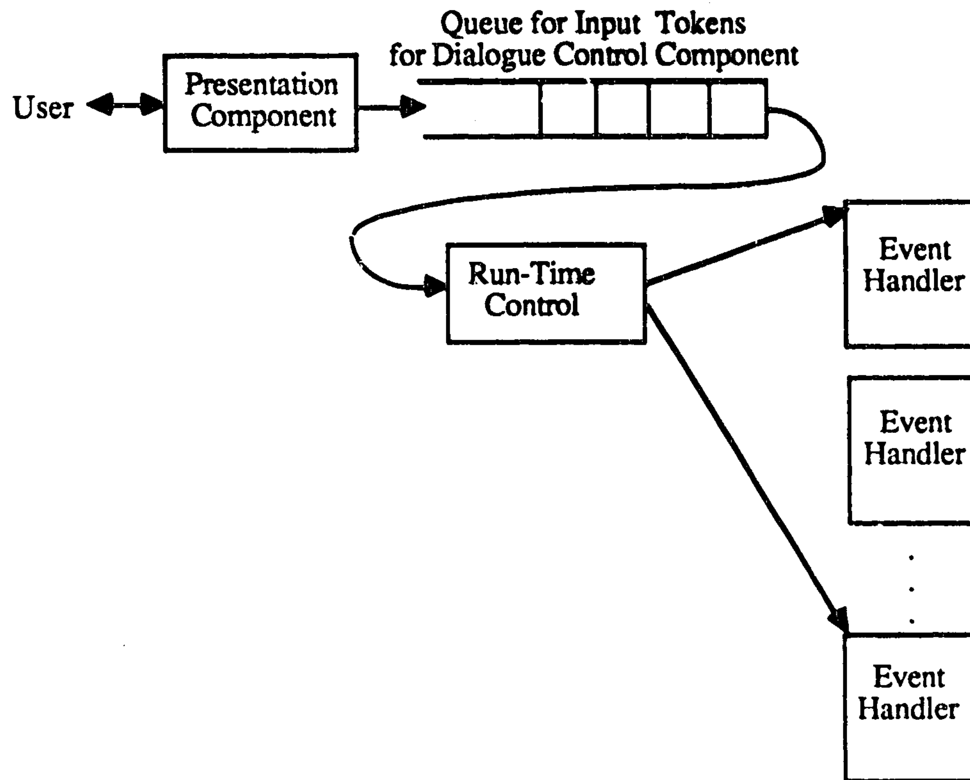


Figure 4.2 Run-Time Control

implemented is postponed until section 4.4.3. None of the event handlers corresponding to the commands in the application have been instantiated at this time.

The input tokens generated by the presentation component are sent to the HOUSE_KEEPER and if activated, to the HELPER. When the user selects a command, an input token identifying the command is sent to the dialogue control component. The run-time control converts this token into an event and sends the event to the HOUSE_KEEPER and the HELPER. The HELPER generates a number of output tokens for the presentation component which produce help messages for the selected command. The

HOUSE_KEEPER, on receiving the event, instantiates the event handler for the command selected by the user. On instantiation, an INIT event is sent to the newly created event handler automatically by the run-time system. The new event handler, with some help from the HOUSE_KEEPER, is responsible for parsing the command.

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

For example, consider an Add_Object command, which accepts two arguments, size and position. Both the arguments must be provided afresh by the user to execute the command. Assume that the command is parsed in PREFIX OPEN_ENDED fashion. The event handler produced by Diction for this command is shown in figure 4.3. In the event handlers produced by Diction, argument names provided by the designer are replaced by the argument names prefixed with the command name they belong to. This is necessary in order to create unique argument names.

When the user selects the Add_Object command, its event handler shown in figure 4.3 is instantiated, and an INIT event is sent to it automatically. On receiving this event, the Add_Object event handler sets the status of size and position arguments to UNDEF. When the user provides either the size or the position value, corresponding events (IN_SIZE or IN_WORK) are sent to the HOUSE_KEEPER by the run-time control. For each of the events, the HOUSE_KEEPER updates the status and value of the corresponding argument, and generates a CHECK token. The CHECK token triggers the Add_Object event handler to determine whether the command can be executed or not. When both arguments are defined, the

```
Add_Object {PREFIX OPEN_ENDED}
  (position : WORK_AREA; size : SIZE)

eventhandler aDD_oBJECT is  /* PREFIX OPEN_ENDED */
  token    /* token declaration */
    CHECK  IN_CHECK;
  event INIT{
    Add_Object_position.status = UNDEF;
    Add_Object_size.status = UNDEF;
  }
  event IN_CHECK{
    if (Add_Object_position.status == DEF && Add_Object_size.status== DEF){
      values = (int *) calloc( 2, sizeof( int ));
      values[0] = Add_Object_position.value;
      values[1] = Add_Object_size.value;
      send_token( APPLICATION, INPUT, Add_Object, values);
      Add_Object_position.status = UNDEF;
      Add_Object_size.status = UNDEF;
    }
  }
end aDD_oBJECT;
```

Figure 4.3 Event Handler for Add_Object Command

Add_Object event handler generates a token for the application. After doing so it resets the status flags for size and position to UNDEF. The Add_Object event handler remains active, enabling the user to repeatedly execute the command without having to reselect it. This event handler is killed by the HOUSE_KEEPER when the user selects another command.

4.3. Producing HOUSE_KEEPER

The HOUSE_KEEPER is mainly responsible for three activities. The first activity is to initialize interaction techniques when the interactive application is started. The second activity is to control the instantiation of event handlers for parsing commands. And the third activity is to maintain the status and values of command arguments.

To initialize interaction techniques for global arguments the HOUSE_KEEPER generates output tokens (of type INITIAL) which are sent to the presentation component. The interaction techniques, after

initializing themselves, generate input tokens for the dialogue control component. On receiving these tokens the dialogue control component (actually the HOUSE_KEEPER) updates the status and values of command arguments which are tied to the tokens.

The HOUSE_KEEPER also produces an INITIAL token for the application. On receiving this token, the application initializes its data structures and sends some setup information to the presentation component.

When the user selects a command, an input token identifying the command is sent from the presentation component to the dialogue control component. After conversion into an event, this token is sent to the HOUSE_KEEPER. On receiving the event, the HOUSE_KEEPER first destroys command event handlers that may be active at that time. Secondly, the HOUSE_KEEPER instantiates the event handler for parsing the selected command. The new event handler is automatically sent an INIT event by the run-time control system.

The third main activity of the HOUSE_KEEPER involves maintaining information for command arguments. When the HOUSE_KEEPER receives input tokens (or events) for arguments, it updates the status of command arguments tied to the received token (or event) to DEF and stores the event_value (or the token value) in the argument values. The HOUSE_KEEPER never changes the status of an argument to UNDEF; this is done by the event handlers for commands.

The HOUSE_KEEPER updates the argument values and status only; it does not determine whether the selected command can be executed or not. This is done by the event handler for the command. The HOUSE_KEEPER, every time it receives an event for a command argument, generates a CHECK token. This token is sent to the command event handlers which determine whether the command can be executed or not.

4.4. Producing Event Handlers for Commands

Depending on the parsing sequence and the selection type of the command, Diction produces event handlers to parse it. In this section, three algorithms, one each for PREFIX, POSTFIX, and NOFIX syntax types, are presented. Based on these algorithms, Diction produces event handlers to implement the dialogue control.

The algorithms are presented in a pseudo-code. In the algorithms, the command arguments which have currently selected values are called CSV-type arguments, whereas the ones which have default values are called default-type arguments. The arguments which are neither CSV-type nor default-type are called regular-type arguments.

```
1. Generate code for INIT event
   for all default-type arguments
      produce tokens for presentation component
      to initialize interaction techniques.
   for all regular-type arguments
      set argument.status to UNDEF.
2. Generate code for IN_CHECK event
   if all argument values are available
      produce a list of argument values.
      send token to application.
      2a. Reset status of other command arguments
      for all CSV-type and regular-type arguments
         produce a set S of tokens the arguments are tied to.
      for all POSTFIX and NOFIX commands
         for all regular-type arguments
            if argument is tied to a token from the set S
               set argument.status to UNDEF.
      2b. Handle selection type
      if command selection is CLOSE_ENDED
         deactivate_self.
      else
         for all default-type arguments.
            produce tokens for presentation component
            to initialize interaction techniques.
         for all regular-type arguments
            set argument.status to UNDEF.
```

Figure 4.4 Producing Code for PREFIX commands

For a PREFIX command, the user first selects the command, and then provides its arguments. The entry of arguments is unordered. The algorithm for generating event handlers for commands of this type is shown in figure 4.4. Diction first produces code for the INIT event, which gets executed immediately after the event handler is instantiated. In this part of the code, tokens for initializing default-type arguments are produced, which are sent to the presentation component. The status flags of all regular-type arguments are also set to UNDEF in this part of the code.

The rest of the event handler code handles the CHECK token generated by the HOUSE_KEEPER. On receiving an input token corresponding to a command argument, the HOUSE_KEEPER updates the status and values of arguments tied to the token and generates a CHECK token. The CHECK token after conversion into an IN_CHECK event is sent to the active command event handler. On receiving the IN_CHECK event, the event handler checks whether the values for all the arguments are available or not. If all the values are not available, the event handler simply returns. Otherwise, the application is notified by sending a token which identifies the command. A list of argument values is also sent to the application along with the token.

The HOUSE_KEEPER, when it receives an argument entry token, updates the status of all the arguments tied to the token. Some of these arguments should now be reset to UNDEF as the purpose for which the user entered the argument values has been served. When a token affects a CSV-type or regular-type argument from the current (PREFIX) command and it also affects some regular-type arguments from other POSTFIX or NOFIX commands in the interface, the values of the affected arguments from the POSTFIX or NOFIX commands should be reset to UNDEF.

If the command selection is CLOSE_ENDED, the event handler commits suicide, and the command is disabled. The event handler, before committing suicide, resets the status of the command status to OFF, which indicates to the HOUSE_KEEPER that the command has been disabled. However, if the command selection is OPEN_ENDED, the event handler resets its default-type arguments to their default values by generating tokens for the presentation component, and resets all regular-type arguments to UNDEF.

Reseting regular-type arguments to UNDEF insures that the command will not be re-executed until the user provides fresh values for the arguments. The command is kept active in this case, which allows the user to repeatedly execute the same command without having to reselect it.

```
1. Generate code for INIT event
   for all default-type arguments
      if argument.status is UNDEF
         send token to presentation component to initialize
         interaction techniques.
   for all regular-type arguments
      if argument.status is UNDEF
         send error token to presentation component.
         deactivate self.
         return.
   if all argument values are available
      produce a list of argument values.
      send token to application.
      for all regular-type arguments
         set argument.status to UNDEF.
   1a. Reset status of other command arguments
   for all CSV-type and regular-type arguments
      produce a set S of tokens the arguments are tied to.
   for all POSTFIX and NOFIX commands
      for all regular-type arguments
         if argument is tied to a token from the set S
            set argument.status to UNDEF.
   deactivate self.
2. Generate code for IN_CHECK event
   if all argument values are available
      produce a list of argument values.
      send token to application.
      for all regular-type arguments
         set argument.status to UNDEF.
      2a. Reset status of other command arguments
      for all CSV-type and regular-type arguments
         produce a set S of tokens the arguments are tied to.
      for all POSTFIX and NOFIX commands
         for all regular-type arguments
            if argument is tied to a token from the set S
               set argument.status to UNDEF.
      deactivate self
```

Figure 4.5 Producing Code for POSTFIX commands

In the case of POSTFIX type commands, the algorithm shown in figure 4.5 is used. For a POSTFIX command to parse correctly, the user must provide all argument values before selecting the command. An exception occurs in the case of default-type arguments. If the value for a default-type argument is not provided, the event handler sends an INITIAL token to the presentation component. The presentation component on receiving the INITIAL token initializes the interaction technique and generates an input token for the dialogue control component. This token is sent to the HOUSE_KEEPER which after updating the argument values generates a CHECK token. On receiving the CHECK token the event handler informs the application to execute the command. It then resets the status of all the regular-type arguments to UNDEF. The regular-type arguments from all POSTFIX and NOFIX commands which are tied to the same token as one of the CSV-type or regular-type arguments from the current command are also set to UNDEF. After this the event handler commits suicide. However, if a regular type parameter does not have a value when the event handler is instantiated, an error message is sent to the presentation component, and the event handler commits suicide. The OPEN_ENDED selection mode does not make sense in case of POSTFIX type commands.

The algorithm shown in figure 4.6 is used for producing event handlers for NOFIX type commands. In NOFIX commands, the selection of the command and its arguments is unordered. The user can follow PREFIX, POSTFIX, or any other sequence for selecting the command and entering its arguments. When the user selects the command, its event handler is instantiated, and an INIT event is sent to it. On receiving the INIT event, the event handler initializes all default-type arguments which do not have values. After doing so it checks whether all argument values are available. If so, it creates a list of all argument values and informs the application to execute the command. It then resets the status of all the regular-type arguments to UNDEF, and sends tokens to the presentation component to initialize default-type arguments. The regular-type arguments from all POSTFIX and NOFIX commands which are tied to the same token as one of the CSV-type or regular-type arguments from the current command are now set to UNDEF. If the command selection is CLOSE_ENDED, the event-handler commits suicide.

```
1. Generate code for INIT event
  for all default-type arguments
    if argument.status is UNDEF
      produce tokens for the presentation component to initialize ITs.
  if all argument values are available
    produce a list of argument values.
    send token to application.
    for all regular-type arguments
      reset argument.status to UNDEF.
    for all default type arguments
      produce tokens for the presentation component to initialize ITs.
    2a. Reset status of other command arguments
    for all CSV-type and regular-type arguments
      produce a set S of tokens the arguments are tied to.
    for all POSTFIX and NOFIX commands
      for all regular-type arguments
        if argument is tied to a token from the set S
          set argument.status to UNDEF.
    if command selection is CLOSE_ENDED
      deactivate self.
2. Generate code for IN_CHECK event
  if all argument values are available
    produce a list of argument values.
    send token to application.
    for all regular-type arguments
      reset argument.status to UNDEF.
    for all default type arguments
      produce tokens for the presentation component
      to initialize interaction techniques.
    2a. Reset status of other command arguments
      for all CSV-type and regular-type arguments
        produce a set S of tokens the arguments are tied to.
      for all POSTFIX and NOFIX commands
        for all regular-type arguments
          if argument is tied to a token from the set S
            set argument.status to UNDEF.
    if command selection is CLOSE_ENDED
      deactivate self.
```

Figure 4.6 Producing Code for NOFIX commands

If all the argument values were not available, the event handler waits for the user to provide argument values. On receiving the IN_CHECK event from the HOUSE_KEEPER, the event handler checks the status of its command arguments. And, when all argument values become available, the application is

asked to execute the command, regular-type arguments are reset to UNDEF, and tokens are sent to the presentation component to initialize interaction techniques for default-type arguments. The regular-type arguments from all POSTFIX and NOFIX commands which are tied to the same token as one of the CSV-type or regular-type arguments from the current command are set to UNDEF. The event-handler then commits suicide if the selection type for the command is CLOSE_ENDED. Otherwise, it remains active and waits for further input.

4.5. Producing HELPER

Diction produces an event handler, called HELPER, which generates help messages for all the commands in the application. The help messages for a command explain its syntax and selection type, and its argument requirements. For each argument the messages explain its type, i.e. whether the argument is CSV, default, or regular type. The help facility can be turned OFF when the interactive session is started or it can be turned OFF or ON any time during the interaction. If the help facility is turned off when the interactive session is started, the run-time control system does not instantiate the HELPER.

When the user selects a command, the run-time control sends its corresponding event to the HELPER. In addition to the HOUSE_KEEPER, the HELPER is the only event handler which can process input tokens which are produced as a result of command selection. If the help facility is ON, the HELPER sends a number of tokens to the presentation component. These tokens contain the text strings which explain the command. If the help facility has been turned off by the user, token generation is suppressed.

4.6. Caveats

Diction is capable of producing dialogue control components in which the prefix, postfix, and nofix types of syntaxes can co-exist. But there are certain cases where caution must be exercised as the behavior of the dialogue control component may differ from what the user expects. The problem occurs if the system has to parse postfix or nofix type commands whose arguments are a subset of that of a prefix open-ended type command. For example, assume that the dialogue control component were to parse the

following two commands.

cmd1 {PREFIX OPEN_ENDED} (arg1 arg2)
cmd2 {POSTFIX CLOSE_ENDED} (arg1 arg2)

When the user selects cmd1 and provides all it argument values, the cmd1 command gets executed. Since the cmd1 command is open-ended, it remains active for further executions. Now, if the user wants to execute the cmd2 command, he should provide the argument values before selecting the command since cmd2 is a postfix type command. But when the user provides the arguments values the system executes cmd1 as it was already active and all its argument values were provided.

There are two ways of avoiding this problem. The first way is make the prefix type command as close-ended, so that the command gets deselected after each execution. And the second is to change the postfix or nofix type command to a prefix type command, or vice-versa.

4.7. An Example

Consider a system which supports three commands shown in figure 4.7. The Add_Object_pr command is a PREFIX CLOSE_ENDED type command, whereas the Add_Object_ps command is POSTFIX CLOSE_ENDED type. The Add_Object_nf command is a NOFIX OPEN_ENDED command.

Appendix A5 shows the event handlers produced by Diction to implement the desired dialogue control. When the user starts the interactive session the run-time control system instantiates HOUSE_KEEPER, and HELPER. On instantiation, each of the event handlers is sent an INIT event. The HOUSE_KEEPER on receiving the INIT token resets the status of all the commands and arguments in the system to UNDEF. It also generates two output tokens. The first token is sent to the presentation component and is used for initializing the interaction technique named SIZE to 0.5. After initializing itself, the interaction technique will generate an input token for the dialogue control component. The second token is sent to the application. The application on receiving this token can initialize its data structures and send some set up information to the presentation component. The HELPER event handler does not do anything on receiving the INIT event.

```
SIZE : [0.0 : 1.0] {INITIAL = 0.5}
WORK : pick2d

Add_Object_pr {PREFIX CLOSE_ENDED}
    (where : WORK;
    size : SIZE {CSV};
    colour1 : (RED GREEN BLUE) {DEFAULT = RED})

Add_Object_ps {POSTFIX CLOSE_ENDED}
    (where : WORK;
    size : SIZE {CSV};
    colour2 : (RED GREEN BLUE) {DEFAULT = RED})

Add_Object_nf {NOFIX OPEN_ENDED}
    (where : WORK;
    size : SIZE {CSV};
    colour3 : (RED GREEN BLUE) {DEFAULT = RED})
```

Figure 4.7 Command Description for the Example System

The SIZE interaction technique initializes itself to 0.5 and generates an input token named SIZE for the dialogue control component. On receiving the token the run-time control system determines the active event handlers which can process this token. At this time, the HOUSE_KEEPER is the only event handler capable of processing the token. Therefore the SIZE token, after conversion into an IN_SIZE event, is sent to the HOUSE_KEEPER. The HOUSE_KEEPER, on receiving the event, changes the status of the size argument for all the three commands to DEF and stores the event value in the value field of the size argument for the commands.

When the user selects the Add_Object_pr command an input token named Add_Object_pr is generated by the presentation component and sent to the dialogue control component. Since both the HELPER and HOUSE_KEEPER event handlers can process the token, the IN_Add_Object_pr event is sent to them. The HELPER on receiving the event, sends a number of output tokens to the presentation component. These tokens explain the syntax and argument requirements of the Add_Object_pr command. On receiving the IN_Add_Object_pr event, the HOUSE_KEEPER first kills any other active command event handlers.

At this time there are no active command event handlers. It then instantiates the event handler for the Add_Object_pr command.

When the event handler for the Add_Object_pr command is instantiated, an INIT event is sent to it by the run-time control system. On receiving the INIT event, it does two things. First it generates a token for the colour1 interaction technique to set its value to the default value which is RED. This token is sent to the presentation component. Secondly it resets the status of its where argument to UNDEF.

The colour1 interaction technique after initializing its value to RED, generates an input token named colour1 for the dialogue control component. On receiving this token, the run-time control system determines that amongst the active event handlers only the HOUSE_KEEPER can process this token. Therefore the colour1 token, after conversion into an IN_colour1 event is sent to the HOUSE_KEEPER, which updates the status and value of the colour1 argument and generates a CHECK token. This CHECK token, after conversion into an IN_CHECK event is sent to the Add_Object_pr event handler. The Add_Object_pr event handler determines that all arguments are not available yet, so it returns.

When the user interacts with the WORK interaction technique to provide the value for the where argument, the presentation component generates the input token named WORK. After conversion into the IN_WORK event, this token is sent to the HOUSE_KEEPER. The HOUSE_KEEPER updates the status and value of the where arguments for all the three commands and generates a CHECK token. Since all argument values are defined at this moment, the Add_Object_pr event handler sends an Add_Object_pr input token to the application. This input token identifies the command that the application should execute. After sending this token, the event handler resets the status of its where argument to UNDEF and reinitializes the colour1 interaction technique. The where arguments of Add_Object_ps and Add_Object_nf commands are also set to UNDEF, and the event handler commits suicide.

It should be noted that the user could have changed the values of SIZE and colour1 interaction techniques for the Add_Object_pr command. When the user changes the value of SIZE interaction technique, IN_SIZE event is sent to the HOUSE_KEEPER. The HOUSE_KEEPER updates the values of size

arguments of all three commands, and generates a CHECK token. This ensures that size argument of all three commands has the most current value. In the case of colour1 interaction technique, IN_colour1 event is sent to the HOUSE_KEEPER which updates the status and value of the colour1 argument of the Add_Object_pr command.

When the user selects the Add_Object_ps or Add_Object_nf command, the HOUSE_KEEPER instantiates the event handler for the selected command. Since the Add_Object_ps command is of POST-FIX type, the user must provide all argument values before selecting the command. In this case the user must provide the value for the where argument. When the user interacts with the WORK interaction technique, an IN_WORK event is sent to the HOUSE_KEEPER which updates the value of the where arguments for all of the commands. The HOUSE_KEEPER then generates a CHECK token. On receiving this value the Add_Object_ps event handler generates an Add_Object_ps input token for the application. It then resets the status of the where arguments of the Add_Object_ps and Add_Object_nf commands to UNDEF and commits suicide.

In the case of Add_Object_nf command, the selection of command and argument entry are unordered. When the user provides argument values before selecting the command the HOUSE_KEEPER maintains them. After the command event handler is instantiated it checks whether the command can be executed. If not, it waits till the user provides the required argument values.

4.8. The Input for Chisel

As explained in section 4.1, Diction's output consists of two parts: event handlers for dialogue control component, and input for Chisel which is used for designing graphical presentation components. From the command description provided to Diction, it produces a procedure in Franz-lisp, which drives Chisel to produce the presentation component for the application.

For each global argument and local argument, Diction produces a call to one of the procedures in Chisel. The parameters to this procedure include the name of the argument, type of the argument, and the

range or enumerations of the argument. For each command supported by the application, a different procedure call is produced. The name of the command is the parameter to this procedure. Additional discussion on what this file contains is presented in chapter 5.

4.9. Chapter Summary

In this chapter, the generation of the dialogue control component of user interfaces is discussed. The system Diction is used for producing the dialogue control components. The input to Diction is a high-level specification of commands supported by the application. And, its output contains event handlers which implement the dialogue control, and a procedure in Franz-lisp which is used by Chisel for designing presentation components.

The algorithms used by Diction to produce event handlers for parsing commands in PREFIX, POSTFIX, and NOFIX fashion are presented. In addition to producing one event handler per command, Diction produces two event handlers which are responsible for house-keeping, and generating help messages for commands.

Finally, how event handlers produced by Diction implement the desired dialogue control is explained by using an example.

How the graphical presentation components are produced is discussed in the next chapter.

Chapter 5

Designing the Presentation Component

The presentation component of a graphical user interface presents a graphical front end to the user. The user provides input by interacting with graphical interaction techniques, and the system presents output in terms of images displayed on the screen. As stated in chapter 1, we will concern ourselves with the design of the input part only; the design of the output part remains the responsibility of the interface designer.

The Chisel system is used for designing presentation components. Chisel selects interaction techniques, determines their attribute values, and places them on the screen of the display device. While doing this, it is capable of considering user's preferences, interface designer's guidelines, and characteristics of the display device. This chapter discusses the working and design of Chisel. A number of examples are used to clarify the discussion.

After Chisel has produced the input part of the presentation component, the interface designer adds the information regarding the output part to it. This is done by using the Chisel's companion system vu. Vu can also be used for refining the designs produced by Chisel, as explained in the next chapter.

5.1. Chisel

5.1.1. Design Principles

The chief design goal for the presentation component design system was that it should enable the interface designer to quickly produce a working prototype of the presentation component, and enable him to refine the prototype afterwards. To achieve this goal two major design decisions were made. The first design decision was to reduce to a minimum the amount of information needed to produce a prototype. Chisel's input may consist of the following three parts:

1. dialogue requirements,
2. description of the device on which the presentation component will be implemented, and
3. user's preferences.

Of these three inputs, the designer does not have prepare the first two, and the third is optional. As explained in section 4.7, the dialogue requirements part of the Chisel's input is prepared by Diction. The device description can be selected from a library of device descriptions. The last part of the input, namely the user's preferences, is optional and may be omitted from the input altogether. Therefore, if the designer chooses not to provide the user's preferences, he does not have to prepare any input for Chisel at all. This is explained in greater detail in section 5.1.2. The second design decision was to provide interactive facilities for refining the prototypes produced by Chisel. The vu system, described in the next chapter, is used for this purpose.

An important design issue was the interface designer's control versus the UIMS control. Chisel pays special attention to allowing the interface designer to control a number of features of the presentation components it designs. Often this control is achieved by providing optional inputs which override the defaults used by Chisel. By providing these inputs the designer can get Chisel to consider some of the user's preferences and also consider his guidelines while designing the presentation components. A number of things can be controlled by the interface designer; examples include the maximum number of items in a menu,

placement of interaction techniques on the screen, and the selection of particular interaction techniques. This is explained in greater detail in section 5.2.4.

The final design goal of Chisel was that it should be easy to use. Unlike a number of existing UIMSs, Chisel does not require detailed specifications from the interface designer. As explained above, the major part of the Chisel's input is either automatically generated or selected from libraries. Once the input is provided to Chisel, it starts designing presentation components without asking for additional information during the design. The fact that the interface designer does not have to deal with cryptic specifications makes the system extremely easy to use.

5.1.2. Chisel's Input

This section explains Chisel's inputs in detail.

5.1.2.1. Dialogue Requirements

The first part of the Chisel's input, called the dialogue requirements, is an ASCII file produced by Diction which depends on the application for which the user interface is being produced. Note that the designer is not concerned with the contents of this file. The general structure of the dialogue requirements file is shown in figure 5.1. It contains a function in Franz Lisp, named "start", which describes the commands and their arguments in the interface.

In figure 5.1, the "init" function initializes Chisel's data structures. For each command in the interface, this file contains a call to the Chisel function called "command". The only parameter to this function is the name of the command. A variable number of commands may exist in the interface.

For each global argument and local argument in the interface, a call to the Chisel function called "argument" is produced. The parameters to this function include the name of the argument, type of the argument, low_range and high_range if applicable, and enumerations, if applicable. In the case of a subrange type argument, the low_range and high_range parameters describe the range of values the argument can assume, whereas in the case of a window, pick2d, or pick3d type argument, they denote the

window drawing limits. In the case of an enumeration type argument, the values of low_range and high_range are set to "nil" and the parameter called "enumerations" is used to represent the list of enumerations. The function called "finish" denotes that all the input has been entered.

```
(defun start ()
     (init)
     (command 'name)
          .
          .
          .
     (argument 'name 'type low_range high_range '(enumerations . . .))
          .
          .
          .
     (finish)
)
```

Figure 5.1 General Structure of the Dialogue Requirements File

5.1.2.2. Device Description

The second part of the input describes the graphical device selected for implementing the presentation component. This description can be pulled from the library of device descriptions, and is provided to Chisel in an ASCII file called "device.l". Figure 5.2 shows the description for the AED-767 colour graphics terminal. The Franz Lisp function "define-device" describes a device to Chisel. This function contains calls to a number of Chisel functions which initialize device properties. The function called "screen" is used for conveying screen properties to Chisel. The parameters to this function are the x and y resolution of the screen, size of character in terms of pixels in the x and y directions, the number of pixels per inch on the screen, and whether the device is colour or monochrome.

The available input devices are also mentioned in this file. A number of colours for the display device can also be defined in this file. A colour is defined by giving it a name and describing the amounts of red, green, and blue components that make the colour.

```
(defun define-device ()
    (screen   768      ;x resolution
              565      ;y resolution
              8        ;size of a char in x
              12       ;size of a char in y
              64       ;pixels/inch
              'yes)    ;colour?

    (input-devices '(mouse))

    (colour 'red       255    0      0)
    (colour 'green     0      255    0)
    (colour 'blue      0      0      255)
    (colour 'white     255    255    255)
    (colour 'black     0      0      0)
    (colour 'grey      128    128    128)
    .
    .
    .
)
```

Figure 5.2 Device Description for AED-767

5.1.2.3. User's Preferences

The third part of the input, called the user's preferences, is optional and may be omitted from the input altogether. This part of the input is used to generate presentation components which are sensitive to the user's preferences. This input is provided through an ASCII file named "user.l" which contains a function in Franz Lisp called "user-preferences". An example user's preferences input is shown in figure 5.3. The function "command-menu-location" is used to force the placing of command menu according to the user's preference. In case this information is not provided the default command menu location is used which is along the right side of the display screen. The user's favorite background and drawing colours can be specified by using the "favorite-bg" and "favorite-dr" functions. The colour names used with these functions should be defined in the device description file. In the absence of these preferences default background and drawing colours are used. The function "assign-colour" is used to assign particular background

and drawing colours to interaction techniques for command arguments. The name of the argument, the background colour, and the drawing colour are the parameters to the assign-colour function. The "select-interaction-technique" function is used to force the selection of the named interaction technique for a particular command argument. The parameters to this function are the name of the command argument and the name of the interaction technique that must be used to enter the argument value. In the absence of this information Chisel decides which interaction will be used to enter the argument value.

```
(defun user-preferences()
    ;location of the command menu left/right/top/bottom
        (command-menu-location 'left)

    ;favorite background colour
        (favorite-bg 'grey)
    ;favorite drawing colour
        (favorite-dr 'white)

    ;use brown as bg and white as dr for interaction technique for Limb
        (assign-colour 'Limb 'brown 'white)
        .
        .
        .
    ;select vertical gpot for the argument named Size
        (select-interaction-technique 'Size 'v-gpot)
        .
        .
        .
)
```

Figure 5.3 An Example of User's Preferences

5.1.3. Chisel's Output

Chisel produces an ASCII file called "design_file" which is used as the input to vu. This file defines the interaction techniques that are used in the presentation component. Figures 5.4 shows an example of Chisel's output. It is important to note that the designer is not concerned with the contents of this file. The information stored in this file is used by vu to show the presentation component and allow the editing of the presentation component in a highly interactive, direct, and graphical manner.

Chisel divides the display screen into a number of windows, and defines each window's name, background colour, drawing colour, and location in terms of screen coordinates. If an interaction technique is associated with a window, the name of the interaction technique and its parameters are defined immediately after the window definition. At the application run time the interaction technique occupies the same space on the screen and inherits the drawing and background colours of the window it is associated with.

5.2. Designing Presentation Components

Once the designer has provided all the inputs, Chisel starts designing the presentation component. During the design Chisel does not ask for any additional information from the designer. There are three main steps involved in designing a presentation component: selecting interaction techniques used in the presentation component, determining attribute values for the selected interaction techniques, and placing interaction techniques on the display screen. Each of these steps is discussed in the following sub-sections.

5.2.1. Interaction Technique Selection

The system has three main concerns while deciding on interaction techniques. These concerns relate to the type and range (or enumerations) of the command argument, user's preferences, and the device requirements of interaction techniques. Each of these concerns limits the set of interaction techniques that can be used to enter the argument values. For example, an interaction technique which generates real numbers cannot be used for entering text.

```
WINDOW
    Name cmenu0-v
    Bg 128 128 128
    Dr 255 255 255
    Llx 152.0 Lly 38.0
    Urx 191.0 Ury 115.0
  MENU
    Menu_type V_STATIC
    Item
      Name add_Limb
    Item
      Name remove_Limb
    Item
      Name move_Limb
    Item
      Name exit
WINDOW
    Name Length
    Bg 128 128 128
    Dr 255 255 255
    Llx 84.0 Lly 0.0
    Urx 147.0 Ury 15.0
  I_TECH
    Itech_name int_hgpot
    PARA
      Ipara_name min_value
      Ipara_type DOUBLE
      Ipara_value 1.0
    PARA
      Ipara_name max_value
      Ipara_type DOUBLE
      Ipara_value 5.0
    PARA
      Ipara_name label
      Ipara_type CHAR
      Ipara_value Length
END
```

Figure 5.4 Example Output of Chisel

The interaction technique implementing the argument must be able to produce the type of data the argument represents. An interaction technique which cannot produce the required type of data cannot be used as it will cause syntax and/or semantic errors in the system. After making sure the data type of the

interaction technique is the same as the type of the argument, Chisel determines whether the interaction technique is good for handling the range (or enumerations) of the argument. In doing so, the system marks those interaction techniques as unsuitable which produce data in a different range than that of the parameter. For example, an interaction technique which is designed to produce integers from 0 to 100 cannot be used for entering integers ranging from -10 to 10. There can be interaction techniques which do not have any pre-defined ranges. These techniques can also be used to enter the argument values. The system prefers techniques with ranges which match the range of the argument. The implication of this strategy is that Chisel selects special purpose interaction techniques for special ranges (e.g. angle technique for entering values between 0 and 360).

For arguments which do not have ranges, Chisel uses other rules for selecting suitable interaction techniques. For example, for enumerated type parameters, it marks vertical and horizontal menus as suitable.

All the commands in the dialogue requirement are implemented as menu items. Chisel uses the number of commands, the device resolution, and the height of a menu item in the generation of menus. The system is capable of generating both horizontal and vertical menus. If all the commands cannot fit into one menu, the system creates multiple menus. In the case of multiple menus, the system can create both overlay and static menus. Chisel has its own defaults for the orientation (vertical) and type (static) of menus, but is capable of considering the designer's preferences for these options. The designer can also control the number of items in a menu or the space occupied by a menu item in the menu. This is explained in greater detail the following section.

The designer can force the selection of particular interaction techniques for command arguments. To do so the designer provides the user's preferences part of the input (see figure 5.3). The "select-interaction-technique" function is used for forcing the selection of interaction techniques for command arguments. The user's preferred interaction techniques are used instead of the ones selected by Chisel.

The last thing the system determines is whether the device requirements of the interaction techniques can be met. If the device requirements of a particular interaction technique cannot be met, it cannot be used.

Based on the above three criteria the system selects or rejects interaction techniques. It is not always possible for the system to select one interaction technique which is most suitable. Often it has two or more interaction techniques which can fulfill the role equally well (for example, vertical and horizontal graphical potentiometers) and the system has no grounds for selecting one or the other. In such cases, the system continues with the design until it has some reason for rejecting extra techniques.

5.2.2. Attribute Determination

After the system has made a preliminary selection of interaction techniques, the next step is to determine attribute values for them. The attributes for interaction techniques can be divided into two categories; one for the attributes which depend upon the dialogue requirements, and the other for the attributes which are interaction technique and device related.

In the first category we have attributes such as the range and enumerations of arguments. The system does not have to do much for this category of attributes, it simply copies the attribute values from the dialogue requirement into the interaction techniques. These attribute values are used to generate values in special ranges or from special sets of choices.

The second category of attributes consists of size, location, and colour of interaction techniques on the display screen. For size and location, there is a data dependency problem which must be solved by Chisel. For some interaction techniques sizes are predetermined, whereas for others the size may depend upon the technique's other attributes, device characteristics, and designer's guidelines. For example, the size of a menu depends upon the number of characters in menu items, number of menu items, and device dimensions etc. Unlike for some other interaction techniques, the size of a menu cannot be fixed in advance. It can only be calculated after all the necessary information is available. The prototype of an

interaction technique (explained in section 5.4) can specify the name of the function which computes its size. Chisel automatically invokes this function when it needs the size of the interaction technique. There is another possibility of interaction techniques for which sizes cannot be determined until after other interaction techniques have been placed on the screen. Interaction techniques such as windows for work areas should be allocated all the space that is available after placing other interaction techniques on the screen. The size calculation for such interaction techniques is postponed until all other interaction techniques have been placed and it is possible to compute free space on the screen.

An interaction technique can constrain its position on the display screen. For example, command menus constrain their position according to the user's preference. If an interaction technique does not specify any positional constraint, the system generates it. While generating positional constraints for interaction techniques the system considers the designer's guidelines, the type of the interaction technique, and the size of the interaction technique. For example, the designer can instruct the system to place command menus along the right edge of the screen and other interaction techniques along the bottom of the screen. While generating positional constraints the system also follows its own criterion for good screen layouts. It likes to place objects with bigger width than height along the horizontal edges of the screen and objects with bigger height than width along the vertical edges. This criterion is used to avoid generating highly fragmented screen layouts.

We will illustrate by an example how the designer's guidelines and Chisel's own preferences are combined to generate positional constraints. Assume that an object has a width of 5 units and height of 1 unit. The designer's guideline is to place objects along the top edge of the screen. The system looks at the dimensions of the object and figures that it should be placed along an horizontal edge (top or bottom). It then looks at the designer's guideline and generates a constraint which states that the object be placed along the top edge. In the case where the two guidelines conflict, the system generates constraints according to what it thinks is more appropriate and considers designer's guideline while actually placing objects on the screen (This is explained in greater detail in the following section).

While generating positional constraints the system has one last opportunity for deleting extra interaction techniques. If the designer's guideline states that all objects be placed along the top edge, then the system prefers to retain objects which can be placed along the horizontal edges over the objects which should be placed along the vertical edges. It does so in its effort to follow the designer's guideline as far as possible without violating its own criterion for good screen layouts. If after this stage the system is left with more than one interaction technique representing an argument, it simply selects one interaction technique arbitrarily and ignores others.

Chisel also determines the background and drawing colours for interaction techniques. In the absence of the user's preferences for colours, Chisel assigns default background and drawing colours to all interaction techniques. The default background and drawing colour values are designer definable. By providing the user's preferences part of the input, colours for individual interaction techniques can be set.

5.2.3. Placing Interaction Techniques

After Chisel has made the final selection of one interaction technique per argument, it starts placing them on the display screen. While doing so the system is concerned with the dimensions of the display screen and interaction technique sizes and their positional constraints. To handle positional constraints the system has the notions of corners, edges, halves, and quadrants. It starts by satisfying more specific positional constraints first (upper-right corner is more specific than any corner). If for some reason a constraint cannot be satisfied, the system relaxes it to a less specific one and tries to satisfy the new constraint. It keeps relaxing constraints till the interaction technique is placed on the screen. To achieve its aim of satisfying more specific constraints first, it assigns a measure of specificity to each interaction technique according to the interaction technique's positional constraint. After doing so it starts placing interaction techniques with higher measures first. Table 1 shows the measures of specificity for different positional constraints.

Table 5.1 Measures of Specificity

Constraint	Measure of Specificity
A particular Corner	100
Left or Right Corner	95
Top or Bottom Corner	95
Any Corner	90
A particular Edge	80
Vertical Edge	75
Horizontal Edge	75
Any Edge	70
A particular Quadrant	60
A particular Half	50
Top or Bottom Half	45
Left or right Half	45
Anywhere	0

For example, consider trying to place an interaction technique along the bottom edge of the screen and there is not enough free space to accommodate the interaction technique along that edge. The system changes the constraint to bottom half and tries to satisfy it. It keeps relaxing the constraint till it can find enough space on the screen to place the interaction technique.

As mentioned in section 5.2.2 there are some interaction techniques whose size is such that following the designer's guidelines could lead to cluttered screen layouts. Recall that for such interaction techniques the system ignores the designer's guidelines and generates positional constraints according to what it thinks more appropriate. While placing such interaction techniques on the screen, Chisel makes one final attempt at getting as close to the designer's guidelines as possible. Assume that the designer's guideline is to place objects along the bottom of the screen, and for a particular object Chisel generated a constraint to place it along a vertical edge. While trying to place this object along the vertical edge, the system starts looking for free space from the bottom of the screen rather than the top. This way it tries to place as many interaction techniques along the bottom edge as possible and produce an uncluttered screen.

There are some interaction techniques for which sizes could not be determined by the system in the earlier steps. After placing all interaction techniques with known sizes the system determines the free area that can be allocated to interaction techniques with unknown sizes. The free area is divided equally amongst all interaction techniques with undefined sizes.

5.2.4. Interface Designer's Control

All UIMSs are restricted in the type of interfaces they can generate [Tanner85]. The interface designer therefore has to select a UIMS which can generate the required type of interfaces. While working within this high level restriction, the UIMSs should allow the interface designer a considerable flexibility in designing interfaces. This issue becomes more important in high level UIMSs, such as ours, which make a large number of decisions about the interface being generated. The UIMS should allow the interface designer to control or influence a large number of decisions it makes. At the same time it should not ask excessive amounts of information from the designer. Clearly, there is trade-off between the amount of information required from the interface designer versus the amount of decision making by the UIMS. If the UIMS requires too much information from the interface designer, it becomes too low level. Whereas if the UIMS makes most of the decisions by itself, it does not allow enough designer control. Using defaults in the UIMS is a way of balancing this trade-off. In the absence of input from the interface designer the UIMS uses its own defaults for the decisions it makes. But when the designer's input is present, it overrides the UIMS's defaults.

Chisel pays special attention to allowing the interface designer to control a number of features of the presentation components it designs. In the case of menus the interface designer can specify the height of a menu item or the maximum number of items in a menu. The default type of menus (static or overlay) can also be changed by the interface designer. The default background and drawing colour values used by Chisel can be changed according the designer's preferences. The designer can also instruct Chisel about the location of interaction techniques on the screen. In addition to changing the defaults, the designer can influence the decisions made by Chisel by providing the user's preference part of the input discussed in

section 5.1.2.3.

5.3. An Example

This section explains the generation of the presentation component for the three dimensional skeleton editing system discussed in section 3.3. The input to Chisel for generating the presentation component shown in figure 3.8 consisted of the dialogue requirements produced from the command description shown in figure 3.6 and the device description for the AED-767 shown in figure 3.5. The user's preferences part of the input was not provided for generating the presentation component shown in figure 3.8.

In the presentation shown in figure 3.8 (in the vu environment) all the commands in the interface are placed in the command menu which is located in the top right corner of the display screen. The size of the command menu depends on the number of commands supported by the application and on the maximum width and height of a menu item. In producing the command menu shown in figure 3.8 Chisel uses the defaults for the type of menu, height of a menu item, colours, and the location of the command menu. Chisel initially marks vertical and horizontal menus as suitable, but later decides to retain the vertical menu as the menu needs to be placed along the right edge (default location) of the screen.

To select the interaction techniques for command arguments, Chisel considers the data type and the range or enumerations of the arguments. For the COFM argument Chisel marks the vertical and horizontal graphical potentiometers as suitable. Both types of potentiometers can be used to produce real numbers in the range from 0.0 to 1.0. The reason for selecting these interaction techniques was that Chisel could not find any other interaction technique which was specially designed to enter real number from 0.0 to 1.0. Later, while placing the interaction techniques along the bottom of the display screen, Chisel retains the horizontal potentiometer. The same reason applies to the selection of graphical potentiometer for LENGTH. The potentiometer for MASS produces real numbers from 1.0 to 20.0. The size for a graphical potentiometer is already known, so Chisel does not have to determine it.

For entering the BEND argument Chisel selects the angle input interaction technique. This interac-

tion technique is specially designed to enter angle values (integers) from 0 to 360. For the same reason Chisel selects angle input technique for the ROTATE argument also. The size for an angle input technique is known to Chisel.

For entering the TORQUE argument vertical and horizontal menus are marked as suitable. The horizontal menu is finally selected as Chisel decides to place the interaction technique along the bottom edge of the screen. The size of the TORQUE menu is based on the number of items in the menu and maximum size of items.

For the LIMB argument Chisel selects an interaction technique which implements three dimensional pick. An ordinary window is selected for the INFO argument. The sizes for the LIMB and INFO interaction techniques are not known in the beginning. After placing all interaction techniques with known sizes Chisel determines the free screen area and distributes the free area equally amongst the LIMB and INFO interaction techniques.

The placing of interaction techniques was guided by Chisel's own defaults. It chose to place the command menu along the right edge of the screen and place other interaction techniques along the bottom edge. An exception to this strategy occurs in the case of interaction techniques which have a bigger height than width which should be placed along the right edge of the screen. Following this strategy, Chisel first placed the (vertical) command menu along the right edge of the screen. It then placed the (horizontal) menu for TORQUE along the bottom edge of the screen. When Chisel tries to place the (horizontal) graphical potentiometers for MASS, LENGTH, and COFM along the bottom edge it discovers that the screen space along the bottom edge is already occupied by the TORQUE menu. So it places the potentiometers in the bottom half of the screen. The angle input techniques have a bigger height than width, so according to the placing strategy they should be place along the right vertical edge. But since the space along the right edge is taken by the command menu, Chisel places the angle input techniques in the right half. After placing the interaction techniques with known sizes, Chisel computes the biggest free rectangle on the screen and divides the rectangle equally amongest LIMB and INFO interaction techniques. As a result of its

placing strategy, Chisel produces the screen layout shown in figure 3.8.

5.3.1. Producing Other Variants

As explained in section 5.2.4, the interface designer can use Chisel to produce presentation components which are sensitive to user's preferences and to the designer's guidelines. To produce user sensitive presentation components the designer needs to provide input through the user's preferences file ("user.l") discussed in section 5.1.3 and the designer can provide his guidelines in the defaults file ("defaults.l"). A large number of variants of a presentation component can be produced by using Chisel, and discussing each variant will take a large amount of space. Therefore only a few example will be discussed in this section. For producing each variant we will start with the input discussed in section 3.3 and modify it.

Using Chisel the designer can force the selection of interaction techniques for commands arguments. To force the selection of a horizontal graphical potentiometer for the BEND argument, the designer enters the following command in the user.l file and uses Chisel to produce the new presentation component shown in figure 5.5.

```
(select-interaction-technique 'BEND 'h-igpot)
```

To get Chisel to place the command menu along the left edge of the screen the designer adds the following command in the user.l file and Chisel produces the presentation component shown in figure 5.6.

```
(command-menu-location 'left)
```

The designer can also modify other defaults used by Chisel. This is done by modifying the defaults file ("defaults.l") used by Chisel. To instruct Chisel to place at most nine items in a menu the designer adds the following command at the end of the defaults file.

```
(set-items-per-menu 9)
```

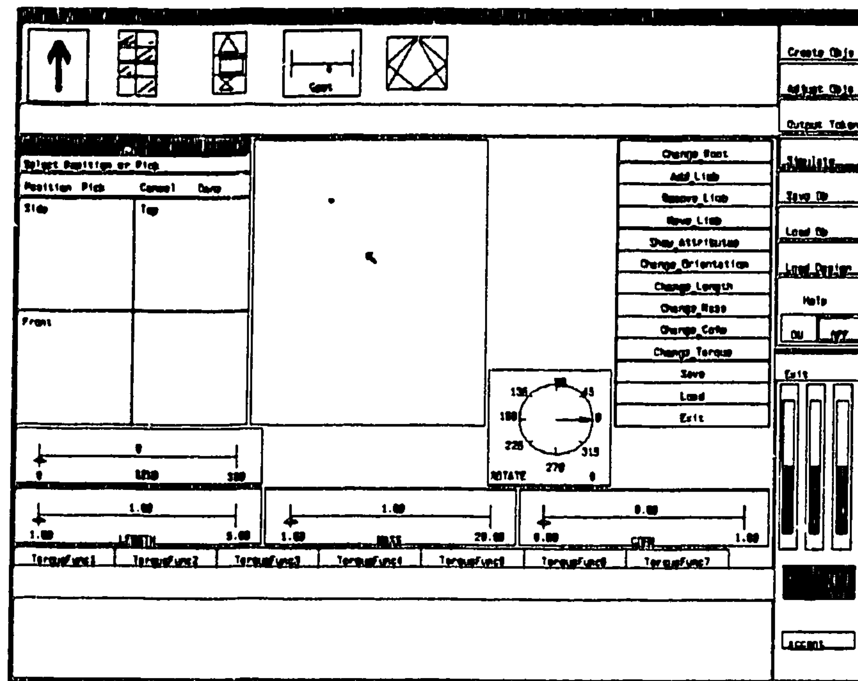


Figure 5.5 Forcing the Selection of Interaction Techniques

The new presentation component produced by Chisel is shown in figure 5.7. Notice that the command menu has been divided into multiple menus and the menus are placed side-by-side. While generating the presentation component shown in figure 5.7 Chisel assumed the default menu type which is static. To change the menu type to overlay, the designer adds the following command at the end of the defaults file.

(overlayed-menus)

The new layout produced by Chisel is shown in figure 5.8.

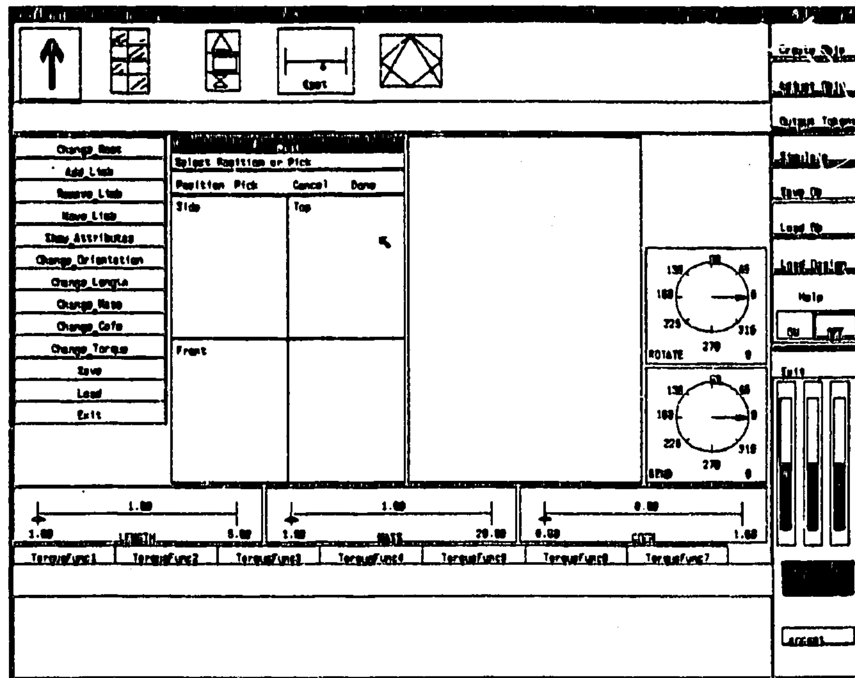


Figure 5.6 Command Menu Placed Along the Left Edge

5.4. Structure of Chisel

The system architecture and flow of information within the system is shown in figure 5.9. At the highest level the system can be logically divided into two parts. The first part stores information regarding the interaction techniques that the system uses for designing presentation components. This part is represented by the frame system in figure 5.9. The second part of the system uses the information stored in the frame system for designing presentation components. As shown in figure 5.9, the three main activities of this part include selecting interaction techniques, determining interaction technique attributes, and placing interaction techniques on the display screen.

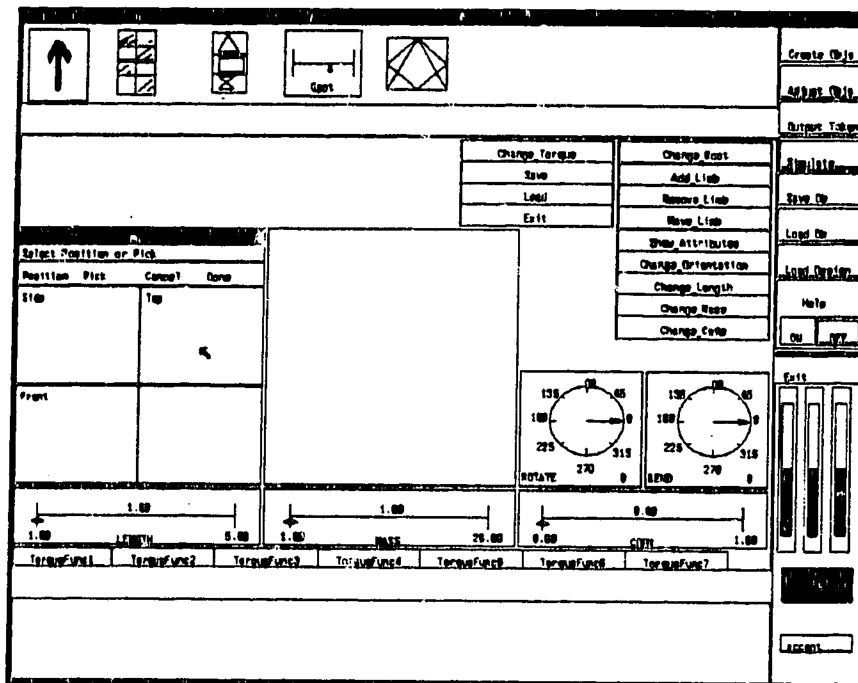


Figure 5.7 Menus with At Most Nine Items

Initially the frame system contains a description of the interaction techniques the system uses for designing presentation components. A prototype frame for an interaction technique has various slots containing its name, its range (if relevant), its size or the name of the function generating its size, constraints on its position if any, and input device requirement. The system uses this information to determine suitable interaction techniques for various dialogue requirements. While designing the presentation component, the system creates new instances of interaction techniques and destroys instances no longer required. An instance of an interaction technique contains actual (rather than default) slot (or attribute) values. An instance may contain some slots which do not appear in its prototype. Slots for location of objects on the display screen, for example, do not appear in prototype frames. Chisel generates positional constraints for

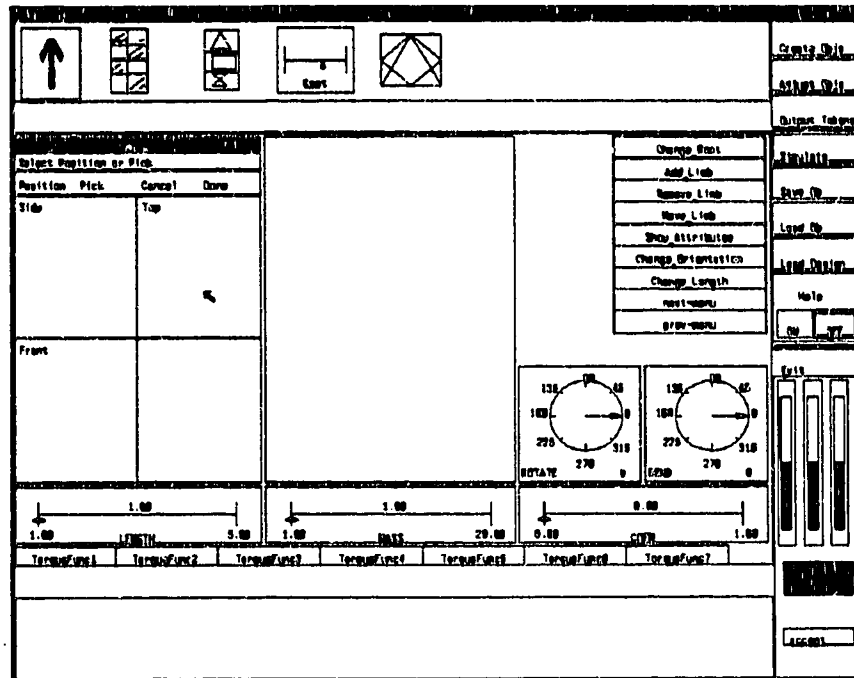


Figure 5.8 Overlayed Menus

placing interaction techniques on the display screen. These constraints are also stored as frames. When the system has completed the design it saves important information from the instance frames in the "design_file" discussed in section 5.1.3. This information is used for generating the presentation component.

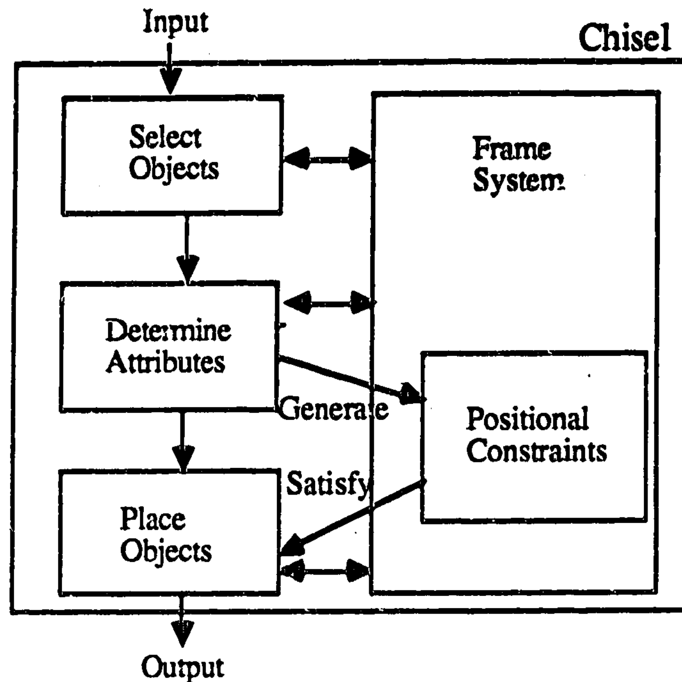


Figure 5.9 Structure of Chisel

5.5. Chapter Summary

The Chisel system is used for generating the input part of graphical presentation components. The inputs and outputs of Chisel are discussed. The strategies used by Chisel to generate the presentation components are discussed in detail. Chisel enables the interface designer to produce a number of variants of a presentation component without much additional effort. This is demonstrated by creating a number of variants of an example presentation component. Finally the structure of Chisel is discussed.

The presentation components generated by Chisel are refined by using a companion system called vu, discussed in the next chapter.

Chapter 6

Refining the Presentation Component

The system Chisel, discussed in chapter 5, is used for generating the input part of graphical presentation components. To complete a presentation component generated by Chisel the interface designer needs to add the information regarding the application output to it. After this information is added, the presentation component can be used to provide a front end to the interactive application. But the appearance and effectiveness of this presentation component can be greatly improved by refining it. These refinements include operations such as changing colours of interaction techniques, resizing interaction techniques, repositioning interaction techniques, and replacing interaction techniques selected by Chisel with other interaction techniques. The vu system is used to perform these refinements and add the information regarding the application output to the presentation components generated by Chisel. Vu provides a visual, highly interactive, and graphical facilities for refining the presentation components generated by Chisel.

This chapter discusses the key issues involved in refining graphical presentation components. How these issues influenced the design of vu is also discussed. A number of examples are used to demonstrate the ease-of-use of vu.

6.1. vu

The two main functions accomplished by using vu are refining the presentation components generated by Chisel and adding the information regarding the application output to the presentation components. The designs principles of vu along with the designer's interface of vu are presented in the following sub-sections.

6.1.1. Design Principles

One of the chief design goals of vu was that it should be easy to edit the graphical presentation components generated by Chisel. This goal was realized by providing facilities to perform the editing in a highly visual and direct manner. Vu provides a visual programming environment [Chang87, MacDonald82, Myers86a] in which the editing of graphical information is performed by using graphical techniques. There are a number of reasons why visual programming is specially suited for this purpose. The first reason is that it eliminates the need for the interface designer to write or modify programs written in conventional languages. This eliminates the necessity to deal with one dimensional (textual) abstractions for two or three dimensional graphical objects. Since visual programming is itself based on using graphical techniques for programming, the graphical objects to be edited can be displayed in their natural representation and manipulated in a natural manner. The second reason for using visual programming is that it eliminates the delay in feedback which is introduced by modify-compile-executes cycles in conventional programming languages. In vu the effect of each editing operation is immediately reflected on the screen, and an accurate and complete picture of the presentation component is always available to the designer. Visibility and direct manipulation of objects of interest coupled with little or no delay in feedback are the important features of vu that make it easy to use.

The second design issue was related to facilitating the editing by providing a large amount of high-level functionality. Although rarely used, one of the operations in editing the designs generated by Chisel is to replace the interaction techniques selected by Chisel with other interaction techniques. To facilitate this the vu environment provides prototypes of a number of interaction techniques. The designer can select interaction techniques from this set and include them in the presentation component being refined. By providing the prototypes of interaction techniques vu saves the interface designer a large amount of time and effort. The designer only needs to select prototypes of interaction techniques and customize them by changing their attributes. Another feature of vu which is of great importance in interactive design is its ability to support the rehearsal of the presentation component under construction. The designer can, at any

time, enter the rehearsal mode and interact with the presentation component just as the end user would. This allows the designer to "feel" the presentation component and modify the parts he is not satisfied with.

6.1.2. Vu's Interface

Vu provides a highly interactive and visual programming environment which enables interface designers to edit and create graphical presentation components in a very natural manner. The general strategy of vu is to enable the designer to draw the screen display that the end user will see, and to perform actions just as the end user would, such as, selecting menu items, selecting values from a potentiometers, or entering text.

Presentation components in vu are treated as a collection of objects (or interaction techniques). The designer can select objects of interest from a large collection of objects provided by vu. These objects can be customized and "glued" together to form a complete presentation component. The design of vu is influenced by our desire to make presentation component design as visual and as easy as possible. Vu capitalizes on the interactive graphical capabilities of modern devices to allow a very high bandwidth communication with the designer. It supports a visual language which allows programming with visual expressions. Interactive graphic communication and object-oriented design in vu yield a powerful source of exploration in user interface design. Vu pays special attention to minimizing extraneous activities that would otherwise compete for the designer's attention. It does so by removing unnecessary interruptions, such as leaving the environment to test the design and reloading it to make changes, and by supporting a self-documenting help system. How vu achieves these features and its interface are explained in the rest of this section.

When using vu the interface designer sees the screen layout (shown in figure 6.1) which consists of five subsystems: workshop, objects, simulation, help, and menu subsystems. Each subsystem accomplishes a well defined function and interacts with other subsystems through a well defined interface. Each subsystem, except the simulation subsystem, is allocated a separate window on the display screen. The simulation subsystem uses the window allocated to the workshop, as only one of these subsystems is active at any

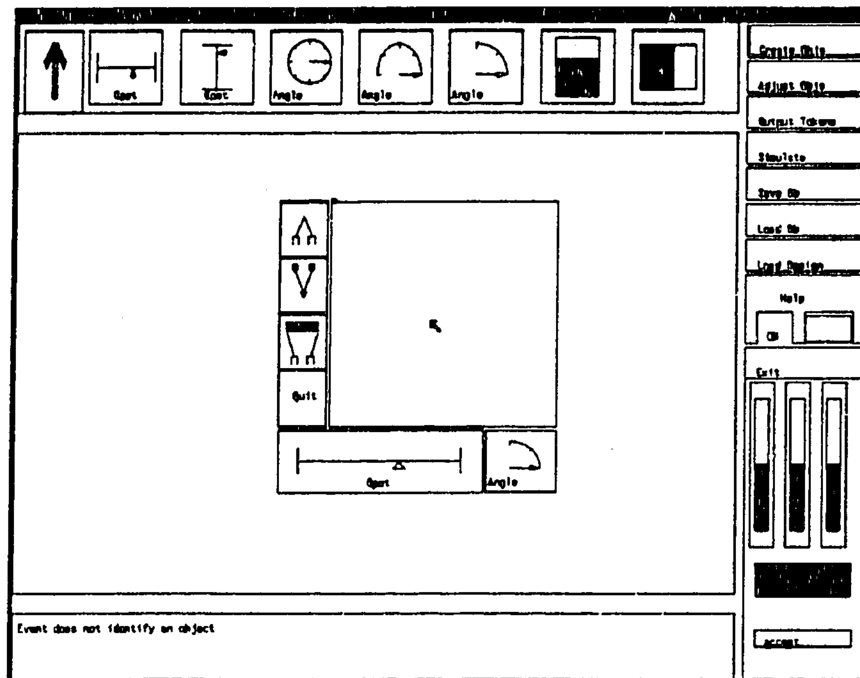


Figure 6.1 A Typical vu Screen

given time.

6.1.2.1. The Workshop Subsystem

The workshop subsystem handles the customization of objects in the presentation component being designed. It occupies the window, called the work window, located in the central part of the screen. The designs generated by Chisel are loaded in the workshop for editing. The objects in the presentation component can be resized and moved until they are the desired size and in the desired place. These operations are performed by using a mouse or tablet. To move an object in the work window, the designer points at a corner of the object and drags it to a new position. To change the size of the object the designer selects a

corner of the object. The system fixes the corner diagonally opposite the selected corner and redraws the object following the mouse. An object can be deleted by reducing it to size zero.

Workshop also enables the designer to customize other attributes of the objects. To do this the designer points at the desired object and the system displays the default (or current) values of the attributes of the object. For example, in the case of a window, its drawing limits and name are displayed. The window has a background colour, and its attributes are displayed in its drawing colour. To change the window limits or name of the window, the designer moves the cursor on top of the attribute and enters a new value. In situations like this, the designer uses the keyboard, all other operations are performed using the mouse or tablet. To change the background colour of the window, the designer clicks the first mouse button inside the window which accepts the colour displayed at the bottom right hand side of the screen. The designer can create colours of his choice by mixing different amounts of red, green, and blue through the use of three colour potentiometers. To change the drawing colour of the window, the designer clicks the second mouse button inside the window which accepts the colour. This facility lets the designer see the colours seen by the end user, and avoids the use of colour numbers or colourmap indices to name colours. The colours can be changed as often as desired until a suitable combination is achieved.

In the case of a menu, the system displays the number of items associated with the menu, which can be changed by typing a new value. The system divides the menu area equally for each item. The designer can associate icons or text with a menu item. To associate an icon, the designer selects an icon from the object window (discussed in the following sub-section) and places it in one of the menu items. To modify or to associate new text with a menu item, the designer selects a position inside the space reserved for the menu item and enters the text to be displayed. The text or icon associated with a menu item can be changed as often as desired. Attributes of other interaction techniques, such as the minimum and maximum values of a graphical potentiometer, can be similarly changed by entering new values.

The workshop subsystem also provides facilities for saving the design, loading a previously created design and modifying it, and creating more than one design without having to re-load the environment. The

designer can save a partially completed design in a database and work on it later. The system does not differentiate between starting a fresh design and starting from a partial design. This helps designers in dividing a complicated design into various sessions and being more thorough in addressing requirements of various parts of the presentation component.

6.1.2.2. The Objects Subsystem

The objects subsystem handles a hierarchy of graphical objects provided by the environment. It occupies the window, called the object window, located across the top of the display screen. There are four basic types of objects managed by this subsystem: windows, menus, interaction techniques, and icons. These objects define the top level of the object hierarchy. Icons corresponding to each type of object are displayed when the environment is first started. Each icon represents a collection of objects differing from each other in some respect, but similar in the general purpose served. When the designer points at an object, the objects subsystem reveals the next level of the hierarchy. The designer can select the objects at the lowest level of the hierarchy (the leaf nodes) and use them in the presentation component he is creating. Examples of leaf nodes include static menus, pull-down menus, pop-up menus, light buttons, radio buttons, various types of graphical potentiometers, interaction techniques for 2 and 3-dimensional picks, text windows, and graphic windows. Each object is represented by a unique name or icon. The results of the hierarchical object organization are that the designer never needs to scan too much information at a time, and because each object in the hierarchy has a distinctive appearance he never becomes confused. The leaf nodes in the hierarchy are distinguished from the internal nodes by the use of colour. Each leaf node contains the prototypical definition of the object it represents. The designer can start with this definition and modify it to suit his requirements. This results in large saving in time and effort. When the designer selects an object (at the leaf node), an instance of it is created in the workshop where it can be customized. Any number of instances of a particular object can be created and used.

6.1.2.3. The Simulation Subsystem

The third subsystem in the vu environment is the simulation subsystem which uses the work window. This subsystem enables the designer to rehearse the presentation component being created. The design created in the workshop is passed to the simulation subsystem which creates windows and activates menus and other interaction techniques defined in the presentation component. The designer can interact with the presentation component just as the end user would by preforming actions such as selecting values from potentiometers, selecting menu items, and entering text. This provides feedback to the designer on how the presentation component will behave when it is completely implemented. The designer can leave the rehearsal and go back to the workshop to make changes in the design and simulate the new design to test the changes. This feature of vu, though extremely useful in interactive design, is not supported by most systems (except Peridot [Myers86b]) which attempt to support visual programming.

6.1.2.4. The Help Subsystem

The help facility in vu is designed to reduce the need to memorize what each command does and what function is served by each object in the vu environment. The information pr[illegible] by vu is always up-to-date and reflects any object customization by the designer. Each object in the objects subsystem has a small description associated with it. This description includes its typical use in the user interface and explains its modifiable attributes. When an instance of this object is created in the workshop this description is copied in to the instance. When the designer modifies any of the instance's attributes this description is automatically updated by the system to reflect the change in attribute values.

To obtain information about the user interface under development, the designer selects the help command and points at the work window. The system provides the names and other important attributes of all the objects used in the user interface. The self-documenting capability in vu is achieved by generating descriptions by referring to the same data structures which control the functioning of the system. In this way vu avoids a mismatch between an object and its description.

6.1.2.5. The Menu Subsystem

The menu subsystem in the vu environment manages the display and selection of commands supported by the system. The menu is displayed in the window located in the top right corner of the screen. The command selected by the designer is highlighted to show the selection.

6.2. Defining Output Tokens

As explained in Chapter 4, in the user interfaces developed using our UIMS, output tokens are used for passing application output to the presentation component. The designer can associate any number of output tokens with a window by selecting the "Output Tokens" command from the vu command menu. Each output token is displayed by a different display procedure. While associating an output token with a window, the designer specifies the token name and the name of the display procedure for it. Different output tokens associated with a window are used for displaying different pictures in the same window. The responsibility of identifying the output token and calling the appropriate display procedure rests with the run-time support environment of the UIMS, whereas the responsibility of interpreting the output token correctly and displaying it rests with the display procedure.

Each object created by the designer is assigned a unique output token name, usually the name of the object. By sending a request to display a particular output token, the other parts of the system can control what is displayed by the presentation component. This mechanism makes the organization very flexible and enables the application to control the display at run time.

6.3. vu in Action

The best way to show the use of vu is to work through an example. This section presents the sequence of operations that a designer might perform to refine, and to add the application output information to the three-dimensional skeleton editing system discussed in section 3.3.

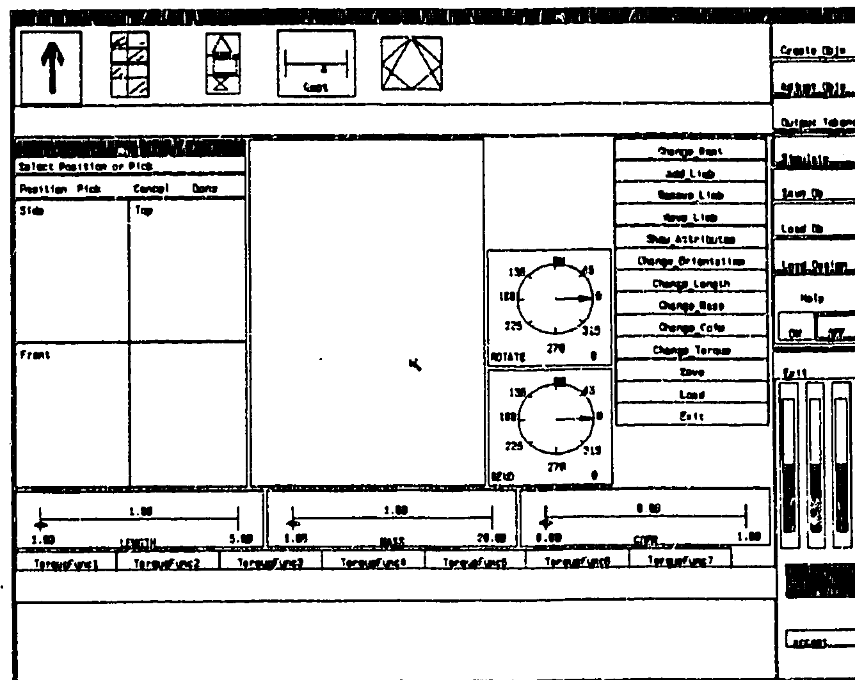


Figure 6.2 Vu Screen with Default Presentation Component

By selecting the "Load Design" command from the vu command menu the presentation component generated by Chisel is loaded for editing. The vu screen at this time is shown in figure 6.2. First the designer associates output tokens with INFO and LIMB windows (see table 6.1). To associate output tokens the designer selects the "Output Tokens" command from the vu command menu and points at the window with which tokens are to be associated. The system shows two smaller text windows within the

selected window. The top text window shows the default output token name generated by vu and the bottom text window is used for entering the display procedure name for the token name displayed in top text window (figure 6.3 shows the default output token in the INFO window).

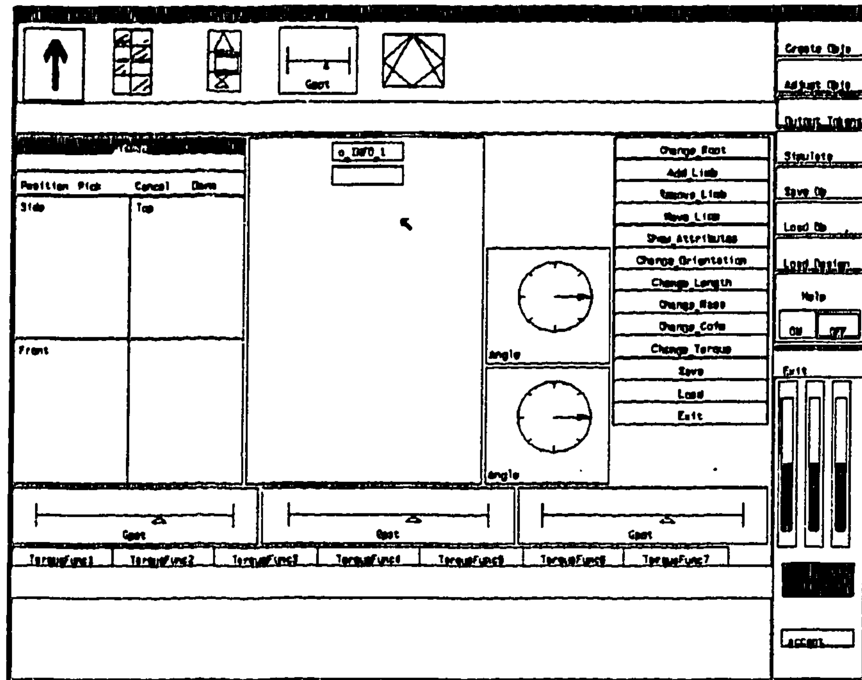


Figure 6.3 Defining Output Tokens

The designer can change the default output token name by typing a new name on top of this name (figure 6.4 shows the "INF" output token in the INFO window). The designer can add or delete tokens when in this mode. Similarly output tokens can be associated with the LIMB window.

Now the designer can resize the bottom menu (named "TORQUE") so that it fills the blank space towards its left. This is done selecting the top left corner of the menu and dragging it to its new location. Vu redistributes the size of the menu amongest its items and displays the new menu (figure 6.5).

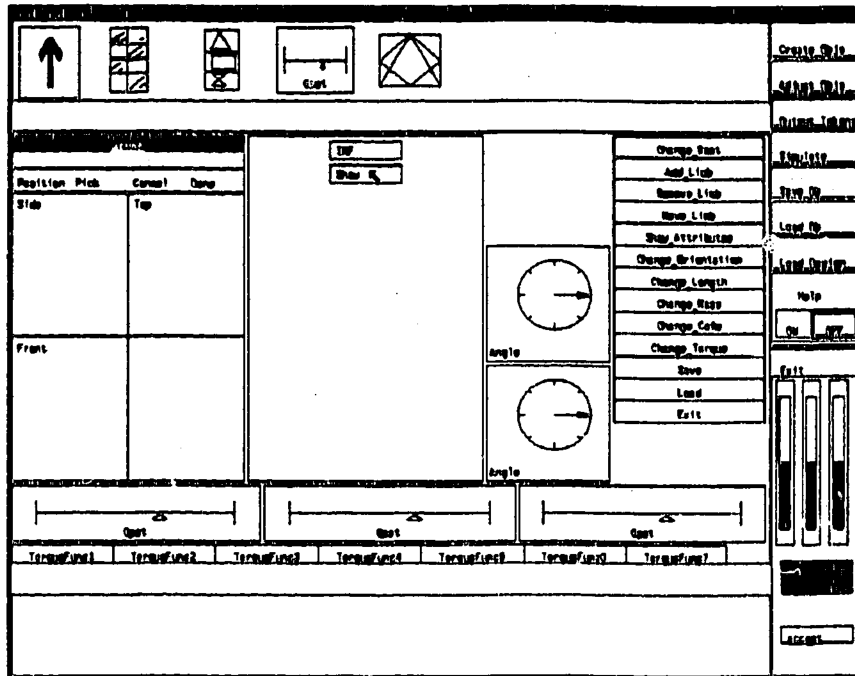


Figure 6.4 Changing Default Token Names

The designer now changes the size of the INFO window and repositions it so that it can fit on top of the angle input techniques (figure 6.6). These operations are performed by using the mouse.

In the next step the designer resizes the LIMB window so that it takes up the space freed by the INFO window (figure 6.7). The designer can now save the design by selecting the "Save db" command from the vu command menu and exit. This completes the editing and produces the presentation component shown in figure 3.10.

This example has so far not demonstrated all the editing that the could be performed by using vu. One of the editing operations is replacing objects selected by Chisel with other objects. To replace, for example, the angle input interaction technique for BEND the designer first deletes it by selecting the

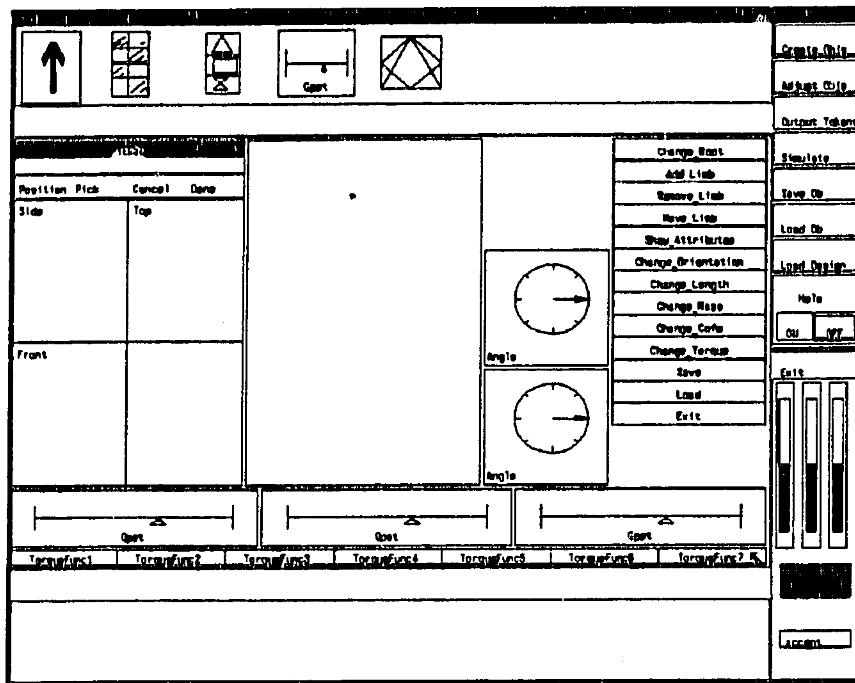


Figure 6.5 Resizing Interaction Techniques

"Create Object" command and reducing the interaction technique to a very small size. In the next step the designer explores the object hierarchy provided by the objects subsystem and locates the desired interaction technique (graphical potentiometer in this case). An instance of this interaction technique can be created in the workshop by pointing at its icon in the objects window. This instance of the interaction technique can be placed and resized by using the techniques described above (figure 6.8).

The name of this interaction technique should be changed to "BEND" so that the right input token is generated when the user interacts with it. This is done pressing the third mouse button inside the interaction technique window so that its name and window limits are displayed. The window name can be changed by typing new name on top of the default name (figure 6.9).

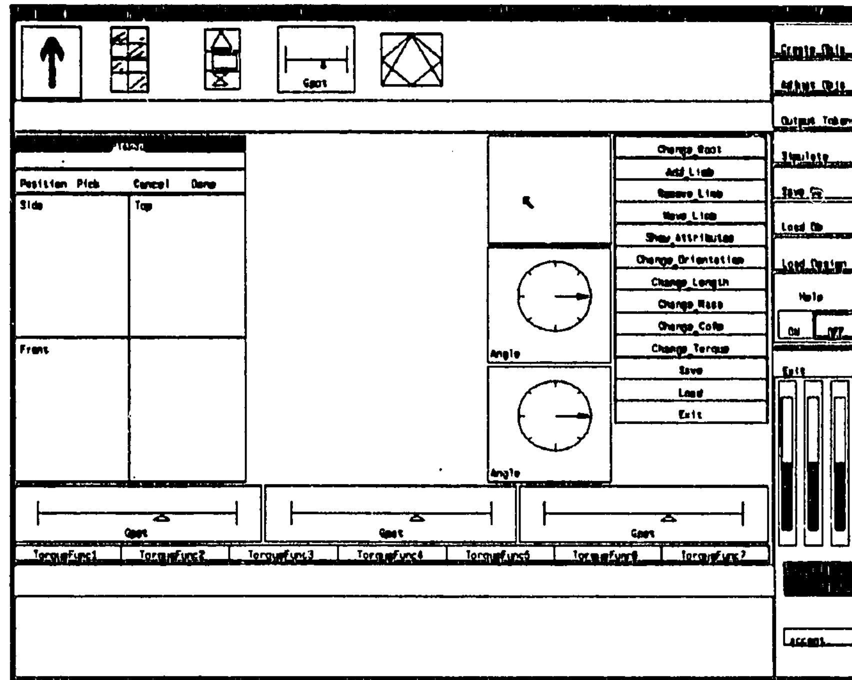


Figure 6.6 Resizing and Repositioning Interaction Techniques

To change the drawing colour of the interaction technique the designer presses the second mouse button inside the interaction technique window and creates the desired colour by using the colour potentiometers and accepts the colour. Vu redisplays the window attributes in the new drawing colour. To change the background colour the designer presses the first mouse button inside the interaction technique window and creates the desired colour and accepts it. This causes the window to be redisplayed with the new background.

To adjust other attributes of the interaction technique the designer selects the "Object Attributes" command from the vu command menu and points at the interaction technique. Vu displays the current (default) values of the interaction technique's attributes (figure 6.10) which can be changed by entering new

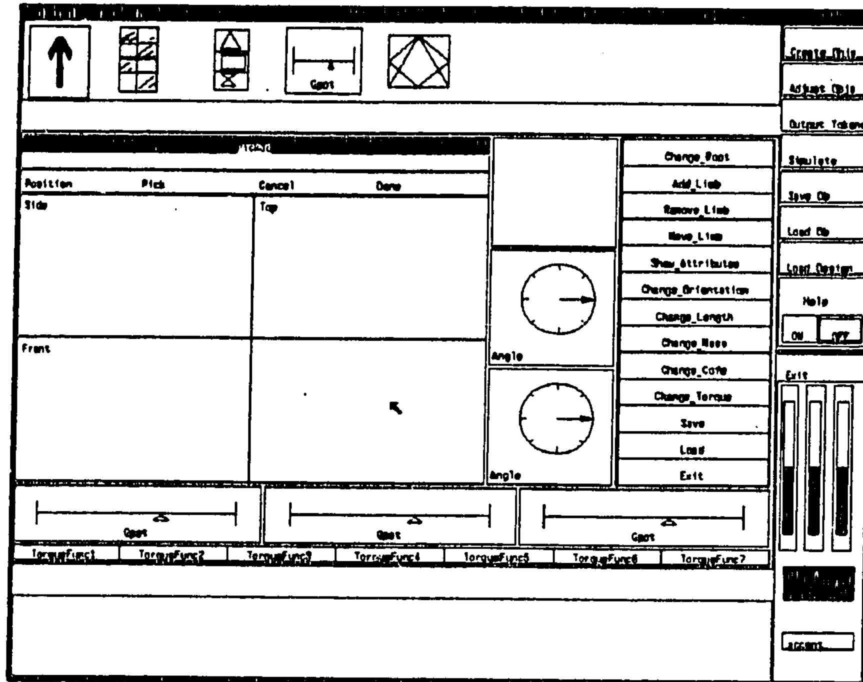


Figure 6.7 The Refined Presentation Component

values (figure 6.11).

While textual description and pictures describing the designer's actions take a fair amount of space, it takes only a few minutes to refine presentation components using this system.

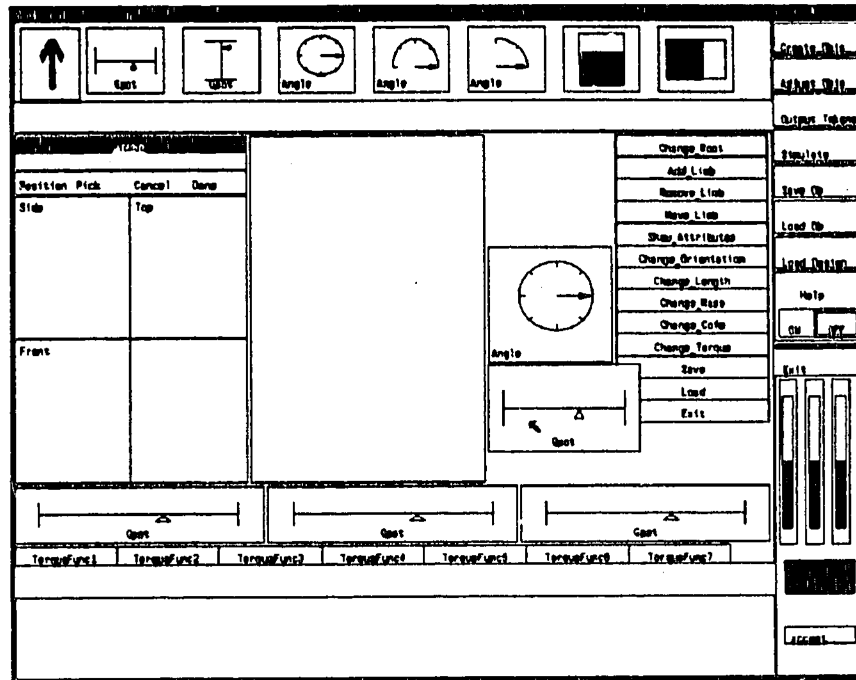


Figure 6.8 Creating New Interaction Techniques

6.4. Output of vu

The output of vu consists of a database which describes the presentation component, two token definition tables - one for input tokens and the other for output tokens, and a file of generated C code. The database is used by the run-time system to generate the presentation component when the interactive application is started. The input and output token tables are used when compiling the event handlers produced by Diction to C programs. The C code generated by vu is used to load the display procedures for the output tokens.

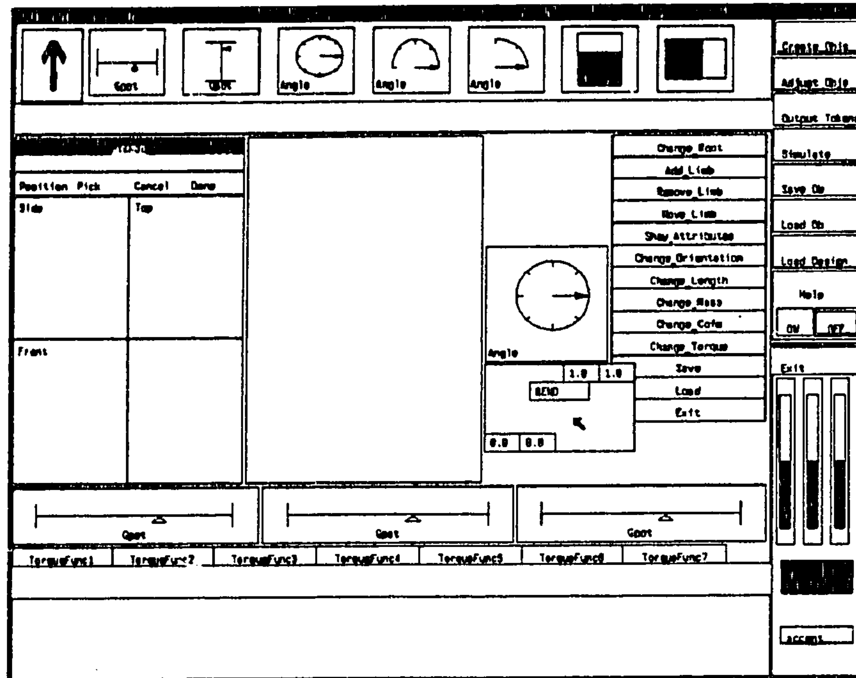


Figure 6.9 Changing Name of Interaction Techniques

6.5. Implementation and Structure of vu

Vu is implemented on a VAX‡ 11/780 running UNIX 4.3 BSD. It uses an AED-767 colour graphics terminal with a tablet. Vu can also be run on Jupitor-7 colour graphics terminal, Sun microsystems workstation, and VT125 terminal. The programs implementing vu are written in the "C" programming language [Kernighan78]. Vu uses a window based graphics subroutine package called WINDLIB [Green84a] and a graphical database package called FDB (Frame Data Base) [Green83].

The logical organization and the interactions between various subsystems in vu are shown in figure

‡ VAX is a trade mark of Digital Equipment Corporation.

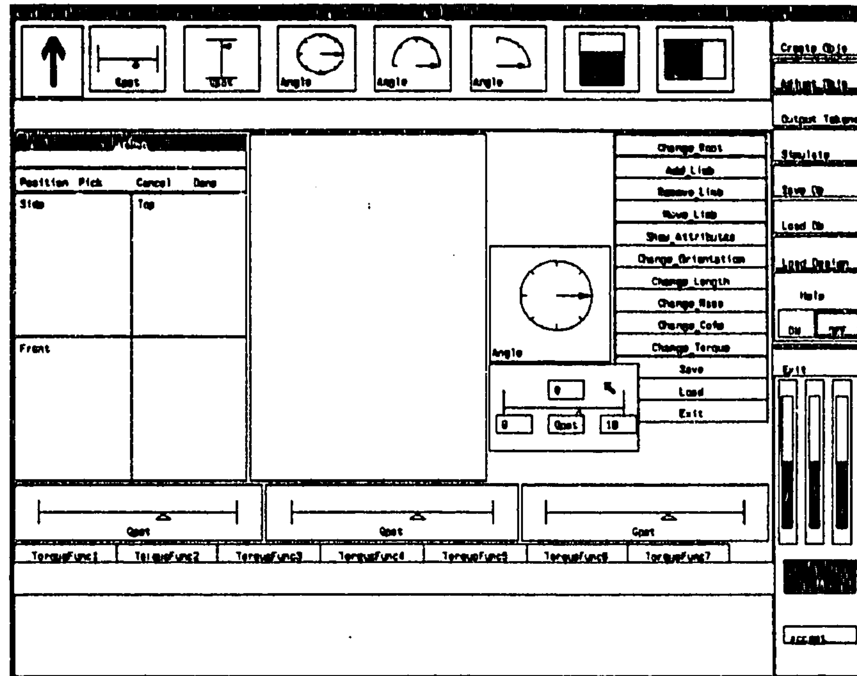


Figure 6.10 Default Attribute Values of Interaction Techniques

6.12. The objects subsystem loads the interaction technique descriptions from the database of interaction techniques. This database describes the modifiable attributes of each interaction technique. The database describing the user interface being developed is produced by the menu subsystem. The menu subsystem can load this database for further modifications or extensions.

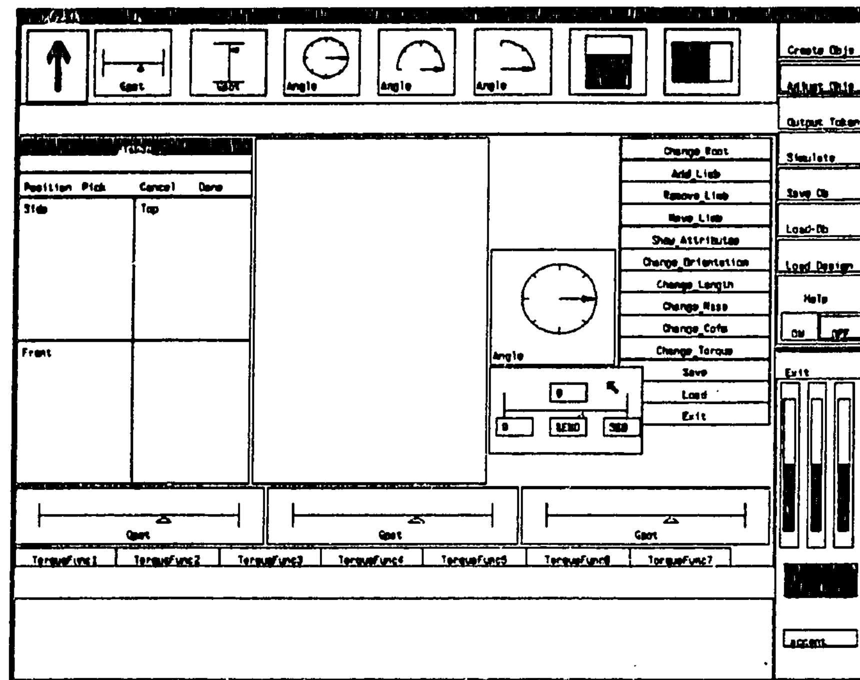


Figure 6.11 Changing Attribute Values of Interaction Techniques

6.6. Comparison with Existing Systems

In this section we compare vu to systems which tend to be visual in some way, we will not compare vu with systems which require a program-like specification of graphical user interfaces. In our opinion, the textual description of something which is graphic and dynamic is usually clumsy and unnatural.

The U of T UIMS [Buxton83], developed at the University of Toronto, allows the designer to implement menu based user interfaces in an interactive and graphical manner. The front end of this UIMS, called MENULAY, enables the designer to define user interfaces which are made up of networks of menus. The specification made by using MENULAY is converted into the C programming language and compiled and linked with the application-specific routines. Like vu, MENULAY allows the designer to enter geometrical

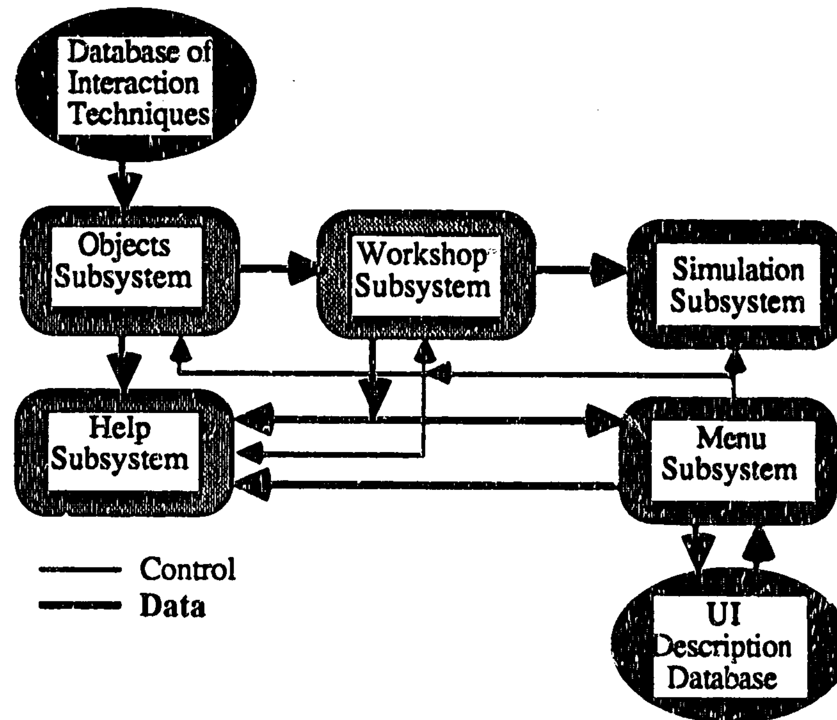


Figure 6.12 Logical Organization of vu

information graphically. But MENULAY does not allow the designer to create more than one window. This is one of the major drawbacks of MENULAY, which restricts the structured design of user interfaces. MENULAY also does not provide prototype objects which the designer can customize for his use.

Peridot [Myers86b] is a User Interface Management System which uses programming by example and visual programming to allow the user interface designer to create user interfaces. The major difference between vu and Peridot is that vu does not support programming by example. The visual programming component in vu is easier to use than in Peridot. The reason for this being that in vu, prototypes of all objects available to the designer already exist in the objects sub-system. The designer only needs to cus-

tomize them to suit his/her requirements. The equivalent of this does not exist in Peridot. Both vu and Peridot support facilities for interacting with the interface under development.

Trillium [Henderson86] is a system used for designing interfaces organized as collection of "functioning frames". A frame is a collection of items, whose attributes can be changed by the designer. The concept of frames used in Trillium is similar to the one used in COUSIN [Hayes83]. The objects in vu are similar to the items in Trillium, but vu does not use frames as a means for organizing interfaces. The items or frames in Trillium are incapable of self-description whereas the vu system can, at all times, provide an up-to-date description of the interface and the objects used in the interface.

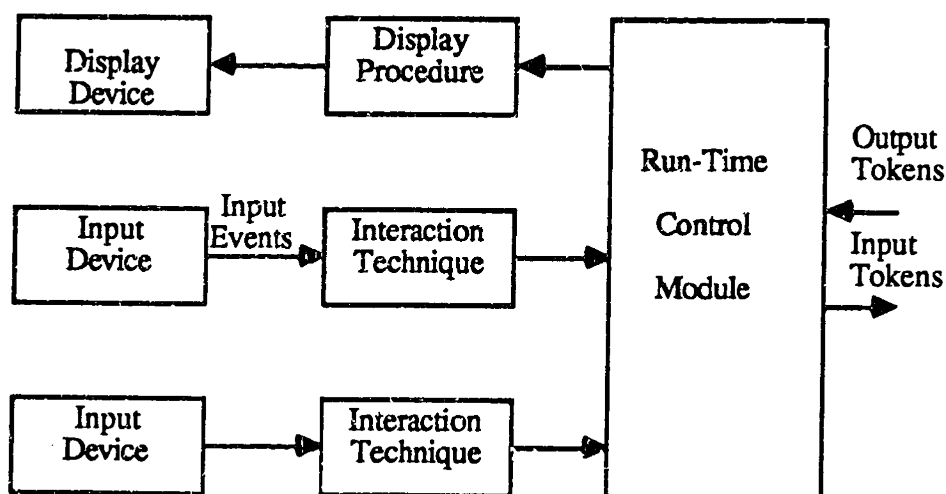


Figure 6.13 Flow of Tokens

The UIMS developed by Luca Cardelli [Cardelli87, Cardelli88] uses direct manipulation techniques for building graphical user interfaces. There are many similarities between Cardelli's system and vu. Like vu, Cardelli's system also provides a collection of objects (or interactors) which are customized by using direct manipulation techniques and glued together to form a complete interface. The approach followed for

adjusting the object attributes (property sheets) is similar to the one followed in ipcs [Singh86]. Vu exploits the graphical appearance of objects to display and customize object attributes, as shown in figure 6.12. The interactors in Cardelli's system are incapable of self-description.

6.7. Run-Time Support

The presentation component generated by vu communicates with other parts of the system by the use of input and output tokens. The use of output tokens is explained earlier in this chapter. The presentation component receives user input in the form of events and converts them into input tokens. An input token is used for identifying things such as commands, values, and locations etc. Vu automatically generates the names of input tokens depending upon the name and type of the object. These tokens are sent to dialogue control component for further processing. Figure 6.13 shows the flow of tokens within the presentation component.

6.8. Chapter Summary

This chapter discusses the vu system which is used for refining the presentation components generated by Chisel. Vu provides a highly interactive, graphical, and visual programming environment for editing graphical information. How designers use vu to refine presentation components is explained by working through a detailed example. The interface of vu is compared with other systems which also use visual programming techniques to create graphical user interfaces.

Chapter 7

Run-Time Support

The run-time support environment is responsible for managing the communication amongst the components of the interactive system and for handling the creation and destruction of event handlers for the dialogue control component. The run-time support environment provides a number of routines and manages a number of data structures. This chapter discusses how the communication between the components of the interactive system is handled and explains the creation and destruction of event handlers.

7.1. Managing Communication

The run-time support environment treats each component in the interactive system as a single logical unit. These components communicate with each other by sending and receiving tokens. The direction and the component that will process a token is known when the token is generated. All the tokens are passed through the run-time support environment, which distributes the tokens to various components and schedules their execution. The run-time structure of a typical interactive system is shown in figure 7.1. Although this structure allows any component to communicate with any other, usually only a few of the possible communication paths are used (see figure 7.2).

The presentation component consists of a number of interaction techniques which generate input tokens for the dialogue control component when the user interacts with them. The dialogue control component receives input tokens generated by the presentation component and processes them. While processing the input tokens the dialogue control component may generate input tokens for itself and for the application, and output tokens for the presentation component. The application receives input tokens generated by the dialogue control component and generates output tokens for the presentation component.

The presentation component receives output tokens generated by both the dialogue control com-

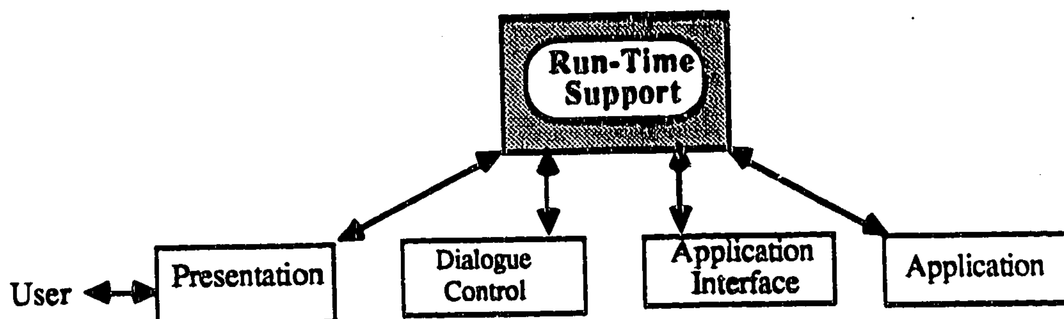


Figure 7.1 Run-Time Structure of a Typical Interactive System

ponent and the application. These tokens are used to adjust the attributes of the interaction techniques, display error messages, and display images.

When a token is generated it is inserted at the end of the token queue associated with the receiving component. An exception occurs in the case of input tokens for the dialogue control component generated from within the dialogue control component. These tokens, called the internal tokens, are placed in the front of the token queue for the dialogue control component. Figure 7.2 shows the token passing amongst the components of interactive system. In this figure the application interface model and the application are merged together and labeled as the application.

The run-time support component examines the token queue associated with each of the components and schedules tokens for execution. It removes tokens from the front of the selected queue one at a time, converts them into events, and schedules their execution. The output token queue associated with the presentation component has a higher priority than the other token queues. All the tokens in the output token queue for the presentation component are processed before the input token queues for the dialogue control and the application interface model are selected. Only one token from the input token queues for

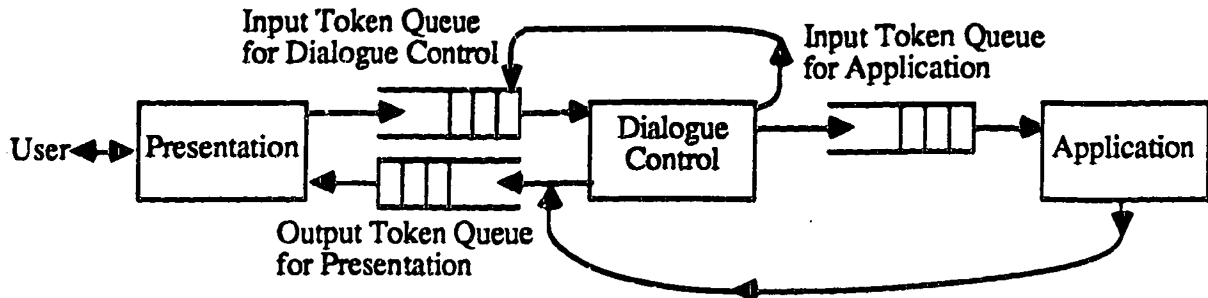


Figure 7.2 Token Passing Amongst Various Components

the dialogue control component and the application is scheduled for execution in one cycle of the scheduler.

Before the scheduler can process a token, the presentation component determines if any of its input devices has a value ready. If this is the case the value is converted into an event and processed by the presentation component. Often this means that one of the interaction techniques will generate an input token for the dialogue control component. The steps in the scheduling process are shown in figure 7.3

1. If an input device has input ready,
   call the presentation component to process it.
2. Process a token from the token queues.
   2a. Select all tokens from the output token queue
   for the presentation component.
   2b. Select a token from the input token queues.
   Alternate between the input token queues for the dialogue
   control component and the application.
3. Go to step 1.

Figure 7.3 Steps in the Scheduling Process

There are two notable effects of this scheduling process. First, by processing the input device data

before anything else it ensures that the feedback to the lexical operations is provided as quickly as possible. Second, by processing all the output tokens for the presentation component before other tokens, it ensures that the semantic feedback is provided to the user as soon as it is available.

7.2. Creating and Destroying Event Handlers

In addition to scheduling tokens for execution, the run-time support environment is responsible for creating and destroying instances of event handlers for the dialogue control component. When the interactive system is first started, the run-time support environment instantiates the HOUSE_KEEPER and the HELPER. The HOUSE_KEEPER then causes the creation and destruction of event handlers for commands depending on the user's actions. The HOUSE_KEEPER calls one of the routines provided by the run-time support environment (called "create_instance") which actually instantiates event handlers.

The creation of an event handler involves updating the run-time support environment's data structures and sending an INIT event to the newly instantiated event handler. It is important to note that the INIT event is sent to the event handler as soon as it is instantiated and not put in the scheduler queues. The new event handler, on receiving the INIT event, initializes its state and gets ready to process the user input.

The destruction of event handlers can be caused by the event handlers themselves or by the HOUSE_KEEPER by calling a routine (called "destroy_instance") provided by the run-time support environment. On receiving a request for destroying an event handler the run-time support environment updates its data structures and removes the entry for the event handler from its scheduling tables.

7.3. Chapter Summary

The components of an interactive system communicate with each other by sending and receiving tokens. All tokens are passed through the run-time support environment which distributes the tokens and schedules their execution. When a token is generated it is added at the end of the token queue associated with the receiving component. An exception occurs in the case of internal tokens, which are added at the front of the token queue of the receiving component. The run-time support environment removes tokens from the front of the queue, one at a time, converts them into events, and calls the appropriate procedures to execute them.

Chapter 8

Summary and Conclusions

This chapter summarizes the research presented in this thesis. The range of interfaces that the UIMS can create is described, the experience with developing the UIMS and with using it is presented, and the contributions of the research are reviewed. In addition, some features which could add to the usefulness of the UIMS, and directions for future research are presented.

8.1. Range of the UIMS

All UIMSs are designed with several restrictions in mind. One of the main restrictions is the type of interfaces generated by the UIMS. The UIMS presented in this thesis is only aimed at graphical user interfaces. It does not help with command language interfaces. It also does not help with the programming of the semantics of the application. Even within graphical interfaces, it cannot produce all kinds of interfaces. The classification of the three user interface control mechanisms presented in [Thomas83] can be used to get an idea about the range of interfaces that the UIMS can generate.

External Control

In this type of control structure, the user interface is in control and calls the application in response to user commands (figure 8.1). The application is modeled as a set of subroutines which can be invoked by the user interface at appropriate times during the interaction with the user. The UIMS presented in this thesis is capable of generating user interfaces which use this type of control mechanism. The three-dimensional skeleton editing system, described in section 3.3 in chapter 3, is an example of this type of control mechanism.

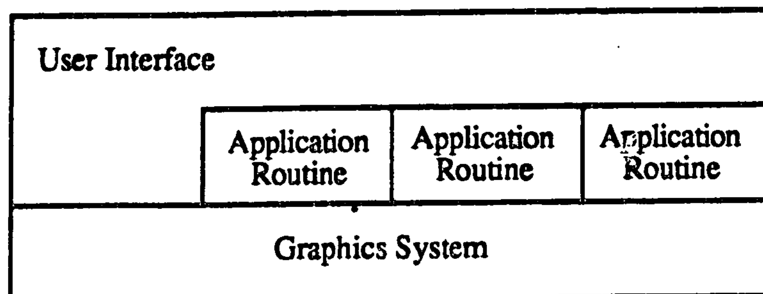


Figure 8.1 External Control (adapted from [Thomas83])

Internal Control

This is the opposite of external control. The application is in control and calls user interface routines when needed (figure 8.2). The user interface routines correspond to atomic input/output actions as far as the application is concerned. The UIMS presented in this thesis is not well suited for this type of control. The reason being that applications of this type bypass the dialogue control component, as only they know about the input required and which user interface routine should be invoked.

Mixed Initiative

The mixed control structure is similar to the external control structure except that the application routines, in addition to returning in the normal way, can call user interface routines to ask for additional information from the user (figure 8.3). The UIMS presented in this thesis is capable of handling this type of control. The application can ask the UIMS to invoke specific user interface routines, such as a dialogue

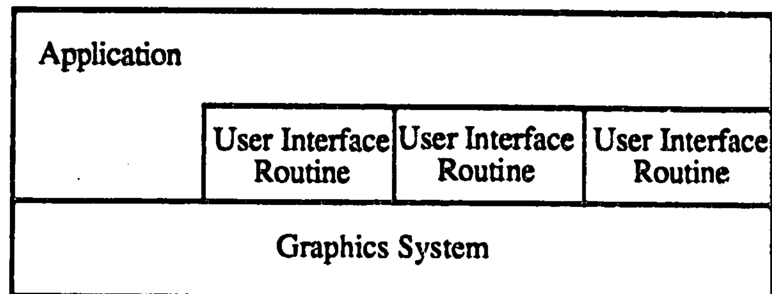


Figure 8.2 Internal Control (adapted from [Thomas83])

box, through which the application can communicate directly with the user.

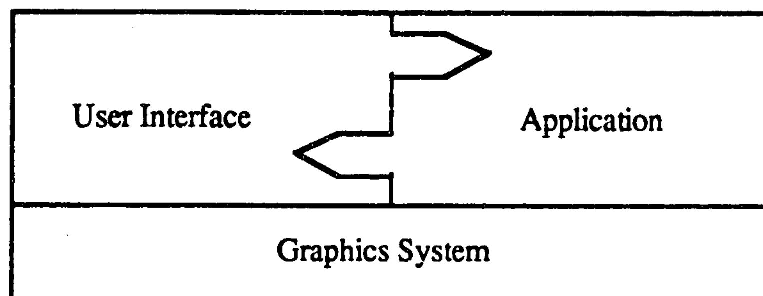


Figure 8.3 Mixed Initiative

In addition to external or mixed control, the interfaces generated by the UIMS are characterized by relatively static screens. The screen of a user interface does not change dramatically over a short period of time. The change in screen display is usually incremental. It is not possible to express interfaces for window managers which allow a variable number of windows, each running a different application. The reason

being that to send information to a window, the application needs to generate output tokens. To handle the output correctly, the binding between the window and the tokens it displays should be done at run-time, which our UIMS cannot handle.

8.2. Experience

This section discusses experience with developing the UIMS and with using it.

8.2.1. Experience with Developing the UIMS

The UIMS was developed in the UNIX 4.3 BSD environment on a VAX 11/780 machine. The primary device for testing graphics was an AED-767 colour graphics terminal. The complete UIMS, except Chisel, has also been ported over to Sun microsystems environment. The UIMS requires nearly 745K bytes of source code. The tools that make up the complete UIMS were developed using facilities which seemed most convenient to use.

Diction was developed using Lex [Lesk75] and Yacc [Johnson75]. The use of Lex and Yacc saved me a great deal of time and effort as I could quickly develop a parser for the input grammer. The code generation in Diction is handled by routines written in the C programming language.

Chisel was developed in Franz Lisp. The interpretive nature of Lisp made the development of Chisel much faster as no time was wasted waiting for compilation, and it was much easier to try out different possibilities quickly and see how they work. Developing Chisel in a compiled language, such as C, would have taken much longer and the source code would be much larger.

Vu was developed in the C programming language. It uses a window based graphics package called WINDLIB [Green84a] and a graphical database package called FDB [Green83]. The main reasons for choosing WINDLIB were the effort I had already invested in producing interaction techniques and familiarity with the package. The use of FDB saved me the time and effort which I would have otherwise spent in writing routines for creating and manipulating special file formats. The choice of WINDLIB and FDB imposed the selection of C as these packages can be called only from C programs.

Additional details about the implementation and structure of the UIMS are provided in sections (overview) 3.2, (chisel) 5.4, (vu) 6.5, 6.7, (run-time structure) 7.1, and 7.2.

Looking back at the choices made for the development tools, I feel that the choice of Lex and YACC for Diction and the choice of Lisp for Chisel were very appropriate. It would have been better to develop vu in an interpretive language which would have saved the time spent in compiling programs and allowed run-time loading of procedures. The run-time loading of procedures is very desirable for loading new icons when vu is running.

8.2.2. Experience with Using the UIMS

The UIMS has been used by a number of users for creating user interfaces for a variety of systems including a three-dimensional skeleton editor used by the animation research group, a distributed network editor used by the distributed systems research group, a stickman animation system, and a paint program. Table 8.1 shows a summary of the users' background and some of the interfaces they developed. The interface for the three-dimensional skeleton editor is described in section 3.3 of chapter 3, and the interface for the distributed network editor is described in Appendix A2. The stickman animation system enables the user to change the orientation of the stickman's limbs and store the orientations as frames. This sequence of frames can then be played back by the user at a desired frame rate. The paint program allows the user to place, move, and remove geometrical shapes of different types and colours on a canvas. The program was designed to handle triangles, rectangles, and diamonds in red, green, blue, and yellow colours. The fish animation system is used for creating an animation of fish in a fish bowl. The user can control the number of fish, their average velocity, the standard deviation in the velocity, the number of frames produced in an animation sequence, and the rate at which the animation is played back. This system has a number of similarities with particle systems [Reeves83].

Table 8.1 Summary of Users' background and Systems

User	Background, Specialization	Previous Experience with UIMSs	Application	Complexity
me	Ph.D. Candidate UIMSs, Graphics	Builder, User	3-d Skeleton Editor	High
<A>	Comp. Sci. Undergrad	Used one User Interface Toolkit	Distributed Network Editor	Moderate
<B>	Ph.D. Candidate Distributed Systems	None	Paint Program	Low
<C>	Ph.D. Candidate UIMSs	Builder, User	Stickman Animation	Low
<D>	M.Sc. Student Graphics	None	Fish Animation	Moderate
<E>	Ph.D. Provisional Theory	None	Stickman Animation	Low
<B>	Ph.D. Candidate Distributed Systems	None	Distributed Network Editor (modified)	High

Since there are no user manuals for the UIMS, it was necessary for me to guide the other users initially. They also had the complete skeleton editor for an example. The distributed network editor was developed by a fourth year computing science undergraduate student. His experience with the UIMS, as described by the user himself, is presented in Appendix A3. The feedback of the users was very encouraging.

Not surprisingly, the main positive aspect of the experience was that the UIMS cuts down tremendously on the time and effort required for developing user interfaces. All of the users were able to develop their interfaces quickly and easily. Most of the users were surprised at how fast they could develop the first version of the interface and refine it. In an informal experiment, it took me nearly two hours (wall clock time) for developing the interface for the skeleton editor using the UIMS. When developing a similar interface for the same application by using the UofA UIMS I spent nearly 56 hours (wall clock time). A speedup of nearly 2800% was achieved over a "conventional" UIMS. The difference in time and effort would have been even greater if a conventional programming language were used.

The second positive aspect of the experience was that all the users found the UIMS to be easy to

learn and easy to use. The users mentioned this as one of their main comments on the UIMS. One user described the experience as "fun". Since it did not take much time and effort to produce an interface, the users built and tested a number of variants of the interfaces. The ability to do so lead to substantially different and better interfaces than were originally planned for. This helped them achieve the "right" look and feel.

One of the aspects of the UIMS, which turned out be very useful when refining presentation components, was the ability to see the refinements without delay. This reduced the number of (unpleasant) surprises to a minimum and increased the confidence in the UIMS. The users of the UIMS seemed to enjoy using the system. I had observed some of the users playing with the facility even after their main task was over. Most people who had seen the UIMS in action also appreciated the ability to make changes and see the effect of the changes immediately.

A result of using the UIMS for building interfaces for a variety of applications was that a number of problems with the UIMS's own interface were discovered. These did not involve any major changes to the UIMS, and were mostly oversights on my part or were minor annoyances that I had just ignored. The interface of the UIMS was revised to fix these problems as they were discovered. Examples of these problems included cryptic error messages, unlabeled interaction techniques, and minor inconsistencies in interaction in vu.

When performing refinements using vu, some of the users did not like enlarging smaller objects to change the objects' colours (see, e.g. page 169 second last paragraph). This is necessary as changing attributes is treated as one step in the system, and therefore to display all of the attributes the smaller objects have to be enlarged. A better alternative would have been for the UIMS to enlarge the smaller objects automatically, and bring them back to their original sizes when the changes are over.

Some of the users did not use the facilities in an optimal fashion. An example is when moving objects on the screen, users often did not select the corner which made the final placing of the object the easiest. But once told, they realized the inefficient use of the facilities and improved in future. A second

example is the use of the dialogue handling capabilities of Diction. One of the users wanted an extension made to Diction for handling a special requirement (variable number of a particular argument in a command) in his interface. But it turned out that his requirement could be handled within the existing facilities. Both of these problems could be avoided by providing the users with a manual which explained the efficient and effective use of the UIMS.

8.3. Implications of the Approach

There are a number of interesting implications of the approach followed by the UIMS. The most important advantage is that the interface is described in a simple and concise notation. As the experience of the users of the UIMS indicates, this notation is readily learned and understood. Also since the description is quite concise, it is easily modified and a new interface produced without a major investment of time or effort. This makes it possible for the designer to devote more resources to trying out different alternatives, rather than to getting a new version of the interface produced. This compares favourably with systems which require detailed program-like specification of the interface.

The second advantage of the kind of specification accepted by the UIMS is that the UIMS can automatically provide a number of sophisticated facilities. An example is the help facility provided by the UIMS. The UIMS automatically generates help messages which explain the syntax, selection, as well as the argument requirements of commands. This is a direct consequence of the UIMS "knowing" about the command structure. A second example is the repeated execution of a command by changing just one of the argument values. In this type of a scenario the user provides a number of values for a particular command argument and the UIMS repeatedly executes the selected command for each argument value. Providing this type of a facility will be harder, if not impossible, in a grammar or transition networks based specification.

The third advantage of the approach is that it leads to separation between the application and the user interface. Because of this separation it is possible to alter the complete user interface with little or no change to the application code. This makes it possible to move the complete application to other devices

and change the dialogue style completely. This has been demonstrated by porting the complete distributed network editor and the skeleton editor from an AED-767 colour graphics device to a SUN3 workstation. Both the applications were ported in a matter of 2-3 hours, rather than in weeks or months which is usually the case.

The next advantage of the approach is its extensibility and flexibility. The UIMS can be tailored to handle different interface styles. For example, it is possible to extend the UIMS to produce interfaces which conform to the OPEN LOOK style of interfaces [OPEN LOOK88]. Once the core of the UIMS is produced it is then possible to keep extending it by adding more interaction techniques and dialogue styles.

Another advantage is the interfaces produced by the UIMS have a uniform and consistent style within an application as well as across applications. The consistency in interfaces helps users learn new applications quickly. To a large extent, this consistency is achieved by a uniform application of design rules by the UIMS, rather than the designer enforcing various interface design guidelines.

There are a number of limitations of the approach. This approach does not work very well with internal control user interfaces in which the application controls the interaction. To be able to support the development of internal control interfaces a different input and control structure for the UIMS will be required. Another limitation is because there is a loose coupling between the application and its interface, certain kinds of semantic feedback becomes very slow or impossible to produce. These include the ones which require knowledge about the application computed data, or output which is closely tied to the input.

8.4. Relation to Existing Approaches/UIMSs

The research presented in this thesis has benefited from earlier work in the area of user interfaces and UIMSs. As a result of this, the approach and UIMS described in this thesis is closely related to a number of existing systems. This section compares and contrasts our research with other approaches and UIMSs. The discussion can be divided into two parts based on the abstractions used by the UIMS.

8.4.1. Low-Level UIMSs

There is a large number of UIMSs which use "programmed style" of specification of the interface. The specification could be textual in which case the designer specifies low-level lexical and syntactic detail of the interface in a textual notation accepted by the UIMS, or the specification could be non-textual, in which case a visual or graphical specification is used. Examples of UIMSs which use textual specification include Reisner's UIMS [Reisner81], SYNGRAPH [Olsen83], UofA UIMS [Green85b], and Sassafras [Hill86]. Examples of UIMSs which use non-textual specification include Jacob's State Transition Network UIMSs [Jacob85, Jacob86], Wasserman's UIMS [Wasserman85], MENULAY [Buxton83], Peridot [Myers86a], Cardelli's UIMS [Cardelli87, Cardelli88]. In these UIMSs, the level of abstractions for describing the information communicated between the user and system is very low. The events used in such specifications may range from primitives of the underlying graphics package (e.g. picks, valuators, strings) to more complex I/O events (e.g. menu selections, bargraphs). A number of problems result because of this. First, the applications quite often have to convert data into its graphical representation and request the UIMS to display the result. As a consequence, any modification in the user-visible parts of the system means modifications in the application. A second problem is that applications generally expect to receive input in the form of keystrokes and mouse movements. This makes for a poor environment for integrating various applications. When two applications have to be integrated, the output from one application has to be converted into keystrokes and mouse movements for the other application. Such a conversion can be very difficult to perform. A third problem is that it is very difficult to move application from one device to another as a large amount of device dependent interface code resides in the application.

Another problem with low-level UIMSs relates to the degree of abstraction in specifying the ordering of I/O events. In these UIMSs, the interface definition explicitly specifies what sequences of I/O events constitute a valid dialogue between user and application. In this type of specification, it is very hard to produce modeless interaction. The modelessness has to be programmed into the interface definition. Which means that the specification could become very hard to construct and manage. Another problem with expli-

cit event ordering is the handling of exception conditions. In this type of specification, all exception conditions must be foreseen and incorporated into the interface definition.

Our approach is based on a high-level semantic definition rather than on lexical/syntactic definition of the interface, as is the case in low-level UIMSs. In our approach, the communication between the user and the application is specified in terms of information that the application needs to obtain from the user. The interface definition does not specify the I/O events necessary to achieve this exchange of information. In our UIMS, more work is shifted from the application to the UIMS which is responsible for gathering low level device inputs, converting them into tokens for the application, and displaying the application data. As a result the application does not have to deal with low-level device input/output primitives. This provides for a better framework for intergrating applications.

In our approach, the interface definition does not explicitly specify the I/O event ordering in the communication between the user and application. Instead, it declares sets of events without mentioning a specific ordering. Any ordering restrictions are implicit in the UIMS. As a result of this it is easy to provide a modeless interaction. Also, since the UIMS has a complete knowledge about the data requirements of commands, it can automatically provide support for a number of useful facilities such as help, erase, and cancel.

8.4.2. High-Level UIMSs

Our approach is similar in spirit to the approaches followed in COUSIN (COoperative USer INterface) [Hayes83-Hayes85], MIKE (Menu Interaction Kontrol Environment) [Olsen86], and UIDE (User Interface Design Environment) [Foley87a-Foley89]. All of these systems emphasize on producing interfaces from highly abstracted interface descriptions. Differences lie in the kind of abstractions used and how the interface definition is used.

COUSIN provides coarse-grained command interaction centered around a form-based metaphor of communication. Using this abstraction, an interface definition for a given application specifies a form con-

taining a field for each piece of information that the user and application need to exchange. This definition is interpreted by COUSIN to realize a graphical representation of the form which shows the current value of each field, which the user or the application can update. This type is abstraction is suitable for a limited class of applications only, such as a print command and electronic mail applications. When a number of different commands exist in an interface, the form based abstraction does not produce very good interaction.

MIKE uses a "command procedure" metaphor for describing the user interface. Corresponding to each command in the interface there is a procedure/function in the application which implements the semantics of the command. While the interface definitions used in MIKE and in our approach look quite similar there are differences in how this definition is used. The first difference is the default interfaces generated by MIKE are almost completely keyboard oriented. In MIKE, the interface definition is used almost solely for syntax purposes. In our approach the data type and other properties of the commands and command arguments are used to produce graphical interfaces. Also, by considering the device properties, our UIMS does a more comprehensive job of designing interfaces than MIKE. The second difference is in the control mechanism. As described in section 2.6.2, the use of functions and the types they return form the basic control mechanism in MIKE's syntax. This makes it very natural to support a prefix type of syntax in which first the command is selected and then the command arguments are entered in the sequence of their specification in the interface definition. This is the only one possible style supported by MIKE, which is highly moded in structure. A number of ambiguities would have to be resolved if MIKE were to parse for command arguments in any order, or if it were to implement other syntax types. Providing these facilities may require a complete overhaul of MIKE's control mechanism. The control mechanism in our UIMS is very flexible, which makes it possible to support modeless interaction. Currently, it supports prefix, postfix, and nofix syntax types. Providing support for additional syntaxes simply means extending Diction to produce event handlers for them.

UIDE provides a high-level conceptual design tool in which the interface is described as a knowledge

base consisting of the class hierarchy of objects, properties of the objects, actions which can be performed on the objects, and pre- and post-conditions for the actions. UIDE can algorithmically transform the knowledge base into a number of functionally equivalent interfaces, each of which is slightly different from the original interface. The transformed interface definition can then be used to automatically implement the interface. UIDE facilitates the creation and testing of design alternatives by performing transformations on the knowledge base which defines the interface. There are two points to note about the creation of design alternatives in UIDE. First, the alternatives differ only slightly from one another. It seems hard to produce radically different interfaces for an application. As a result, it may be proper to conclude that UIDE facilitates the exploration of designs alternatives in a close vicinity of the initial design provided by the designer. The second point to note is that when producing design alternatives the designer decides which transformations to apply. Once the designer provides this information, UIDE performs the mechanics of creating implicit parameters and adding commands etc. As opposed to UIDE, our UIMS enables the designer to easily create interface designs which are radically different from one another. Changing the syntax of commands or influencing the selection of interaction techniques is more easy and automatic in our UIMS than in UIDE. An area where UIDE provides more help than our UIMS is in producing consistent designs. An example is when the designer wants to factor a command argument, UIDE makes sure that all the affected commands are updated. This does not happen automatically in our UIMS; the designer has to update all the affected commands.

Other notable differences between our UIMS and other UIMSs discussed in this section are in the consideration of user's preferences and in the designer's control of the UIMS. None of the other UIMSs place much emphasis on user's preferences and on the designer's control of the UIMS's functioning. In our UIMS, optional inputs can be provided to tailor the generated interface according to the user's preferences. Our UIMS provides the designer with a number of ways of enfluencing the decisions made by the UIMS. This is supported by using designer-modifiable defaults in the UIMS.

8.5. Review of Contributions

The research presented in this thesis shows that it is possible to automatically produce the initial design of graphical user interfaces. This initial design can then be refined very easily and rapidly by the designer, and automatically implemented by the UIMS. To the best of our knowledge, the UIMS presented in this thesis is amongst the first to follow such an approach, and takes the state-of-the-art beyond the capabilities of a "conventional" UIMS.

The UIMS supports extremely rapid prototyping of user interfaces by reducing the time and effort involved in creating user interfaces. The designer can quickly create different prototypes by modifying the UIMS's defaults or by providing optional inputs. Also, significant portions of the presentation component can be changed without affecting the dialogue control component, and vice-versa. The ease and efficiency of creating prototypes encourages experimentation and may therefore lead to better user interfaces. The interfaces generated by the UIMS are efficient enough so that they can be used with actual applications. This means the UIMS is not merely used for prototyping; the actual interfaces are being implemented.

This research makes a significant step forward in the direction of increasing the ease of use of UIMSs. The interface designer is no longer required to deal with detailed interface specifications, which are often cryptic, too time-consuming to produce, and error-prone. In our UIMS, a high level description of the commands supported by the application is directly transformed into the interface design and implemented. In addition, when editing interfaces, the changes are immediately visible and executable. This makes it much easier to develop interfaces.

The implementation of the user interface is physically separated from the implementation of the application routines, yet these two components can communicate at run-time. This separation is an implicit goal of all UIMSs, but few existing UIMSs achieve it to the extent it is achieved in our system.

The system Diction demonstrates that a variety of syntax types can be automatically designed and implemented, and that different syntax types can co-exist in the same interface. As far as we know, Diction is amongst the first systems to allow this.

To the best of our knowledge, Chisel is the first system to handle the design of graphical presentation components at a detailed level. Chisel demonstrates that it is possible to generate presentation components which are sensitive to user preferences, designer guidelines, and hardware devices.

The vu system allows the editing of graphical presentation components in a manner which is highly interactive, graphical and direct. It combines visual programming, object oriented design, and self-updating help systems to achieve ease of use. Visibility and direct manipulation of the objects of interest, and the ability to interact with the interface under development are the major strengths of vu.

8.6. Future Work

Although the UIMS described in this thesis allows interfaces to be created, edited, and used in actual applications, there is a great deal that could be done to make it better. The experience gained through this research and the development of the UIMS has suggested some new approaches and directions that might be profitably investigated in the future.

8.6.1. Additional Features

This section lists some additional features that could be added to the UIMS to make it more useful. These do not involve radical changes to the UIMS's interface or implementation.

8.6.1.1. Chisel Reading in Refined Presentation Components

Currently, Chisel cannot read in presentation components after they are revised through vu. It always starts afresh from the dialogue requirements file and produces a new design. This design is then revised by the designer by using vu. If Chisel were able to read in revised designs and incrementally add new interaction techniques to it, the designer could save the time he has already spent in revising the old design. This feature was not added to Chisel as it did not present any interesting research questions, and due to the time constraint on the implementation.

8.6.1.2. Chisel Producing Presentation Database

An important feature that is missing from Chisel is its ability to produce the database for the presentation component that is used at the run-time. Currently, Chisel produces a "design_file" which is read by vu which produces the database. As a result, even if the designer does not want to refine or add output token information to the design produced by Chisel he still has to use vu to convert the design into a database. This feature was not added because of the large implementation effort involved in doing it. Since Chisel is written in Franz Lisp, and the database can only be accessed through C programs, I would have to write code which could interface them properly. And, this required substantial effort.

8.6.1.3. Interactive Entry of the Input Syntax

Currently, the dialogue requirements file for Diction is produced by using a text editor. It would be easier to enter the same information through an interactive syntax-directed editor which could flag syntax errors and provide help during the entry.

8.6.2. Areas for Future Research

As stated in section 1.4, this thesis does not provide the final solution to the user interface design problem, neither does it produce a comprehensive tool for designing user interfaces. This is a very large and complicated task and needs more time and resources than available to a graduate student. There is a considerable amount of work yet to be done in the area of user interface design and user interface design tools. This section presents some interesting areas for future research.

8.6.2.1. Run-Time Structures

An important area for research is to compare run-time structures of UIMSs. The comparison can be based on factors such as efficiency of the structure, reliability of the structure, and suitability for various types of interfaces. It may then be possible to produce a UIMS which can support multiple run-time structures, and depending on the type of interface being supported select the one which matches the requirements the best.

8.6.2.2. Automatic Analysis

One of the areas where not much work has been done is the evaluation of graphical user interfaces. The results of this research could be profitably used in a high level UIMS such as ours. If Chisel had an evaluation system built in it, it could evaluate the designs it generates and improve upon them. This could reduce the time the interface designer spends in manually refining the user interface.

The transparent, automated interaction monitors can be extremely useful for evaluating and improving interface designs, but not much work has been done in this area. Buxton et al [Buxton83] and Olsen and Halversen [Olsen88] produce data from the user/system interaction, but they have not built tools for analyzing this data. Considerable additional research attention needs to be given to the use of such monitors while evaluating user interfaces.

8.6.2.3. Application Output

Another issue which needs more research is the issue of handling the application feedback in user interfaces. In a number of UIMSs, including ours, this is left for the interface designer to design and program. Some recent UIMSs which have initiated work in this area include [Hudson86, Hudson88], [Mackinlay86] and [Zanden88]. An interesting approach could be to produce interactive tools which enable the designer to establish relationship between application information (variables) and the graphical display of a user interface. The designer should be allowed to use graphical marks, such as points, lines, and areas, to encode information via their positional, temporal, and retinal properties.

8.6.2.4. Higher Level Design Tools

Great interest is beginning to arise in creating high level UIMSs. This thesis along with Dan Olsen's [Olsen86] and James Foley et al's [Foley87a-Foley89] work provide a beginning in this direction. An interesting approach could be to automatically determine the commands along with their arguments, and the application routines to perform certain actions. In this type of a scenario each application routine is specified in terms of its data requirements, its pre-conditions, and its post-conditions. The designer specifies the pre- and post-conditions of the actions that the user wants to perform. Given this information, the UIMS determines the commands to perform these actions and selects the application routines required to implement the semantics. Once this is done, a UIMS, such as the one presented in this thesis, can be used to design and implement the interface.

This basic idea could be extended to produce user interfaces which adapt to the user's level of expertise or security clearance. Each command in the interface can be assigned a severity or security rating. The system could then prevent users below a certain expertise or security rating from executing commands which have a higher rating. The other option could be to ask novice users for confirmation before executing a potentially dangerous command.

8.6.2.5. User Interface Design Guidelines

Though great advances have been made over the past few years, the state of the art in user interface design is still in its infancy. The area that seems to be in greatest need of work has to do with user interface design guidelines. A number of guidelines for user interface design do exist (see, e.g., [Shneiderman80] and [Foley82] for a compilation of guidelines), but most of these guidelines are too general and therefore do not help much when dealing with specific situations. Also a number of the guidelines have not been evaluated experimentally for their usefulness and validity. A recent effort by AT&T, Xerox, and Sun Microsystems has resulted in a document called the OPEN LOOK Graphical User Interface Functional Specification [OPEN LOOK88]. This specification describes guidelines for graphical user interfaces. The

aim of this document is to "bring ... consistent interface to users, no matter who builds the application and what hardware the software runs on" [Kannegaard88]. The guidelines presented in this document are more specific than the ones which were available so far. We need to get user feedback on the interfaces based on these guidelines to check the guidelines' validity. Also, we need to keep similar efforts going at other research and development institutions. The hope is that in future when we have enough good guidelines and appropriate tools, designing user interfaces will become more predictable and more enjoyable.

References

Armstrong85a.

William W. Armstrong and Mark Green, The Dynamics of Articulated Rigid Bodies for Purposes of Animation, *Proc. Graphics Interface'85*, Montreal, Canada, May 27-31, 1985, 407-415.

Armstrong85b.

William W. Armstrong and Mark Green, The Dynamics of Tree Linkages for Purposes of Animation, *The Visual Computer 1*, 4 (1985), 231-240.

Barth86.

Paul S. Barth, An Object-Oriented Approach to Graphical Interfaces, *ACM Transactions on Graphics 5*, 2 (Apr. 1986), 142-172.

Betts87.

Bill Betts, David Burlingame, Gerhard Fischer, Jim Foley, Mark Green, David Kasik, Stephen T. Kerr, Dan Olsen and James Thomas, Goals and Objectives for User Interface Software, *Compter Graphics 21*, 2 (Apr. 1987), 73-78.

Boies85.

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

Buxton80.

William A. S. Buxton and R. Sniderman, Iteration and the Design of the Human-Computer Interface, *Proc. 13th Annual Meeting of the Human Factors Association of Canada*, 1980, 72-81.

Buxton83.

William A. S. Buxton, M. R. Lamb, D. Sherman and K. C. Smith, Towards a Comprehensive User Interface Management System, *Computer Graphics 17*, 3 (July 1983), 35-42. (Proc. SIGGRAPH'83 Conf., July 25-29, 1983, Detroit, Michigan).

Card80.

Stuart K. Card, Thomas P. Moran and Allen Newell, The Keystroke-Level Model for User Performance Time with Interactive Systems, *Comm. ACM 23*, (1980), 396-410.

Cardelli85.

Luca Cardelli and Rob Pike, Squeak: A Language for Communicating with Mice, *Computer Graphics 19*, 3 (1985), 199-204. (Proc. SIGGRAPH'85 Conf., July 22-26, 1985, San Francisco, CA).

Cardelli87.

Luca Cardelli, *Building User Interfaces by Direct Manipulation*, Technical Report# 22, Digital, System Research Center, 130 Lytton Av., Palo Alto, California 94301, Oct. 1987.

Cardelli88.

Luca Cardelli, Building User Interfaces by Direct Manipulation, *Proc. ACM SIGGRAPH Symposium on User Interface Software*, Banff, Alberta, Canada, Oct. 17-19, 1988, 152-166.

Chang87.

Shi-Kuo Chang, Visual Languages: A Tutorial and Survey, *IEEE Software 4*, 1 (Jan. 1987), 29-39.

Chia85.

M. S. Chia, *An Event Based Dialogue Specification for Automatic Generation of User Interfaces*, M.Sc. Thesis, Dept. of Computing Science, Univ. of Alberta, Edmonton, Alberta, Canada, 1985.

Feldman82.

M. Feldman and G. Rogers, Towards the Design and Development of Style Independent Interactive Systems, *Proc. 1st Annual Conf. on Human Factors in Computer Systems, Gaithersburg Maryland*, Mar. 1982, 111-116.

Foley82.

J. D. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison Wesley,

Reading Mass., 1982.

Foley84a.

James D. Foley, Managing the Design of User-Computer Interfaces, *Proc. Fifth Annual NCGA Conf. and Exposition, Anaheim, CA, Vol II,*, May 13-17, 1984, 436-451.

Foley84b.

J. D. Foley, V. L. Wallace and P. Chan, The Human Factors of Computer Graphics Interaction Techniques, *IEEE-Computer Graphics and Applications 4*, 11 (Nov. 1984), 13-48.

Foley87a.

James D. Foley, Transformations on a Formal Specification of User-Computer Interfaces, *Computer Graphics 21*, 2 (Apr. 1987), 109-113.

Foley87b.

James D. Foley, Won Chul Kim and Christina A. Gibbs, Algorithms to Transform the Formal Specification of a User-Computer Interface, *Human-Compter Interface - INTERACT'87*, B.V.(North Holland), 1987, 1001-1005.

Foley88a.

James D. Foley, Won Chul Kim, Srdjan Kovacevic and Kevin Murray, *The User Interface Design Environment*, Technical Report# GWU-IIST-88-4, Dept. of EE&CS, The George Washington University, Wasington, DC 20052, Jan. 1988. (16 pages).

Foley88b.

James D. Foley, Christina Gibbs, Won Chul Kim and Srdjan Kovacevic, A Knowledge-Based User Interface Management System, *Proc. CHI'88 Human Factors in Computer Systems*, Washington, D.C., May 15-19, 1988, 67-72.

Foley89.

James D. Foley, Won Chul Kim, Srdjan Kovacevic and Kevin Murray, Defining Interfaces at a High

Level of Abstractions, *IEEE Software*, Jan. 1989, 25-32.

Furuya87.

Katsuji Furuya, Shuichi Tayama, Eiko Kutsuwada and Kazuo Matsumura, Approach to Standardize Icons, *Proc. 1987 Workshop on Visual Languages*, Linkoping, Sweden, Aug. 19-21, 1987, 29-38.

Good84.

Michael D. Good, John A. Whiteside, Dennis R. Wilson and Sandra J. Jones, Building a User-Derived Interface, *Comm. ACM 27*, 10 (Oct. 1984), 1032-1043.

Green83.

Mark Green, M. Burnell, H. Vernjak and M. Vernjak, Experience with a Graphical Data Base System, *Proc. Graphics Interface'83*, 1983, 257-270.

Green84a.

Mark Green and Nancy Bridgeman, *WINDLIB Programmer's Manual*, Dept. of Computing Science, Univ. of Alberta, Edmonton, Alberta, Canada, 1984.

Green84b.

Mark Green, *The Design of Graphical User Interfaces*, Ph.D. Thesis, Univ. of Toronto, Toronto, Canada, 1984.

Green85a.

Mark Green, Report on Dialogue Specification Tools , in *User Interface Management Systems*, Gunther E. Pfaff (ed.), Springer-Verlag, 1985, 9-20.

Green85b.

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

Green86.

Mark Green, A Survey of Three Dialogue Models, *ACM Transactions on Graphics 5*, 3 (July 1986),

244-275.

Green87.

Mark Green and Jonathan Schaeffer, FrameWorks: A Distributed Computer Animation System, *Proc. CIPS (Edmonton) Annual Conf.*, Edmonton, Alberta, Nov. 16-19, 1987, 305-310.

Guedj80.

R. A. Guedj, P. J. W. ten Hagen, F. R. A. Hopgood, H. A. Tucker and D. A. Duce, eds., *Methodology of Interaction*, North-Holland Publishing Company, Amsterdam, 1980.

Hanau80.

P.R. Hanau and D.R. Leronovitz, Prototyping and Simulation Tools for User/Computer Dialogue Design, *Computer Graphics 14*, 3 (1980), 271-278. (Proc. SIGGRAPH'80 Conf., July 14-18, 1980, Seattle, Washington).

Hayes83.

Phil J. Hayes and P. A. Szekely, Graceful interaction through the COUSIN Command Interface, *Internation Journal of Man-Machine Studies 19*, 3 (Sep. 1983), 285-305.

Hayes84.

Phil J. Hayes, An Executable Specification Technique for Describing Human-Computer Interface, in *Advances in Human-Computer Interaction*, H. R. Hartson (ed.), Ablex, New Jersey, 1984, 161-189.

Hayes85.

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

Heindel75.

L.E. Heindel and J.T. Roberto, *LANG-PAK - An Interactive Language Design System*, American Elsevier Publishing Company, Inc., New York, 1975.

Helfman87.

Janathan Helfman, Panther: A Tabular User-Interface Specification System, *Proc. CHI+GI'87 Human Factors in Computing Systems*, Toronto, Ont., Canada, Apr. 5-9, 1987, 279-284.

Henderson86.

D. A. Henderson, The Trillium User Interface Design Environment, *Proc. CHI' 86 Human Factors in Computing Systems, Boston, MA*, Apr. 13-17, 1986, 221-227.

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

Hill87a.

Ralph D. Hill, *Supporting Concurrency, Communication, and Synchronization in Human-Computer Interaction*, Tech. Report CSRI-197, Computer Systems Research Institute, Univ. of Toronto, May 1987. (Ph.D. Thesis, 206 pages).

Hill87b.

Ralph D. Hill, Event-Response Systems - A Technique for Specifying Multi-Threaded Dialogues, *Proc. CHI+GI'87 Human Factors in Computing Systems*, Toronto, Ontario, Canada, Apr. 5-9, 1987, 241-248.

Hudson86.

Scott E. Hudson, A Generator for Direct Manipulation Office Systems, *ACM Transactions on Office Systems 4*, 2 (Apr. 1986), 132-163.

Hudson88.

Scott E. Hudson and Roger King, Semantic Feedback in the Higgins UIMS, *IEEE Transactions on Software Engineering SE-14*, 8 (Aug. 1988), 1188-1206.

Jacob83.

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

Jacob84.

Robert J.K. Jacob, An Executable Specification Technique for Describing Human-Computer Interaction, in *Advances in Human-Computer Interaction*, H.R. Hartson (ed.), Ablex Publishing Co., 1984.

Jacob85.

Robert J. K. Jacob, A State Transition Diagram Language for Visual Programming, *IEEE Computer 18*, 8 (Aug. 1985), 51-59.

Jacob86.

Robert J.K. Jacob, A Specification Language for Direct-Manipulation User Interfaces, *ACM Transaction on Graphics 5*, 4 (Oct. 1986), 283-317.

Johnson75.

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

Kannegaard88.

Jon Kannegaard, OPEN LOOK: Outlook/Overview, *Sun Technology 1*, 4 (1988), 58-62 .

Kasik82.

David J. Kasik, A User Interface Management System, *Computer Graphics 16*, 3 (July 1982), 99-106.

Kernighan78.

Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice_Hall, Englewood-Cliffs, NJ, 1978.

Lau85.S. C. Lau, *The Use of Recursive Transition Networks for Dialogue in User Interfaces*, M.Sc. Thesis,

Dept. of Computing Science, Univ. of Alberta, Edmonton, Canada, 1985.

Lesk75.

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

MacDonald82.

Alan MacDonald, Visual Programming, *Datamation 28*, 11 (1982), 132-140.

Mackinlay86.

Jock Mackinlay, Automating the Design of Graphical Presentations of Relation Information, *ACM Transactions on Graphics 5*, 2 (Apr. 1986), 110-141.

Mason83.

R.E.A. Mason and T.T. Carey, Prototyping Interactive Infromation Systems, *Comm. ACM 26*, 5 (May 1983), 347-354.

Mittal86.

Sanjay Mittal, Clive L. Dym and Mahesh Morjaria, Pride: An Expert System for the Design of Paper Handling Systems, *IEEE Computer 19*, 7 (July 1986), 102-114.

Morgan83.

C. Morgan, G. William and P. Lemmons, An Interview with Wayne Rosing, Bruce Daniels, and Larry Tesler, *Byte 8*, 2 (Feb. 1983), 90-114.

Mussio87.

P. Mussio, M. Padula and M. Protti, Description Based Icon Design, *Proc. 1987 Workshop on Visual Languages*, Linkoping, Sweden, Aug. 19-21, 1987, 118-129.

Myers86a.

Brad A. Myers and William A. S. Buxton, Creating Highly-Interactive and Graphical User Interfaces by Demonstration, *Computer Graphics 20*, 4 (1986), 249-258. (Proc. SIGGRAPH'86 Conf., Aug.

18-22, 1986, Dallas, Texas).

Myers86b.

Brad A. Myers, Visual Programming, Programming by Example, and Program Visualization: A Taxonomy, *Proc. CHI'86 Human Factors in Computing Systems*, Boston, MA, Apr. 13-17, 1986, 59-66.

Myers87a.

Brad A. Myers, Creating Dynamic Interaction Techniques by Demonstration, *Proc. CHI+GI'87 Human Factors in Computing Systems*, Toronto, Ont., Canada, Apr. 5-9, 1987, 271-278.

Myers87b.

Brad A. Myers, *Creating User Interfaces by Demonstration*, Tech. Report CSRI-196, Computer Systems Research Institute, Univ. of Toronto, May 1987. (Ph.D. Thesis, 266 pages).

Newman68.

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

OPEN LOOK88.

OPEN LOOK, *OPEN LOOK Graphical User Interface Functional Specification*, Sun Microsystems, San Francisco, CA, 1988.

Olsen83.

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

Olsen84.

Dan R. Olsen, William A. S Buxton, R. Ehrich, David Kasik, James Rhyne and John Sibert, A Context for User Interface Management, *IEEE Computer Graphics and Applications 4*, 12 (Dec.

1984), 33-42.

Olsen85.

Dan R. Olsen, Elizabeth P. Dempsey and Roy Rogge, Input/Output Linkage in a User Interface Management System, *Computer Graphics 19*, 3 (July 1985), 191-197. (Proc. SIGGRAPH'85 Conf., July 22-26, 1985, San Francisco, California).

Olsen86.

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

Olsen87.

Dan R. Olsen, David Kasik, James Rhyne and James Thomas, eds., Proc. ACM SIGGRAPH Workshop on Software Tools for User Interface Management, Nov. 1986, Battelle, Seattle, Washington, *Computer Graphics 21*, 2 (Apr. 1987), 71-147.

Olsen88.

Dan R. Olsen and Bradley W. Halversen, Interface Usage Measurements in a User Interface Management System, *Proc. ACM SIGGRAPH Symposium on User Interface Software*, Banff, Alberta, Canada, Oct. 17-19, 1988, 102-108.

Parnas69.

David L. Parnas, On the Use of Transition Diagrams in the Design of a User Interface for an Interactive Graphics System, *Proc. 24th National ACM Conf.*, 1969, 379-385.

Pfaff85.

Gunther E. Pfaff, in *User Interface Management Systems*, Springer-Verlag, 1985.

Reeves83.

William T. Reeves, Particle Systems - A Technique for Modeling a Class of Fuzzy Objects, *ACM Transactions on Graphics* 2, 2 (1983), 91-109.

Reisner81.

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

Roach82.

J. Roach, R. Hartson, R. Ehrich, T. Yunten and D. Johnson, DMS: A Comprehensive System for Managing Human-Computer Dialogue, *Proc. 1st Annual Conf. on Human Factors in Computer Systems*, Gaithersburg Maryland, Mar. 1982, 102-105.

Schmucker86.

K. J. Schmucker, MacApp: An Application Framework, *BYTE*, Aug. 1986, 189-192.

Sheil83.

Beau Sheil, Power Tools for Programmers, *Datamation 29*, 2 (Feb. 1983), 131-144.

Shneiderman80.

Ben Shneiderman, *Software Psychology*, Winthrop Publishers, Inc., Cambridge, Massachusetts, 1980.

Shneiderman82.

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

Sibert86.

John L. Sibert, William D. Hurley and Teresa W. Bleser, An Object-Oriented User Interface Management System, *Computer Graphics 20*, 4 (1986), 259-268. (Proc. SIGGRAPH'86 Conf., Aug. 18-22, 1986, Dallas, Texas).

Singh85.

Gurminder Singh, *Presentation Component for the U of A UIMS*, M.Sc. Thesis, Dept. of Computing Science, Univ. of Alberta, Edmonton, Canada, 1985.

Singh86.

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

Singh87.

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

Singh88a.

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

Singh88b.

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

Singh89a.

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

Singh89b.

Gurminder Singh and Mark Green, Generating Graphical User Interfaces from High-Level Descriptions, *Graphics Interface' 89*, London, Ontario, Canada, June 19-23, 1989, (in press).

Singh89c.

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

Smith82.

David C. Smith, Charles Irby, Ralph Kimball, Bill Verplank and Erik Harslem, Designing the Star

User Interface, *Byte Magazine*, Apr. 1982, 242-282.

Sutton78.

Jimmy A. Sutton and Ralph H. Sprague, *A Survey of Interactive Business Applications*, IBM Research Report RJ2388, Nov. 9, 1978.

Swartout82.

W. Swartout and R. Blazer, The Inevitable Intertwining of Specification and Implementation, *Comm. ACM 25*, 7 (1982), 438-440.

Tanner85.

Peter Tanner and William A. S. Buxton, Some Issues in Future User Interface Management System (UIMS) Development, in *User Interface Management Systems*, Gunther E. Pfaff (ed.), Springer-Verlag, 1985, 67-80.

Thomas83.

James J. Thomas and Griffith Hamlin, eds., Graphics Input Interaction Technique (GIIT) Workshop Summary, June 2-4, 1982, Battelle Seattle, *Computer Graphics 17*, 1 (Jan. 1983), 5-66.

Wasserman79.

Anthony I. Wasserman and S. K. Stinson, A Specification Method for Interactive Systems, *Proc. IEEE Conf. on Specification of Reliable Software, Long Beach, CA*, 1979, 68-79.

Wasserman85.

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

William83.

Gregg William, The Lisa Computer System, *Byte Magazine*, Feb. 1983, 33-50.

Zanden88.

Bradley T. Vander Zanden, Constraint Grammars in User Interface Management Systems, *Proc.*

Graphics Interface'88, Edmonton, Alberta, Canada, June 6-10, 1988, 176-184.

Appendix A1

BNF Specifiction for Diction

Note: the following symbols are meta-symbols belonging to the BNF formalism, and are not part of the interface specification

::= | { }

When curly brackets are part of the specification, they are printed in bold. The curly brackets denote possible repetition of the enclosed symbols zero or more times. In general,

A ::= {B}

is a short form for the purely recursive rule:

A ::= <empty> | AB

<description> ::= <parse selection> <global args> <commands>
<parse selection> ::= <empty> | <parse> <selection> | <selection> <parse>
<parse> ::= <empty> | **PARSE** <parse type>
<selection> ::= <empty> | **SELECTION** <selection type>

<global args> ::= {<global arg>}
<global arg> ::= <identifier> : <garg spec>
<garg spec> ::= <garg type> <initval>

<garg type> ::= <type> | <subrange> | <enumerated>
<subrange> ::= [<number> : <number>]
<enumerated> ::= (<identifier> {<identifier>})

<initval> ::= <empty> | **{ INITIAL** = <val> **}**
<val> ::= <number> | <identifier>

<commands> ::= <command> {<command>}
<command> ::= <identifier> <pstype> (<args>)
<pstype> ::= <empty> | **{** <ptype> <stype> **}**
<ptype> ::= <empty> | <parse type>
<stype> ::= <empty> | <selection type>
<args> ::= <empty> | <arg> {; <arg>}
<arg> ::= <identifier> : <arg spec>
<arg spec> ::= <arg type> <defval>
<arg type> ::= <identifier> | <garg type>

<defval> ::= <empty> | { **DEFAULT** = <val> }

<parse type> ::= **PREFIX | POSTFIX | NOFIX**
<selection type> ::= **OPEN_ENDED | CLOSE_ENDED**
<type> ::= **int | real | char | window | pick2d | pick3d**
<number> ::= <integer number> | <real number>
<integer number> ::= <sign> <unsigned integer>
<unsigned integer> ::= <digit> {<digit>}
<sign> ::= <empty> | + | -
<real number> ::= <sign> <unsigned integer> . <digit> {<digit>}
<identifier> ::= <letter> {<letter or digit>}
<letter or digit> ::= <letter> | <digit>
<letter> ::= **A | B | ... Y | Z | a | b | ... | y | z**
<digit> ::= **1 | 2 | ... 8 | 9 | 0**
<empty> ::=

Appendix A2

Developing an Interface for FrameWorks

This section presents an introduction to the FrameWorks model of distributed computing and discusses the creation of its interface using the UIMS described in this thesis. This interface is designed for use in the distributed computing research [Green87, Singh89c] in our department. The primary developer of this interface was Glenn Klettke, a fourth year computing science undergraduate student at the University of Alberta. Glenn's experience with using the UIMS is presented in Appendix A3. The introduction to the FrameWorks model was written by Ajit Singh, and edited by me.

1. Introduction

The FrameWorks model offers the designers of distributed applications a conceptual framework to support facile construction of distributed programs. In this model, the organization of processes is described in a high level language using basic structuring primitives of the model called **templates.** These templates can be viewed analogically as sub-structures commonly found in human organizations. From the architectural point of view the resulting structure is similar to a data flow network of processes. Since processes are organized in a network so that the output of a process can be fed into appropriate input of other processes, all the processes whose input values are ready can be executed simultaneously.

The templates provide simple structures that can further be used for building much more complicated organizations. A process's inter-connection with other processes can be described by a 3-tuple:

(input_template, output_template, body_template)

The designer selects a suitable template from a set of templates for each type. Input templates describe the interface through which a process receives its input. There are three options for input template: **executive, pipeline** and **assimilator.** A process with executive template has its standard input connected to the user's

terminal. Only one process in the whole application is allowed to have this status. In a sequential environment it is analogous to the main program. A process with pipeline input receives its input from a single process, whereas in the case of an assimilator, several processes supply parts of input data needed by the process.

Similarly, there are three choices for describing the output characteristics of a process: **pipeline**, **manager**, and **terminal**. A pipeline process passes all its output to a single process. A manager distributes its output among a number of processes. A process whose output is marked as terminal does not pass its output to any process. Normally, by the time processing of data is completed by a process whose output is of type terminal, any useful output is already stored in files or tables. Together, input and output templates provide several ways of structuring the input and output of a process.

In a network of processes very often there are processes that require much more computation than other processes. In order to achieve more flexibility at execution time in using idle processors and relieving the designer from specifying the exact number of processors required by such processes (and consequently by the complete application), the model provides a body template type called **contractor**. When a process's body is declared as contractor, it means that the process gets its work done by **employee** processes. A contractor process hires an unspecified number of employee processes to get the job done.

Processes in a distributed environment often need to access resources such as files, tables, devices etc. The last of the primitives used by the FrameWorks model is designed precisely for this purpose. It is called **resource manager.** A process that is supposed to regulate access to a shared resource is declared as resource manager. The only way other processes can perform operations on any resource is by sending a request to its resource manager.

Although a textual description of the processes' characteristics and their interconnection is possible, it is certainly not very convenient. A graphical user interface is more convenient for systems that need to deal with designs that have several components interconnected together. A graphical representation is much easier to produce and understand as compared to the textual description.

Load (application_name)
Add_First_Template (icon_name, place, process_name, script_file)
Add_Second_Template (icon_name, place)
Remove_Template (place)
Make_Contractor (process_name)
Remove_Contractor (process_name)
Add_Resource_Manager (process_name, place, script_file)
Add_Routines (process_name, calling_frame, reply_frame, routine_name)
Remove_Resource_Manager (place)
Add_Link (template1, template2, calling_frame, reply_frame)
Remove_Link (template1, template2)
Save (application_name)
Close (application_name)
Quit ()

Figure A2.1 Commands Supported by the Distributed Network Editor

2. Developing the Distributed Network Editor

The main purpose of the distributed network editor is to interactively enter and edit templates and connections between them. A preliminary task analysis of the editor by the designer resulted in the commands shown in figure A2.1. The editor was designed to allow the creation and editing of template networks for multiple applications in the same editing session. This necessitated the addition of Load, Close, Save, and Quit commands in the editor. The Add_First_Template command is used to create either an input or an output type template. Each template created by the Add_First_Template command is placed at a particular location on the display, and assigned a process name and a script file name by the user of the editor. The argument script_file is the name of the file which contains the location of the program for the process, and the structure definitions of the input and output data of the program. The Add_Second_Template command is used to create the second template in the input-output template pair. The Make_Contractor and Remove_Contractor commands are used to handle the creation and destruction of contractor templates. The Add_Resource_Manager and Add_Routines commands are used to creat a resource manager and update its properties. The calling_frame and reply_frame arguments in the Add_Routines command define the structures of the data received and generated by the resource manager.

```
ICONS : (executive inpipeline pipeline terminal assimilator manager),
APPLICATION_NAME : char,
PROCESS_NAME : char,
FRAME_NAME : char,
ROUTINE_NAME : char,
FILE_NAME : char,
WIND : pick,
Load (application_name : APPLICATION_NAME)
Add_First_Template (icon_name : ICONS, place : WIND,
        process_name : PROCESS_NAME, script_file : FILE_NAME)
Add_Second_Template (icon_name : ICONS, place : WIND)
Remove_Template (place : WIND)
Make_Contractor (process_name : PROCESS_NAME)
Remove_Contractor (process_name : PROCESS_NAME)
Add_Resource_Manager (process_name : PROCESS_NAME,
        place : WIND, script_file : FILE_NAME)
Add_Routines {OPEN_ENDED} (process_name : PROCESS_NAME {CSV},
        calling_frame : FRAME_NAME, reply_frame : FRAME_NAME,
        routine_name : ROUTINE_NAME)
Remove_Resource_Manager (place : WIND)
Add_Link (template1 : WIND, template2 : WIND,
        calling_frame : FRAME_NAME, reply_frame : FRAME_NAME)
Remove_Link (template1 : WIND, template2 : WIND)
Save (application_name : APPLICATION_NAME)
Close (application_name : APPLICATION_NAME {CSV})
Quit ()
```

Figure A2.2 Command Description for the UIMS

```
(defun user-preferences()
        (assign-colour 'WIND 'black 'white)
)
```

Figure A2.3 User's Preferences

And, the routine_name argument is used to add action routines to the resource manager. A resource manager can be destroyed by using the Remove_Resource_Manager command. The Add_Link and Remove_Link commands are used to create and remove links between a pair of templates.

Table A2.1 Output Tokens

WINDOW	Token Name	Display Procedure
WIND	IICON	Display_input_icon
WIND	OICON	Display_output_icon
WIND	IERASE	Erase_input_icon
WIND	OERASE	Erase_output_icon
WIND	MAN	Display_manager
WIND	ERASEMAN	Erase_manager
WIND	CONT	Display_contractor
WIND	ERASECONT	Erase_contractor
WIND	LINK	Display_link
WIND	UNLINK	Erase_link

The description of commands shown in figure A2.1 was refined and described in the notation accepted by the UIMS. This resulted in the command description shown in figure A2.2. It was decided to parse all commands, except the Add_Routines command, in PREFIX CLOSE_ENDED fashion. The Add_Routines command is parsed in PREFIX OPEN_ENDED fashion which allows the user add a number of routines to a resource manager without having to reselect the command. The process_name argument in the Add_Routines command and the application_name argument in the Close command are of CSV type; all other arguments are of regular type.

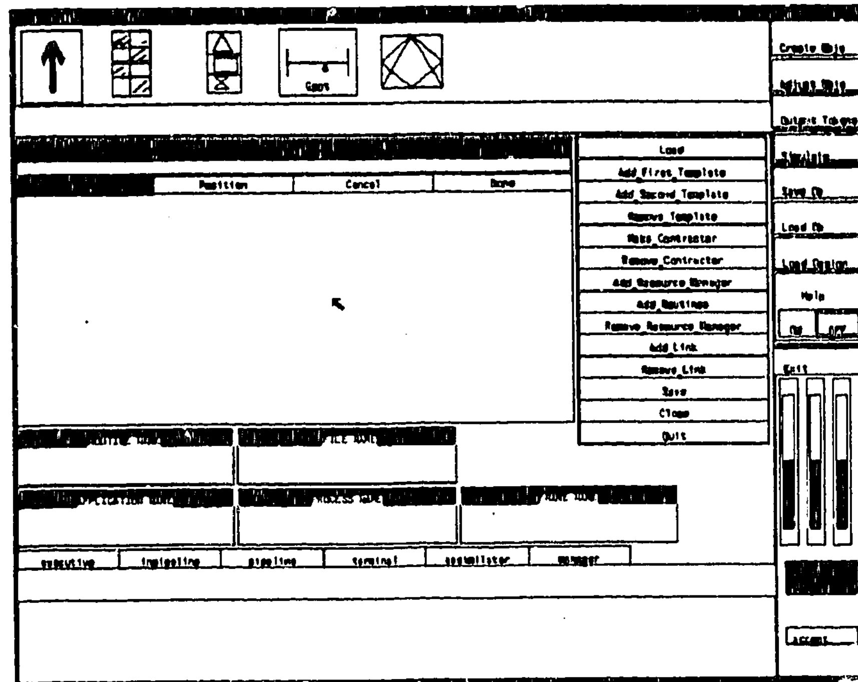


Figure A2.4 Presentation Component Generated by Chisel

The editor was to be implemented on AED-767 colour graphics terminal, so the device description for AED-767 (shown in figure 5.2 in chapter 5) was selected from the library of device descriptions and provided to the UIMS. The user's preferences part of the input was used to assign preferred colours to interaction techniques. This input is shown in figure A2.3.

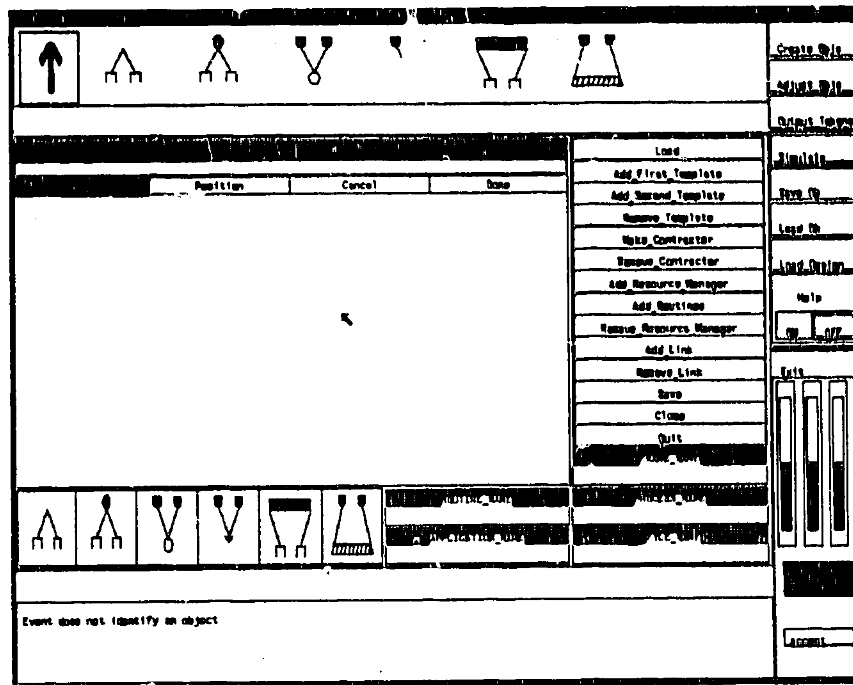


Figure A2.5 Refined Presentation Component

Table A2.1 shows the output tokens that are used by the editor. All of the output tokens are associated with the window named WIND. Five of these token (IICON, OICON, MAN, CONT, and LINK) are used when templates are added and the other five are used when templates are deleted. The UIMS was then used to produce the presentation and dialogue control components of the editor. The initial presentation component generated by Chisel is shown in figure A2.4. This presentation component after adding the output token information to it and after refining is shown in figure A2.5. The refining was done by using the facilities provided by vu. The completed editor in action is shown in figure A2.6.

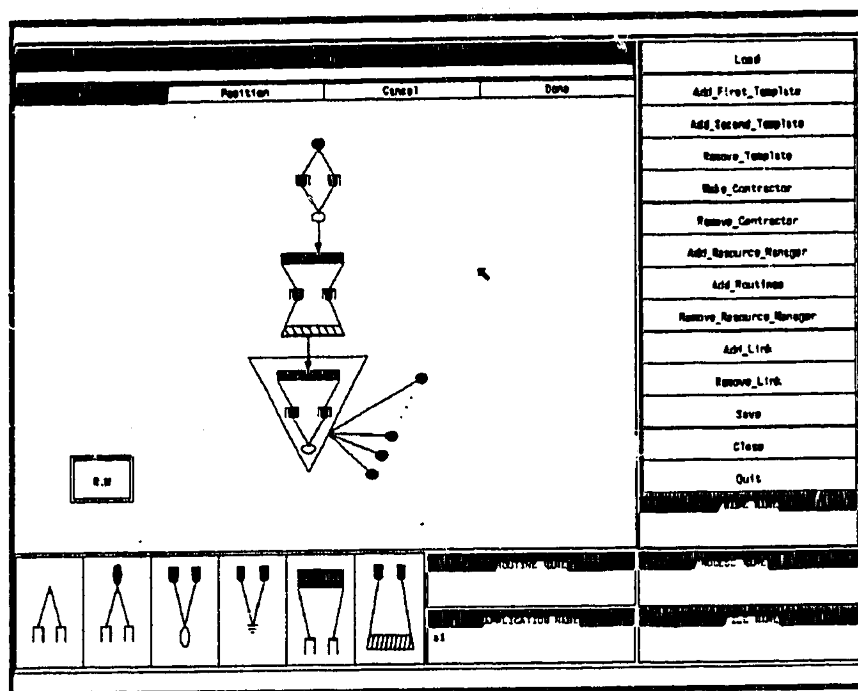


Figure A2.6 Completed Editor in Action

Appendix A3

Experience with Using the UIMS

This appendix presents Glenn Kelttke's experience with using the UIMS for developing the distributed network editor described in appendix A2. The rest of this appendix has been written by Glenn himself.

In need of a graphical user interface for a distributed processing application, I decided to give the UIMS described in this thesis a chance. One of my goals was to produce a working prototype as quickly as possible with minimum effort and allow successive refinements as needed. This was an important issue since all the details of the project were not known in advance. I wanted to be able to develop the application and interface in a parallel manner and be able to interact with the interface under development. The project was to be device independent so the UIMS would have to allow a mechanism whereby simple adjustments could be made to allow independence.

The project was to produce a interactive graphics facility. In its simplest form, the user would interact with the system and produce a picture of a distributed network system. The user by selecting commands from a main menu, controls the addition, deletion, linking, and removal of icons representing processors. Other commands were to be provided for loading and saving current or previously generated network systems.

Producing the initial specification for the prototype was simple using the UIMS. All that was required was to write a simple specification. I did not have to spend a great deal of time learning a complex specification language nor worry about a large number of details. One of the nice advantages was that the entire design was handled by the UIMS, thus, I did not have to produce a design and convert into a special format that the UIMS would understand.

Once the initial design was complete, I started adding more details to my interface. Successive refinement was very simple and did not require a great deal of effort. The UIMS allowed me a great deal of freedom to control almost all aspects of style. The ability to see the effect of refinements without delay was very satisfying and allowed me to see if in fact my refinements were satisfactory or not. The ability to create a large number of interfaces quickly, each with different behavior and look, allowed me to compare different models and choose the one I thought best.

Using the UIMS saved me a great deal of time and trouble. To program the whole interface from scratch would have taken considerably more effort. There was no need to create menu or interaction technique code and the code required to maintain them. I did not have to worry about the syntax checking or make sure that all information required to execute a command had been in fact entered since the UIMS does all of this. In the past I have used the Macintosh Toolbox for a few applications. As far as I know, there is no mechanism whereby the same effect or versatility can be achieved. Using the presentation component of the UIMS I could move a menu around the display window, change background and foreground colors, and resize a menu in minutes. Comparatively, to do the same thing from scratch would have necessitated changing a fair amount of code for each operation each time a change was required. Another example of time saving would be the fact that I wrote the initial application on a AED 767 Colorware computer and ported my application to a SUN3 workstation in less than one hour.

There were a few things that I would have liked to see included or changed in the UIMS. One of these is a way to allow a variable number of arguments for a particular command. Another was that in order to change the color of a small window, one had to enlarge it to a reasonable size and then change the color, and finally reduce it back to it's initial size. Finally, the initial layout of the interface by the UIMS tends to be less than ideal; however, the initial layout is easily and quickly changed.

I believe that the savings in programming time and effort provided by the UIMS where substantial. I would not hesitate to use the UIMS again in the future and look forward in doing so.

Appendix A4

Event Handlers for the Skeleton Editor (section 3.3)

This appendix contains the event handlers produced by Diction for the example dialogue shown in figure 3.6.

```
eventhandler help is
  token
    Change_Root IN_Change_Root;
    Add_Limb     IN_Add_Limb;
    Remove_Limb       IN_Remove_Limb;
    Move_Limb  IN_Move_Limb;
    Show_Attributes     IN_Show_Attributes;
    Change_Orientation IN_Change_Orientation;
    Change_Length       IN_Change_Length;
    Change_Mass IN_Change_Mass;
    Change_Cofm IN_Change_Cofm;
    Change_Torque       IN_Change_Torque;
    Save     IN_Save;
    Load     IN_Load;
    Exit     IN_Exit;
  event IN_Change_Root{
    if (!ee_command_help) return(1);
    send_token( PRESENTATION, HELP1, HELP1, "Change_Root: select command
            followed by args in any sequence" );
    send_token( PRESENTATION, HELP, HELP, "command selection valid for one
            execution only" );
    send_token( PRESENTATION, HELP, HELP, "args:position  - must be selected" );
  }

  event IN_Add_Limb{
    if (!ee_command_help) return(1);
    send_token( PRESENTATION, HELP1, HELP1, "Add_Limb: select command
            followed by args in any sequence" );
    send_token( PRESENTATION, HELP, HELP, "command remains active till
            another command is selected" );
    send_token( PRESENTATION, HELP, HELP, "args:limb  - must be selected" );
    send_token( PRESENTATION, HELP, HELP, "     mass length cofm bend rotate
            torque  - have current values" );
  }

  event IN_Remove_Limb{
    if (!ee_command_help) return(1);
```

```
  send_token( PRESENTATION, HELP1, HELP1, "Remove_Limb: select command
          followed by args in any sequence" );
  send_token( PRESENTATION, HELP, HELP, "command remains active till
          another command is selected" );
  send_token( PRESENTATION, HELP, HELP, "args:limb  - must be selected" );
}

event IN_Move_Limb{
  if (!ec_command_help) return(1);
  send_token( PRESENTATION, HELP1, HELP1, "Move_Limb: select command
          followed by args in any sequence" );
  send_token( PRESENTATION, HELP, HELP, "command remains active till
          another command is selected" );
  send_token( PRESENTATION, HELP, HELP, "args:limb new_parent  - must
          be selected" );
}

event IN_Show_Attributes{
  if (!ec_command_help) return(1);
  send_token( PRESENTATION, HELP1, HELP1, "Show_Attributes: select command
          followed by args in any sequence" );
  send_token( PRESENTATION, HELP, HELP, "command remains active till
          another command is selected" );
  send_token( PRESENTATION, HELP, HELP, "args:limb  - must be selected" );
}

event IN_Change_Orientation{
  if (!ec_command_help) return(1);
  send_token( PRESENTATION, HELP1, HELP1, "Change_Orientation: select
          command followed by args in any sequence" );
  send_token( PRESENTATION, HELP, HELP, "command remains active till
          another command is selected" );
  send_token( PRESENTATION, HELP, HELP, "args:limb new_bend new_rotation
          - must be selected" );
}

event IN_Change_Length{
  if (!ec_command_help) return(1);
  send_token( PRESENTATION, HELP1, HELP1, "Change_Length: select command
          followed by args in any sequence" );
  send_token( PRESENTATION, HELP, HELP, "command remains active till
          another command is selected" );
  send_token( PRESENTATION, HELP, HELP, "args:limb new_length  - must be
          selected" );
}

event IN_Change_Mass{
  if (!ec_command_help) return(1);
  send_token( PRESENTATION, HELP1, HELP1, "Change_Mass: select command
          followed by args in any sequence" );
```

```
  send_token( PRESENTATION, HELP, HELP, "command remains active till
          another command is selected" );
  send_token( PRESENTATION, HELP, HELP, "args:limb new_mass  - must be
          selected" );
}

event IN_Change_Cofm{
  if (!ee_command_help) return(1);
  send_token( PRESENTATION, HELP1, HELP1, "Change_Cofm: select command
          followed by args in any sequence" );
  send_token( PRESENTATION, HELP, HELP, "command remains active till
          another command is selected" );
  send_token( PRESENTATION, HELP, HELP, "args:limb new_cofm  - must be
          selected" );
}

event IN_Change_Torque{
  if (!ee_command_help) return(1);
  send_token( PRESENTATION, HELP1, HELP1, "Change_Torque: select command
          followed by args in any sequence" );
  send_token( PRESENTATION, HELP, HELP, "command remains active till
          another command is selected" );
  send_token( PRESENTATION, HELP, HELP, "args:limb new_torque  - must be
          selected" );
}

event IN_Save{
  if (!ee_command_help) return(1);
  send_token( PRESENTATION, HELP1, HELP1, "Save: select command followed
          by args in any sequence" );
  send_token( PRESENTATION, HELP, HELP, "command selection valid for one
          execution only" );
  send_token( PRESENTATION, HELP, HELP, "args:No args required." );
}

event IN_Load{
  if (!ee_command_help) return(1);
  send_token( PRESENTATION, HELP1, HELP1, "Load: select command followed
          by args in any sequence" );
  send_token( PRESENTATION, HELP, HELP, "command selection valid for one
          execution only" );
  send_token( PRESENTATION, HELP, HELP, "args:No args required." );
}

event IN_Exit{
  if (!ee_command_help) return(1);
  send_token( PRESENTATION, HELP1, HELP1, "Exit: select command followed
          by args in any sequence" );
  send_token( PRESENTATION, HELP, HELP, "command selection valid for one
          execution only" );
```

```
    send_token( PRESENTATION, HELP, HELP, "args:No args required." );
  }

end help;

eventhandler HOUSE_KEEPER is
  token
    Change_Root  IN_Change_Root;
    Add_Limb  IN_Add_Limb;
    Remove_Limb  IN_Remove_Limb;
    Move_Limb  IN_Move_Limb;
    Show_Attributes  IN_Show_Attributes;
    Change_Orientation  IN_Change_Orientation;
    Change_Length  IN_Change_Length;
    Change_Mass  IN_Change_Mass;
    Change_Cofm  IN_Change_Cofm;
    Change_Torque  IN_Change_Torque;
    Save  IN_Save;
    Load  IN_Load;
    Exit  IN_Exit;
    LENGTH  IN_LENGTH;
    MASS  IN_MASS;
    COFM  IN_COFM;
    BEND  IN_BEND;
    ROTATE  IN_ROTATE;
    TORQUE  IN_TORQUE;
    LIMB  IN_LIMB;
    INFO  IN_INFO;

  event INIT{
    cmd_Change_Root.status = OFF;
    cmd_Add_Limb.status = OFF;
    cmd_Remove_Limb.status = OFF;
    cmd_Move_Limb.status = OFF;
    cmd_Show_Attributes.status = OFF;
    cmd_Change_Orientation.status = OFF;
    cmd_Change_Length.status = OFF;
    cmd_Change_Mass.status = OFF;
    cmd_Change_Cofm.status = OFF;
    cmd_Change_Torque.status = OFF;
    cmd_Save.status = OFF;
    cmd_Load.status = OFF;
    cmd_Exit.status = OFF;
    Change_Root_position.status = UNDEF;
    Change_Root_position.value = NULL;
    Add_Limb_limb.status = UNDEF;
    Add_Limb_limb.value = NULL;
    Add_Limb_mass.status = UNDEF;
    Add_Limb_mass.value = NULL;
    Add_Limb_length.status = UNDEF;
```

```
Add_Limb_length.value = NULL;
Add_Limb_cofm.status = UNDEF;
Add_Limb_cofm.value = NULL;
Add_Limb_bend.status = UNDEF;
Add_Limb_bend.value = NULL;
Add_Limb_rotate.status = UNDEF;
Add_Limb_rotate.value = NULL;
Add_Limb_torque.status = UNDEF;
Add_Limb_torque.value = NULL;
Remove_Limb_limb.status = UNDEF;
Remove_Limb_limb.value = NULL;
Move_Limb_limb.status = UNDEF;
Move_Limb_limb.value = NULL;
Move_Limb_new_parent.status = UNDEF;
Move_Limb_new_parent.value = NULL;
Show_Attributes_limb.status = UNDEF;
Show_Attributes_limb.value = NULL;
Change_Orientation_limb.status = UNDEF;
Change_Orientation_limb.value = NULL;
Change_Orientation_new_bend.status = UNDEF;
Change_Orientation_new_bend.value = NULL;
Change_Orientation_new_rotation.status = UNDEF;
Change_Orientation_new_rotation.value = NULL;
Change_Length_limb.status = UNDEF;
Change_Length_limb.value = NULL;
Change_Length_new_length.status = UNDEF;
Change_Length_new_length.value = NULL;
Change_Mass_limb.status = UNDEF;
Change_Mass_limb.value = NULL;
Change_Mass_new_mass.status = UNDEF;
Change_Mass_new_mass.value = NULL;
Change_Cofm_limb.status = UNDEF;
Change_Cofm_limb.value = NULL;
Change_Cofm_new_cofm.status = UNDEF;
Change_Cofm_new_cofm.value = NULL;
Change_Torque_limb.status = UNDEF;
Change_Torque_limb.value = NULL;
Change_Torque_new_torque.status = UNDEF;
Change_Torque_new_torque.value = NULL;
dptr = (double *)malloc( sizeof( double ));
*dptr = 2.000000;
send_token( PRESENTATION, INITIAL, "LENGTH", dptr);
dptr = (double *)malloc( sizeof( double ));
*dptr = 10.000000;
send_token( PRESENTATION, INITIAL, "MASS", dptr);
dptr = (double *)malloc( sizeof( double ));
*dptr = 0.500000;
send_token( PRESENTATION, INITIAL, "COFM", dptr);
send_token( PRESENTATION, INITIAL, "BEND", 0);
send_token( PRESENTATION, INITIAL, "ROTATE", 0);
```

```
  send_token( PRESENTATION, INITIAL, "TORQUE", "TorqueFunc1");
  send_token( APPLICATION, INITIAL, INITIAL, NULL );
}
event IN_Change_Root{
  if (cmd_Change_Root.status == OFF){;
    deactivate_other_commands();
    cmd_Change_Root.status = ON;
    instantiate( cHANGE_rOOT );
  }
}
event IN_Add_Limb{
  if (cmd_Add_Limb.status == OFF){;
    deactivate_other_commands();
    cmd_Add_Limb.status = ON;
    instantiate( aDD_lIMB );
  }
}
event IN_Remove_Limb{
  if (cmd_Remove_Limb.status == OFF){;
    deactivate_other_commands();
    cmd_Remove_Limb.status = ON;
    instantiate( rEMOVE_lIMB );
  }
}
event IN_Move_Limb{
  if (cmd_Move_Limb.status == OFF){;
    deactivate_other_commands();
    cmd_Move_Limb.status = ON;
    instantiate( mOVE_lIMB );
  }
}
event IN_Show_Attributes{
  if (cmd_Show_Attributes.status == OFF){;
    deactivate_other_commands();
    cmd_Show_Attributes.status = ON;
    instantiate( sHOW_aTTRIBUTES );
  }
}
event IN_Change_Orientation{
  if (cmd_Change_Orientation.status == OFF){;
    deactivate_other_commands();
    cmd_Change_Orientation.status = ON;
    instantiate( cHANGE_oRIENTATION );
  }
}
event IN_Change_Length{
  if (cmd_Change_Length.status == OFF){;
    deactivate_other_commands();
    cmd_Change_Length.status = ON;
    instantiate( cHANGE_lENGTH );
```

```
  }
}
event IN_Change_Mass{
  if (cmd_Change_Mass.status == OFF){;
    deactivate_other_commands();
    cmd_Change_Mass.status = ON;
    instantiate( cHANGE_mASS );
  }
}
event IN_Change_Cofm{
  if (cmd_Change_Cofm.status == OFF){;
    deactivate_other_commands();
    cmd_Change_Cofm.status = ON;
    instantiate( cHANGE_cOFM );
  }
}
event IN_Change_Torque{
  if (cmd_Change_Torque.status == OFF){;
    deactivate_other_commands();
    cmd_Change_Torque.status = ON;
    instantiate( cHANGE_tORQUE );
  }
}
event IN_Save{
  if (cmd_Save.status == OFF){;
    deactivate_other_commands();
    cmd_Save.status = ON;
    instantiate( sAVE );
  }
}
event IN_Load{
  if (cmd_Load.status == OFF){;
    deactivate_other_commands();
    cmd_Load.status = ON;
    instantiate( lOAD );
  }
}
event IN_Exit{
  if (cmd_Exit.status == OFF){;
    deactivate_other_commands();
    cmd_Exit.status = ON;
    instantiate( eXIT );
  }
}
event IN_LIMB{
  Change_Root_position.status = DEF;
  Change_Root_position.value = event_value;
  Add_Limb_limb.status = DEF;
  Add_Limb_limb.value = event_value;
  Remove_Limb_limb.status = DEF;
```

```
    Remove_Limb_limb.value = event_value;
    if (Move_Limb_limb.status == UNDEF){
      Move_Limb_limb.status = DEF;
      Move_Limb_limb.value = event_value;
    }else{
      Move_Limb_new_parent.status = DEF;
      Move_Limb_new_parent.value = event_value;
    }
    Show_Attributes_limb.status = DEF;
    Show_Attributes_limb.value = event_value;
    Change_Orientation_limb.status = DEF;
    Change_Orientation_limb.value = event_value;
    Change_Length_limb.status = DEF;
    Change_Length_limb.value = event_value;
    Change_Mass_limb.status = DEF;
    Change_Mass_limb.value = event_value;
    Change_Cofm_limb.status = DEF;
    Change_Cofm_limb.value = event_value;
    Change_Torque_limb.status = DEF;
    Change_Torque_limb.value = event_value;
    send_token( DIALOGUE, INPUT, CHECK, NULL );
  }
  event IN_MASS{
    Add_Limb_mass.status = DEF;
    Add_Limb_mass.value = event_value;
    Change_Mass_new_mass.status = DEF;
    Change_Mass_new_mass.value = event_value;
    send_token( DIALOGUE, INPUT, CHECK, NULL );
  }
  event IN_LENGTH{
    Add_Limb_length.status = DEF;
    Add_Limb_length.value = event_value;
    Change_Length_new_length.status = DEF;
    Change_Length_new_length.value = event_value;
    send_token( DIALOGUE, INPUT, CHECK, NULL );
  }
  event IN_COFM{
    Add_Limb_cofm.status = DEF;
    Add_Limb_cofm.value = event_value;
    Change_Cofm_new_cofm.status = DEF;
    Change_Cofm_new_cofm.value = event_value;
    send_token( DIALOGUE, INPUT, CHECK, NULL );
  }
  event IN_BEND{
    Add_Limb_bend.status = DEF;
    Add_Limb_bend.value = event_value;
    Change_Orientation_new_bend.status = DEF;
    Change_Orientation_new_bend.value = event_value;
    send_token( DIALOGUE, INPUT, CHECK, NULL );
  }
```

```
  event IN_ROTATE{
    Add_Limb_rotate.status = DEF;
    Add_Limb_rotate.value = event_value;
    Change_Orientation_new_rotation.status = DEF;
    Change_Orientation_new_rotation.value = event_value;
    send_token( DIALOGUE, INPUT, CHECK, NULL );
  }
  event IN_TORQUE{
    Add_Limb_torque.status = DEF;
    Add_Limb_torque.value = event_value;
    Change_Torque_new_torque.status = DEF;
    Change_Torque_new_torque.value = event_value;
    send_token( DIALOGUE, INPUT, CHECK, NULL );
  }
end HOUSE_KEEPER;

eventhandler cHANGE_rOOT is  /* PREFIX CLOSE_ENDED */
  token     /* token declaration */
    CHECK  IN_CHECK;
  event INIT{
    Change_Root_position.status = UNDEF;
    break;
  }
  event IN_CHECK{
    if (Change_Root_position.status == DEF){
      values = (int *) calloc( 1, sizeof( int ));
      values[0] = Change_Root_position.value;
      send_token( APPLICATION, INPUT, Change_Root, values);
      send_token( PRESENTATION, INITIAL, "cmenu", "-1");
      cmd_Change_Root.status = OFF;
      destroy_instance( self_id );
    }
  }
end cHANGE_rOOT;

eventhandler aDD_lIMB is  /* PREFIX OPEN_ENDED */
  token     /* token declaration */
    CHECK  IN_CHECK;
  event INIT{
    Add_Limb_limb.status = UNDEF;
    break;
  }
  event IN_CHECK{
    if (Add_Limb_limb.status == DEF && Add_Limb_mass.status== DEF &&
       Add_Limb_length.status== DEF && Add_Limb_cofm.status== DEF &&
       Add_Limb_bend.status== DEF && Add_Limb_rotate.status== DEF &&
       Add_Limb_torque.status== DEF){
      values = (int *) calloc( 7, sizeof( int ));
      values[0] = Add_Limb_limb.value;
      values[1] = Add_Limb_mass.value;
```

```
      values[2] = Add_Limb_length.value;
      values[3] = Add_Limb_cofm.value;
      values[4] = Add_Limb_bend.value;
      values[5] = Add_Limb_rotate.value;
      values[6] = Add_Limb_torque.value;
      send_token( APPLICATION, INPUT, Add_Limb, values);
    Add_Limb_limb.status = UNDEF;
    }
  }
end aDD_lIMB;

eventhandler rEMOVE_lIMB is   /* PREFIX OPEN_ENDED */
  token     /* token declaration */
    CHECK  IN_CHECK;
  event INIT{
    Remove_Limb_limb.status = UNDEF;
    break;
  }
  event IN_CHECK{
    if (Remove_Limb_limb.status == DEF){
      values = (int *) calloc( 1, sizeof( int ));
      values[0] = Remove_Limb_limb.value;
      send_token( APPLICATION, INPUT, Remove_Limb, values);
    Remove_Limb_limb.status = UNDEF;
    }
  }
end rEMOVE_lIMB;

eventhandler mOVE_lIMB is   /* PREFIX OPEN_ENDED */
  token     /* token declaration */
    CHECK  IN_CHECK;
  event INIT{
    Move_Limb_limb.status = UNDEF;
    Move_Limb_new_parent.status = UNDEF;
    break;
  }
  event IN_CHECK{
    if (Move_Limb_limb.status == DEF && Move_Limb_new_parent.status== DEF){
      values = (int *) calloc( 2, sizeof( int ));
      values[0] = Move_Limb_limb.value;
      values[1] = Move_Limb_new_parent.value;
      send_token( APPLICATION, INPUT, Move_Limb, values);
    Move_Limb_limb.status = UNDEF;
    Move_Limb_new_parent.status = UNDEF;
    }
  }
end mOVE_lIMB;

eventhandler sHOW_aTTRIBUTES is   /* PREFIX OPEN_ENDED */
  token     /* token declaration */
```

```
    CHECK  IN_CHECK;
  event INIT{
    Show_Attributes_limb.status = UNDEF;
    break;
  }
  event IN_CHECK{
    if (Show_Attributes_limb.status == DEF){
      values = (int *) calloc( 1, sizeof( int ));
      values[0] = Show_Attributes_limb.value;
      send_token( APPLICATION, INPUT, Show_Attributes, values);
    Show_Attributes_limb.status = UNDEF;
    }
  }
end sHOW_aTTRIBUTES;

eventhandler cHANGE_oRIENTATION is   /* PREFIX OPEN_ENDED */
  token      /* token declaration */
    CHECK  IN_CHECK;
  event INIT{
    Change_Orientation_limb.status = UNDEF;
    Change_Orientation_new_bend.status = UNDEF;
    Change_Orientation_new_rotation.status = UNDEF;
    break;
  }
  event IN_CHECK{
    if (Change_Orientation_limb.status == DEF &&
       Change_Orientation_new_bend.status== DEF &&
       Change_Orientation_new_rotation.status== DEF){
      values = (int *) calloc( 3, sizeof( int ));
      values[0] = Change_Orientation_limb.value;
      values[1] = Change_Orientation_new_bend.value;
      values[2] = Change_Orientation_new_rotation.value;
      send_token( APPLICATION, INPUT, Change_Orientation, values);
    Change_Orientation_limb.status = UNDEF;
    Change_Orientation_new_bend.status = UNDEF;
    Change_Orientation_new_rotation.status = UNDEF;
    }
  }
end cHANGE_oRIENTATION;

eventhandler cHANGE_lENGTH is   /* PREFIX OPEN_ENDED */
  token      /* token declaration */
    CHECK  IN_CHECK;
  event INIT{
    Change_Length_limb.status = UNDEF;
    Change_Length_new_length.status = UNDEF;
    break;
  }
  event IN_CHECK{
    if (Change_Length_limb.status == DEF &&
```

```
      Change_Length_new_length.status== DEF){
      values = (int *) calloc( 2, sizeof( int ));
      values[0] = Change_Length_limb.value;
      values[1] = Change_Length_new_length.value;
      send_token( APPLICATION, INPUT, Change_Length, values);
    Change_Length_limb.status = UNDEF;
    Change_Length_new_length.status = UNDEF;
    }
  }
end cHANGE_lENGTH;

eventhandler cHANGE_mASS is   /* PREFIX OPEN_ENDED */
  token      /* token declaration */
    CHECK  IN_CHECK;
  event INIT{
    Change_Mass_limb.status = UNDEF;
    Change_Mass_new_mass.status = UNDEF;
    break;
  }
  event IN_CHECK{
    if (Change_Mass_limb.status == DEF && Change_Mass_new_mass.status== DEF){
      values = (int *) calloc( 2, sizeof( int ));
      values[0] = Change_Mass_limb.value;
      values[1] = Change_Mass_new_mass.value;
      send_token( APPLICATION, INPUT, Change_Mass, values);
    Change_Mass_limb.status = UNDEF;
    Change_Mass_new_mass.status = UNDEF;
    }
  }
end cHANGE_mASS;

eventhandler cHANGE_cOFM is   /* PREFIX OPEN_ENDED */
  token      /* token declaration */
    CHECK  IN_CHECK;
  event INIT{
    Change_Cofm_limb.status = UNDEF;
    Change_Cofm_new_cofm.status = UNDEF;
    break;
  }
  event IN_CHECK{
    if (Change_Cofm_limb.status == DEF && Change_Cofm_new_cofm.status== DEF){
      values = (int *) calloc( 2, sizeof( int ));
      values[0] = Change_Cofm_limb.value;
      values[1] = Change_Cofm_new_cofm.value;
      send_token( APPLICATION, INPUT, Change_Cofm, values);
    Change_Cofm_limb.status = UNDEF;
    Change_Cofm_new_cofm.status = UNDEF;
    }
  }
end cHANGE_cOFM;
```

```
eventhandler cHANGE_tORQUE is  /* PREFIX OPEN_ENDED */
  token    /* token declaration */
    CHECK  IN_CHECK;
  event INIT{
    Change_Torque_limb.status = UNDEF;
    Change_Torque_new_torque.status = UNDEF;
    break;
  }
  event IN_CHECK{
    if (Change_Torque_limb.status == DEF &&
       Change_Torque_new_torque.status== DEF){
      values = (int *) calloc( 2, sizeof( int ));
      values[0] = Change_Torque_limb.value;
      values[1] = Change_Torque_new_torque.value;
      send_token( APPLICATION, INPUT, Change_Torque, values);
     Change_Torque_limb.status = UNDEF;
     Change_Torque_new_torque.status = UNDEF;
     }
  }
end cHANGE_tORQUE;

eventhandler sAVE is  /* PREFIX CLOSE_ENDED */
  event INIT{
    send_token( APPLICATION, INPUT, Save, NULL);
    send_token( PRESENTATION, INITIAL, "cmenu", "-1");
    cmd_Save.status = OFF;
    destroy_instance( self_id );
    break;
  }
end sAVE;

eventhandler lOAD is  /* PREFIX CLOSE_ENDED */
  event INIT{
    send_token( APPLICATION, INPUT, Load, NULL);
    send_token( PRESENTATION, INITIAL, "cmenu", "-1");
    cmd_Load.status = OFF;
    destroy_instance( self_id );
    break;
  }
end lOAD;

eventhandler eXIT is  /* PREFIX CLOSE_ENDED */
  event INIT{
    send_token( APPLICATION, INPUT, Exit, NULL);
    send_token( PRESENTATION, INITIAL, "cmenu", "-1");
    cmd_Exit.status = OFF;
    destroy_instance( self_id );
    break;
  }
end eXIT;
```

Appendix A5

Event Handlers for the Example Dialogue of Figure 4.7

This appendix contains the event handlers produced by Diction for the example dialogue shown in figure 4.7.

```
eventhandler help is
  token
    Add_Object_pr      IN_Add_Object_pr;
    Add_Object_ps      IN_Add_Object_ps;
    Add_Object_nf      IN_Add_Object_nf;
  event IN_Add_Object_pr{
    if (!ee_command_help) return(1);
    send_token( PRESENTATION, HELP1, HELP1, "Add_Object_pr: select command
                        followed by args in any sequence" );
    send_token( PRESENTATION, HELP, HELP, "command remains active till
                          another command is selected" );
    send_token( PRESENTATION, HELP, HELP, "args:where  - must be selected" );
    send_token( PRESENTATION, HELP, HELP, "    colour1  - have defaults" );
    send_token( PRESENTATION, HELP, HELP, "    size  - have current values" );
  }

  event IN_Add_Object_ps{
    if (!ee_command_help) return(1);
    send_token( PRESENTATION, HELP1, HELP1, "Add_Object_ps: select args
                             followed by command" );
    send_token( PRESENTATION, HELP, HELP, "command selection valid for
                             one execution only" );
    send_token( PRESENTATION, HELP, HELP, "args:where  - must be selected" );
    send_token( PRESENTATION, HELP, HELP, "    colour2  - have defaults" );
    send_token( PRESENTATION, HELP, HELP, "    size  - have current values" );
  }

  event IN_Add_Object_nf{
    if (!ee_command_help) return(1);
    send_token( PRESENTATION, HELP1, HELP1, "Add_Object_nf: select command
                           and args in any sequence" );
    send_token( PRESENTATION, HELP, HELP, "command remains active till
                          another command is selected" );
    send_token( PRESENTATION, HELP, HELP, "args:where  - must be selected" );
    send_token( PRESENTATION, HELP, HELP, "    colour3  - have defaults" );
    send_token( PRESENTATION, HELP, HELP, "    size  - have current values" );
  }
```

```
end help;

eventhandler HOUSE_KEEPER is
  token
    Add_Object_pr  IN_Add_Object_pr;
    Add_Object_ps  IN_Add_Object_ps;
    Add_Object_nf  IN_Add_Object_nf;
    SIZE  IN_SIZE;
    WORK  IN_WORK;
    colour1  IN_colour1;
    colour2  IN_colour2;
    colour3  IN_colour3;

  event INIT{
    cmd_Add_Object_pr.status = OFF;
    cmd_Add_Object_ps.status = OFF;
    cmd_Add_Object_nf.status = OFF;
    Add_Object_pr_where.status = UNDEF;
    Add_Object_pr_where.value = NULL;
    Add_Object_pr_size.status = UNDEF;
    Add_Object_pr_size.value = NULL;
    Add_Object_pr_colour1.status = UNDEF;
    Add_Object_pr_colour1.value = NULL;
    Add_Object_ps_where.status = UNDEF;
    Add_Object_ps_where.value = NULL;
    Add_Object_ps_size.status = UNDEF;
    Add_Object_ps_size.value = NULL;
    Add_Object_ps_colour2.status = UNDEF;
    Add_Object_ps_colour2.value = NULL;
    Add_Object_nf_where.status = UNDEF;
    Add_Object_nf_where.value = NULL;
    Add_Object_nf_size.status = UNDEF;
    Add_Object_nf_size.value = NULL;
    Add_Object_nf_colour3.status = UNDEF;
    Add_Object_nf_colour3.value = NULL;
    dptr = (double *)malloc( sizeof( double ));
    *dptr = 0.500000;
    send_token( PRESENTATION, INITIAL, "SIZE", dptr);
    send_token( APPLICATION, INITIAL, INITIAL, NULL );
  }
  event IN_Add_Object_pr{
    if (cmd_Add_Object_pr.status == OFF){
      deactivate_other_commands();
      cmd_Add_Object_pr.status = ON;
      instantiate( aDD_oBJECT_PR );
    }
  }
  event IN_Add_Object_ps{
    if (cmd_Add_Object_ps.status == OFF){
      deactivate_other_commands();
```

```
      cmd_Add_Object_ps.status = ON;
      instantiate( aDD_oBJECT_PS );
    }
  }
  event IN_Add_Object_nf{
    if (cmd_Add_Object_nf.status == OFF){
      deactivate_other_commands();
      cmd_Add_Object_nf.status = ON;
      instantiate( aDD_oBJECT_NF );
    }
  }
  event IN_WORK{
    Add_Object_pr_where.status = DEF;
    Add_Object_pr_where.value = event_value;
    Add_Object_ps_where.status = DEF;
    Add_Object_ps_where.value = event_value;
    Add_Object_nf_where.status = DEF;
    Add_Object_nf_where.value = event_value;
    send_token( DIALOGUE, INPUT, CHECK, NULL );
  }
  event IN_SIZE{
    Add_Object_pr_size.status = DEF;
    Add_Object_pr_size.value = event_value;
    Add_Object_ps_size.status = DEF;
    Add_Object_ps_size.value = event_value;
    Add_Object_nf_size.status = DEF;
    Add_Object_nf_size.value = event_value;
    send_token( DIALOGUE, INPUT, CHECK, NULL );
  }
  event IN_colour1{
    Add_Object_pr_colour1.status = DEF;
    Add_Object_pr_colour1.value = event_value;
    send_token( DIALOGUE, INPUT, CHECK, NULL );
  }
  event IN_colour2{
    Add_Object_ps_colour2.status = DEF;
    Add_Object_ps_colour2.value = event_value;
    send_token( DIALOGUE, INPUT, CHECK, NULL );
  }
  event IN_colour3{
    Add_Object_nf_colour3.status = DEF;
    Add_Object_nf_colour3.value = event_value;
    send_token( DIALOGUE, INPUT, CHECK, NULL );
  }
end HOUSE_KEEPER;

eventhandler aDD_oBJECT_PR is  /* PREFIX OPEN_ENDED */
  token    /* token declaration */
    CHECK  IN_CHECK;
  event INIT{
```

```
    Add_Object_pr_where.status = UNDEF;
    send_token( PRESENTATION, INITIAL, "colour1", "RED");
    break;
  }
  event IN_CHECK{
    if (Add_Object_pr_where.status == DEF && Add_Object_pr_size.status== DEF
        && Add_Object_pr_colour1.status== DEF){
      values = (int *) calloc( 3, sizeof( int ));
      values[0] = Add_Object_pr_where.value;
      values[1] = Add_Object_pr_size.value;
      values[2] = Add_Object_pr_colour1.value;
      send_token( APPLICATION, INPUT, Add_Object_pr, values);
      Add_Object_ps_where.status = UNDEF;
      Add_Object_nf_where.status = UNDEF;
      Add_Object_pr_where.status = UNDEF;
      send_token( PRESENTATION, INITIAL, "colour1", "RED");
    }
  }
end aDD_oBJECT_PR;

eventhandler aDD_oBJECT_PS is   /* POSTFIX CLOSE_ENDED */
  token     /* token declaration */
    CHECK  IN_CHECK;
  var
    int lflag = 0;

  event INIT{
    if (Add_Object_ps_colour2.status == UNDEF){
      send_token( PRESENTATION, INITIAL, "colour2", "RED");
      lflag = 1;
    }
    if (!lflag)
      if (Add_Object_ps_where.status == DEF && Add_Object_ps_size.status == DEF
          && Add_Object_ps_colour2.status == DEF){
      values = (int *) calloc( 3, sizeof( int ));
      values[0] = Add_Object_ps_where.value;
      values[1] = Add_Object_ps_size.value;
      values[2] = Add_Object_ps_colour2.value;
      send_token( APPLICATION, INPUT, Add_Object_ps, values);
      Add_Object_ps_where.status = UNDEF;
      Add_Object_nf_where.status = UNDEF;
      } else
        send_token( PRESENTATION, ERROR, ERROR, "arg selection incomplete, reselect
                command after complete arg selection" );
        send_token( PRESENTATION, INITIAL, "cmenu", "-1");
        cmd_Add_Object_ps.status = OFF;
        destroy_instance( self_id );
  }
  event IN_CHECK{
      if (Add_Object_ps_where.status == DEF && Add_Object_ps_size.status == DEF &&
```

```
            Add_Object_ps_colour2.status == DEF){
        values = (int *) calloc( 3, sizeof( int ));
        values[0] = Add_Object_ps_where.value;
        values[1] = Add_Object_ps_size.value;
        values[2] = Add_Object_ps_colour2.value;
        send_token( APPLICATION, INPUT, Add_Object_ps, values);
        Add_Object_ps_where.status = UNDEF;
        Add_Object_nf_where.status = UNDEF;
        send_token( PRESENTATION, INITIAL, "cmenu", "-1");
        cmd_Add_Object_ps.status = OFF;
        destroy_instance( self_id );
      }
    }
end aDD_oBJECT_PS;

eventhandler aDD_oBJECT_NF is   /* NOFIX OPEN_ENDED */
  token     /* token declaration */
    CHECK  IN_CHECK;
  event INIT{
    if (Add_Object_nf_colour3.status == UNDEF)
      send_token( PRESENTATION, INITIAL, "colour3", "RED");
    if (Add_Object_nf_where.status == DEF && Add_Object_nf_size.status == DEF
        && Add_Object_nf_colour3.status == DEF){
      values = (int *) calloc( 3, sizeof( int ));
      values[0] = Add_Object_nf_where.value;
      values[1] = Add_Object_nf_size.value;
      values[2] = Add_Object_nf_colour3.value;
      send_token( APPLICATION, INPUT, Add_Object_nf, values);
      Add_Object_ps_where.status = UNDEF;
      Add_Object_nf_where.status = UNDEF;
      Add_Object_nf_where.status = UNDEF;
      send_token( PRESENTATION, INITIAL, "colour3", "RED");
    }
  }
  event IN_CHECK{
    if (Add_Object_nf_where.status == DEF && Add_Object_nf_size.status == DEF
        && Add_Object_nf_colour3.status == DEF){
      values = (int *) calloc( 3, sizeof( int ));
      values[0] = Add_Object_nf_where.value;
      values[1] = Add_Object_nf_size.value;
      values[2] = Add_Object_nf_colour3.value;
      send_token( APPLICATION, INPUT, Add_Object_nf, values);
      Add_Object_ps_where.status = UNDEF;
      Add_Object_nf_where.status = UNDEF;
      Add_Object_nf_where.status = UNDEF;
      send_token( PRESENTATION, INITIAL, "colour3", "RED");
    }
  }
end aDD_oBJECT_NF;
```

Appendix A6

Rules Used in Chisel

This appendix summarizes the functioning of Chisel in terms of steps or rules. As explained in Chapter 5, there are three main steps followed by Chisel: selecting interaction techniques, determining attribute values for the selected interaction techniques, and placing interaction techniques on the display screen. Each of these steps is summarized in the following sections. The content of this appendix may not be clearly understood unless the reader has read chapter 5.

1. Selecting Interaction Techniques

Chisel has three main concerns while deciding on interaction techniques. These concerns relate to the type and range (or enumerations) of the command argument, user's preferences, and the device requirements of interaction techniques. Each of these concerns limits the set of interaction techniques that can be used to enter the argument values. The following table shows the selection of interaction techniques in the absence of user's preferences and assuming that device requirements of the selected interaction techniques are satisfied. In the presence of user's preferences or when the device requirements of the following interaction techniques cannot be met, other interaction techniques may be selected.

Table A6.1 Selecting Interaction Techniques

Type	Interaction Technique
Command	Horizontal and Vertical Static Menus
Enumerated	Horizontal and Vertical Static Menus
Sub-Range - Real/Integer	Select Real/Integer interaction techniques which match the range exactly. If none matches the range, select Real/Integer Horizontal and Vertical graphical potentiometers.
Real/Integer - No Range	Select Real/Integer interaction techniques which do not have range restrictions.
Text	Select text windows in page mode.
pick2d/pick3d	Select techniques which implement 2d/3d picks.
Window	Select graphical window.

1. Determining Object Attributes

This section describes the computation of size, location, and colour of interaction techniques.

2. Determining Size

Type	Computation
Fixed Sizes	Copy the size from prototype to instance.
Menus	Size depends on width of the biggest item in the menu, number of menu items, maximum number of items to be placed in a menu, height of a menu item, and the type of menu (overlay/non-overlay).
Unknown Sizes - but computable	Invoke the size function specified in the prototype.
Unknown Sizes	Leave the size computation for the last step.

3. Generating Positional Constraints

Command menus constrain their position depending upon the user's preferences. If the prototype of an interaction specifies a positional constraint, Chisel duplicates the constraint in the instance of the interaction technique. Otherwise, for objects with bigger height than width, Chisel generates constraints to place them along vertical edges, and for objects with bigger width than height, Chisel generates constraints to place them along horizontal edges. For objects with unknown sizes, Chisel generates constraints to place them anywhere on the screen. Objects which violate the general or the designer's strategy for placing

objects are deleted by Chisel. If at this stage, an argument is represented by more than one interaction techniques, Chisel selects one interaction technique randomly.

4. Determining Colour

All objects are assigned a default drawing and a default background colour. The defaults can be changed by the designer. User's preferred colours can be assigned to objects.

1. Placing Interaction Techniques

Chisel assigns a measure of specificity to all objects based on their positional constraint. The following table shows this measure for various constraints.

Table A6.2 Measures of Specificity

Constraint	Measure of Specificity
A particular Corner	100
Left or Right Corner	95
Top or Bottom Corner	95
Any Corner	90
A particular Edge	80
Vertical Edge	75
Horizontal Edge	75
Any Edge	70
A particular Quadrant	60
A particular Half	50
Top or Bottom Half	45
Left or right Half	45
Anywhere	0

The objects are then sorted in the decreasing order of their measure of specificity. Objects with higher measure are placed first. If a positional constraint cannot be satisfied, Chisel relaxes it to the next lower level constraint, and tries to satisfy the new constraint. After all objects with known sizes are placed, Chisel computes free space on the screen and divides the free space equally amongst all objects with unknown sizes.