An enhanced user interface management system

This item was submitted to Loughborough University's Institutional Repository by the/an author.

Additional Information:


Metadata Record: [https://dspace.lboro.ac.uk/2134/27970](https://dspace.lboro.ac.uk/2134/27970)

Publisher: © Haishan Huang

Rights: This work is made available according to the conditions of the Creative Commons Attribution-NonCommercial-NoDerivatives 2.5 Generic (CC BY-NC-ND 2.5) licence. Full details of this licence are available at: [http://creativecommons.org/licenses/by-nc-nd/2.5/](http://creativecommons.org/licenses/by-nc-nd/2.5/)

Please cite the published version.
This item was submitted to Loughborough University as a PhD thesis by the author and is made available in the Institutional Repository (https://dspace.lboro.ac.uk/) under the following Creative Commons Licence conditions.

For the full text of this licence, please go to: http://creativecommons.org/licenses/by-nc-nd/2.5/
AN ENHANCED USER INTERFACE MANAGEMENT SYSTEM

by

HAISHAN HUANG

A Doctoral Thesis

Submitted in partial fulfilment of the requirements
for the award of

the Degree of Doctor of Philosophy of the Loughborough University

September 1997

Supervisor: Professor E. A. Edmonds

Department of Computer Studies
Loughborough University
Loughborough
England, U.K.

©by Haishan Huang 1997
DEDICATION

To my wife Qiuhui and my parents.
ABSTRACT

A User Interface Management System (UIMS) called the Harness has been developed to support the front-ending of existing software systems. In the Harness, a set of core Abstract Interaction Objects (AIOs) was provided for knowledge-based modules to interact with the user. It is not realistic to expect the Harness to be a closed and fixed system. On the other hand, help capabilities are today considered essential components of well designed software and on-line help systems have come to be a standard feature of most new software systems. The work described in this thesis is an attempt to enhance some capabilities and flexibilities to the Harness UIMS in two aspects: 1) providing the designer with an AIO editor to make the Harness UIMS extensible in a graphical, direct manipulation manner without becoming involved in the programming and implementation details of the existing UIMS; 2) separating the domain/application independent part of help systems from applications to provide the designer with a general help manager to make the context-sensitive help possible without having help resided in an application. To achieve these extensions, three modules - AIO Editor, AIO Knowledge Base and Help Manager - have been extended into the FOCUS KBFE (Knowledge-Based Front-End) architecture. An AIO editor which is used to introduce new AIOs into the Harness and to maintain AIO knowledge bases, the relevant components which deal with the new AIO management, and a general help manager which manages the help information have been designed, implemented and presented in great detail in this thesis. Finally, the GRASS, which is a public-domain Geographical Information System (GIS) with a poor command line user interface, has been chosen to test the enhanced UIMS and tools. A prototype system, which is a KBFE to the GRASS, has been developed to demonstrate that 1) the approaches are feasible to extend the capabilities and flexibilities of an existing UIMS; and 2) the enhanced UIMS, with the support of the AIO editor and the help manager, provides the developer with great flexibility and the freedom to develop more sophisticated KBFE system rapidly.
ACKNOWLEDGEMENTS

I would like to express my appreciation to my supervisor, Professor Ernest Edmonds, for his guidance and encouragement throughout the course of this research and for the financial support in the first two years; and to Dr. H. Bez who was my director of research.

I would like to thank everyone in the FOCUS and MUMS projects who contributed the facilities and useful discussion. Particularly, I wish to thank Linda Candy, Jenia Ghazikhanian and Rachel Jones who gave me much help when I was working in these projects at the LUTCNI Research Centre.

I would also like to thank my friend Mr. Robert Kerr for reading and checking the grammar on the final draft of this thesis.

Finally I am greatly indebted to my wife Qiuhui and my daughter Anne for their understanding and encouragement during my thesis writing up.

The work described in this thesis partly funded by the ESPRIT FOCUS project (2620) and the IED4/1/1256 MUMS project.
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACM</td>
<td>Application Communications Manager</td>
</tr>
<tr>
<td>AIO</td>
<td>Abstract Interaction Object</td>
</tr>
<tr>
<td>ALGAE</td>
<td>A Language for Generating Asynchronous Event handlers</td>
</tr>
<tr>
<td>ATN</td>
<td>Augmented Transition Network</td>
</tr>
<tr>
<td>BEM</td>
<td>Back-End Manager</td>
</tr>
<tr>
<td>BNF</td>
<td>Backus-Naur Form</td>
</tr>
<tr>
<td>CKB</td>
<td>Configuration Knowledge Base</td>
</tr>
<tr>
<td>CSP</td>
<td>Communicating Sequential Processes</td>
</tr>
<tr>
<td>DDL</td>
<td>Dialogue Definition Language</td>
</tr>
<tr>
<td>DDM</td>
<td>Dynamic Dialogue Manager</td>
</tr>
<tr>
<td>DPM</td>
<td>Dynamic Presentation Manager</td>
</tr>
<tr>
<td>Druid</td>
<td>Demonstrational Rapid User Interface Development</td>
</tr>
<tr>
<td>ERA</td>
<td>Entity-Relationship-Attribute</td>
</tr>
<tr>
<td>ERL</td>
<td>Event Response Language</td>
</tr>
<tr>
<td>ERS</td>
<td>Event Response System</td>
</tr>
<tr>
<td>FCG</td>
<td>Function Chaining Graph</td>
</tr>
<tr>
<td>FOCUS</td>
<td>Front Ends for Open and Closed User Systems</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>Garnet</td>
<td>Generating an Amalgam of Real-time, Novel Editors and Toolkits</td>
</tr>
<tr>
<td>GENIUS</td>
<td>GENerator for user Interfaces Using Software ergonomic rules</td>
</tr>
<tr>
<td>Gilt</td>
<td>Garnet Interface Layout Tool</td>
</tr>
<tr>
<td>GIS</td>
<td>Geographical Information System</td>
</tr>
<tr>
<td>GRASS</td>
<td>Geographical Resources Analysis Support System</td>
</tr>
<tr>
<td>GTN</td>
<td>Generative Transition Network</td>
</tr>
<tr>
<td>Guide</td>
<td>Graphic User Interface Design Editor</td>
</tr>
<tr>
<td>HCM</td>
<td>Harness Communications Manager</td>
</tr>
<tr>
<td>HUMANOID</td>
<td>High-level UIMS for Manufacturing Applications Needing</td>
</tr>
</tbody>
</table>
Organized Iterative Development

ITS
Interactive Transaction System

Jade
Judgement-based Automatic Dialog Editor

KBFE
Knowledge-Based Front-End

KBM
Knowledge-Based Module

KMS
Knowledge Management System

Lapidary
Lisp-based Assistant for Prototyping Interface Designs Allowing Remarkable Yield

Marquise
Mostly Automated, Remarkably Quick User Interface Software Environment

MIKE
Menu Interaction Kontrol Environment

MVC
Model-View-Controller

OOP
Object-Oriented Programming

OSF
Open Software Foundation

OSU
Oregon Speedcode Universe

PAC
Presentation-Abstraction-Control

Peridot
Programming by Example for Real-time Interface Design Obviating Typing

PKB
Presentation Knowledge Base

PPL
Physical Presentation Layer

ResDez
Resource Designer

RTN
Recursive Transition Network

SDK
Software Development Kit

SPI
Specifying and Prototyping Interaction

STN
State Transition Network

SUIT
Simple User Interface Toolkits

SYNGRAPHSYNtax directed GRAPHics

SYNICS
SYNtax and semantiCS

TDE
Transition Diagram Editor
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDI</td>
<td>Transition Diagram Interpreter</td>
</tr>
<tr>
<td>TICCL</td>
<td>Tiger Interactive Command and Control Language</td>
</tr>
<tr>
<td>TIGER</td>
<td>The Interactive Graphic Engineering Resource</td>
</tr>
<tr>
<td>TPL</td>
<td>TemPlate Language</td>
</tr>
<tr>
<td>UIMS</td>
<td>User Interface Management System</td>
</tr>
<tr>
<td>UIDE</td>
<td>User Interface Design Environment</td>
</tr>
<tr>
<td>UIDS</td>
<td>User Interface Development System</td>
</tr>
<tr>
<td>UIT</td>
<td>User Interface Toolkits</td>
</tr>
<tr>
<td>Xtk</td>
<td>X-window Toolkit</td>
</tr>
<tr>
<td>XTL</td>
<td>eXTraction Language</td>
</tr>
<tr>
<td>XVT</td>
<td>eXtensible Virtual Toolkit</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>i</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>ii</td>
</tr>
<tr>
<td>Abbreviations</td>
<td>iii</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>vi</td>
</tr>
<tr>
<td><strong>Chapter 1</strong> Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Objectives of the Research</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Overview of the Thesis</td>
<td>4</td>
</tr>
<tr>
<td><strong>Chapter 2</strong> Focus Architecture</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>6</td>
</tr>
<tr>
<td>2.2 User Interface Management Systems</td>
<td>6</td>
</tr>
<tr>
<td>2.2.1 UIMS Concepts</td>
<td>7</td>
</tr>
<tr>
<td>2.2.2 The Seeheim Model</td>
<td>8</td>
</tr>
<tr>
<td>2.2.3 Post-Seeheim Models</td>
<td>10</td>
</tr>
<tr>
<td>2.3 FOCUS KBFE Architecture</td>
<td>15</td>
</tr>
<tr>
<td>2.3.1 The Harness</td>
<td>17</td>
</tr>
<tr>
<td>2.3.2 Knowledge-Based Modules</td>
<td>17</td>
</tr>
<tr>
<td>2.3.3 The Back-End Manager</td>
<td>18</td>
</tr>
<tr>
<td>2.4 Harness Architecture</td>
<td>21</td>
</tr>
<tr>
<td>2.4.1 Communications Manager</td>
<td>22</td>
</tr>
<tr>
<td>2.4.2 Dynamic Dialogue Manager</td>
<td>24</td>
</tr>
<tr>
<td>2.4.3 Presentation</td>
<td>24</td>
</tr>
<tr>
<td>2.5 Messages, AIOs and User Interface</td>
<td>27</td>
</tr>
<tr>
<td>2.5.1 Messages</td>
<td>27</td>
</tr>
<tr>
<td>2.5.2 AIOs and User Interface</td>
<td>28</td>
</tr>
<tr>
<td>2.6 Summary</td>
<td>30</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

## CHAPTER 3  INVESTIGATIVE STUDY OF USER INTERFACE TOOLS  33

3.1 Introduction ............................................................................................ 33  
3.2 User Interface Toolkit Approaches ........................................................ 34  
  3.2.1 Toolkits ...................................................................................... 34  
  3.2.2 Virtual Toolkits .......................................................................... 37  
  3.2.3 Disadvantages with the Toolkit Approaches ............................. 39  
3.3 Language-Based User Interface Specification .................................... 40  
  3.3.1 Menu Networks ........................................................................ 41  
  3.3.2 State Transition Networks .......................................................... 43  
  3.3.3 Context-Free Grammars ............................................................. 49  
  3.3.4 Event-Based Systems ................................................................. 52  
  3.3.5 Declarative Approaches ............................................................. 56  
  3.3.6 Object-Oriented Approaches ...................................................... 56  
3.4 Direct Graphical User Interface Specification ....................................... 58  
  3.4.1 Creating User Interface by Direct Manipulation ................ 58  
  3.4.2 Creating User Interface by Demonstration ................................ 60  
  3.4.3 Frame/Card-Based Systems ....................................................... 64  
3.5 Automated User Interface Generation ................................................... 65  
3.6 Model-Based User Interface Development Systems ..................... 67  
3.7 Summary ................................................................................................ 70

## CHAPTER 4  THE ABSTRACT INTERACTION OBJECT EDITOR  73

4.1 Introduction ............................................................................................ 73  
4.2 Programming the Harness ...................................................................... 74  
4.3 The Architecture of the AIO Editor ....................................................... 77  
4.4 Editing the Presentation of Interaction Objects ..................................... 78  
  4.4.1 Creating General Interaction objects ......................................... 80  
  4.4.2 Creating Variety Menus ............................................................ 85  
  4.4.3 Creating Complex Interaction Objects ...................................... 88  
4.5 Editing the Behaviour of Interaction Objects ........................................ 89  
4.6 Simulation Environment ........................................................................ 92  
4.7 The AIO Knowledge Base ..................................................................... 92  
4.8 Comparison with Other Similar Tools ............................................... 96  
4.9 Conclusions ............................................................................................ 99

## CHAPTER 5  IMPLEMENTATION OF THE NEW AIO MANAGEMENT  100

5.1 Introduction ............................................................................................ 100  
5.2 Analysis of the Implementation of the Presentation Layer ................ 100  
  5.2.1 BNF Specifications of Core AIOs .............................................. 101
TABLE OF CONTENTS

5.2.2 Data Structures of Core AIOs ..................................................... 103
5.2.3 Syntactic and Lexical Analysis .................................................. 104
5.2.4 Core AIO Management in the Presentation Layer ..................... 105
5.3 Implementation of New AIOs ....................................................... 106
  5.3.1 BNF Specification of New AIOs .............................................. 107
  5.3.2 Data Structures of New AIOs .................................................. 109
  5.3.3 New AIO Management in the Presentation Layer ..................... 110
5.4 Summary ....................................................................................... 114

CHAPTER 6 THE GENERAL HELP SYSTEM MANAGER ..................... 116
  6.1 Introduction ................................................................................... 116
  6.2 Background: Development, Design and Implementation ............... 117
  6.3 The Help System Manager ........................................................... 121
  6.4 Building Context-Sensitive Help .................................................. 124
  6.5 Using Hypertext .......................................................................... 126
  6.6 Conclusions .................................................................................. 127

CHAPTER 7 GRASS KBFE: A PROTOTYPE SYSTEM DEVELOPED UNDER THE ENHANCED UIMS ................................................. 129
  7.1 Introduction ................................................................................... 129
  7.2 What Is GRASS ............................................................................ 129
  7.3 GRASS KBFE System Configuration ............................................ 130
  7.4 Integrating GRASS To Be A FOCUS Back-End ............................... 132
    7.4.1 GRASS KBMs ...................................................................... 133
    7.4.2 Environment Knowledge Base .............................................. 134
    7.4.3 Task Knowledge Base ........................................................... 135
    7.4.4 Action Knowledge Base ....................................................... 135
    7.4.5 Templates ............................................................................. 135
    7.4.6 Extraction of the Output ....................................................... 136
  7.5 User Tasks .................................................................................... 137
  7.6 The User Interface ........................................................................ 138
  7.7 The Help System .......................................................................... 141
  7.8 Evaluation .................................................................................... 143
  7.9 Summary ...................................................................................... 145

CHAPTER 8 DISCUSSION AND CONCLUSIONS ................................ 147
  8.1 Introduction ................................................................................... 147
  8.2 Summary ....................................................................................... 148
  8.3 Discussion and Proposals for the Further Work ............................... 155
  8.4 Conclusions .................................................................................. 156
# TABLE OF CONTENTS

**REFERENCES** .............................................................................................................. 158

**APPENDICES** .................................................................................................................. 178

A.1 Message and AIO Syntax ......................................................................................... 178

A.1.1 General Message Form ....................................................................................... 178

A.1.2 Message to the End User ...................................................................................... 179

A.1.2.1 Container Contents ......................................................................................... 179

A.1.2.2 Container Attributes ....................................................................................... 179

A.1.2.3 Abstract Interaction Objects ........................................................................... 181

A.1.2.4 Selection AIO ................................................................................................. 182

A.1.2.5 Inform AIO ...................................................................................................... 182

A.1.2.6 Hypertext AIO ................................................................................................. 183

A.1.2.7 Question AIO ................................................................................................... 183

A.1.2.8 Panel AIO ........................................................................................................ 184

A.1.2.9 New AIO ......................................................................................................... 184

A.1.3 Return Messages ................................................................................................. 185

A.2 Yacc Program List ...................................................................................................... 186

A.3 Lex Program List ....................................................................................................... 203

A.4 AIO Data Structure Definitions .............................................................................. 205
CHAPTER 1

INTRODUCTION

1.1 Introduction

There are many software systems for engineering and scientific applications in existence that were developed with great care in relation to accuracy and function and represent a significant investment. Their potential usefulness is considerable, and the cost of replacing them is often prohibitive. However, their benefits can be offset by the end user's difficulties in locating, selecting and employing the facilities that they offer. In addition, such software often suffers from badly designed or obsolete user interfaces that do not take advantage of modern user interface technology. Such software may be seen, therefore, as useful but not readily usable [Edmonds & MacDaid, 1990].

The FOCUS project (ESPRIT2 project 2620: Front Ends for Open and Closed User Systems) addresses these problems by developing generic tools and techniques for constructing and maintaining knowledge-based front-ends (KBFES) to existing open user systems (e.g. libraries, reusable software components) and closed user systems (e.g. fee-standing software, packages) for scientific and engineering applications. The major objective is to enhance the usability of the systems by providing both improved user interfaces and knowledge-based support to the end user. The project is undertaken by a consortium of eight European groups in six countries. The participants are NAG
A User Interface Management System (UIMS) called the Harness [Edmonds & MacDaid, 1990 and Edmonds et. al., 1992] has been developed for knowledge-based front-ends (KBFEs). This has been done as part of the FOCUS project, in which an architecture and set of tools have been developed to support the front-ending of existing software systems. FOCUS front-ends typically contain knowledge-based modules, which offer user support in both task-related matters and in selecting and using the appropriate existing packages. The Harness is responsible both for creating the user interface and for communications between the various subsystems. This communication takes place using a clearly defined abstract message structure. In the Harness, a set of core Abstract Interaction Objects (AIOs) was provided for knowledge-based modules to interact with the user. It is not realistic to expect the Harness to be a closed and fixed system. For each particular application, it will be necessary to decide, at least, on the presentation style. It may also be necessary to introduce new Abstract Interaction Objects, such as specialised forms for the knowledge-based module designer to construct a specific user interface and, again, it is most convenient to build such information into the Harness.

On the other hand, explicit help systems are relatively recent developments in the design of computer software. Prior to about 1975, help capabilities were rare and to be found only in a small number of systems [Kearsley 1988]. Today, help capabilities are considered essential components of well designed software and on-line help systems have come to be a standard feature of most new software systems. A software system that lacks help features is considered to be incomplete or inadequate. In the FOCUS
project, it was felt necessary to develop general tools to aid the construction of help systems. However, the writing of various help windows was very time consuming. There is a need for software assistance in the initial construction of help windows and in the tasks of modifying windows and transferring window to new environments. It has been suggested that some textual display facilities should be provided as part of the Harness UIMS that the FOCUS project has developed. Therefore, it is an unnecessary repetition for each module to provide its own interface to handle help information [Huang & Edmonds 1991].

Based on the above points, this thesis enhances the Harness' capabilities in the following two aspects: 1) developing a graphic interface builder called AIO Editor which is used by the interface designer to create and to extend new AIOs into the Harness, and 2) separating the domain/application independent part of help systems from applications to develop a general tool called General Help System Manager which is used in the design and the management of help systems.

1.2 Objectives of the Research

The main objectives of the research work presented in this thesis are:

1) To develop an Abstract Interaction Object Editor for user interface designers to create and modify AIO knowledge in a high graphical, direct manipulation manner.

2) To extend the Dialogue Control and the Presentation components in the FOCUS Harness to deal with the new AIOs.

3) To develop a domain/application-independent Help System Manager for the interface designers to create context-sensitive help facilities in applications.

4) To develop a prototype application system: a knowledge-based front-end to a GIS (Geographic Information System) application using the AIO Editor and the General Help System Manager mentioned above to demonstrate the advantages under the enhanced UIMS.
1.3 Overview of the Thesis

The thesis consists of three main parts, the first concerning the provision of the Abstract Interaction Object Editor, the second the General Help System Manager and the last a prototype application system. Chapter 2, 3, 4 and 5 are concerned with the work related to the provision of the AIO Editor, whilst chapter 6 deals with the provision of the General Help System Manager in the FOCUS Harness. Chapter 7 demonstrates a prototype application system developed under the enhanced FOCUS Harness UIMS.

Chapter 2, first, gives an overview of the well-known UIMS model, the Seeheim model, and then presents the FOCUS KBFE (Knowledge-Based Front-End) architecture. Two of the three components, the KBMs (Knowledge-Based Modules) and the BEM (Back-End Manager) are described in general. The Harness, the central component, is described in detail. The communication protocols (in an abstract message structure) between the various components are presented. Although most of this is not the author's work, some knowledge of the architecture is essential to understand the work to be described in later chapters.

Chapter 3 presents an investigative study on user interface tools in two aspects: user interface toolkit approaches and User Management System approaches.

Chapter 4 details the architecture of the AIO Editor and describes its role in the application interface design under the FOCUS KBFE.

Chapter 5 gives the implementation details of the new AIO management in the enhanced Harness.

Chapter 6 first briefly discusses the development, design and implementation of help systems and concludes that it is necessary to separate the domain/application-independent part of help systems from applications and to develop a general tool called General Help System Manager which is used in the design and the management of help
systems. This chapter then describes the General Help System Manager in detail: architecture, implementation, and message structures used to build context-sensitive help.

Chapter 7 presents a knowledge-based front-end to the GRASS, a geographic information system, to demonstrate the use of the AIO Editor and the General Help System Manager in developing an application system.

Chapter 8 summarises the work described in earlier chapters, proposes further work, which could be carried out in this area and concludes the thesis.
CHAPTER 2

FOCUS ARCHITECTURE

2.1 Introduction

In this chapter, first the concepts of the User Interface Management Systems (UIMS) are discussed, followed by a description of the logical components of a well-known UIMS model: the Seeheim model. Several post-Seeheim models are discussed briefly. Then the FOCUS architecture is described in detail. The core component in the FOCUS architecture, the Harness, is discussed in greater detail.

2.2 User Interface Management Systems

Recognition that human productivity in the use of computer systems is dramatically affected by the nature of the human-computer interface has created a demand for improved human-computer interaction. Since significant portions of all application systems are part of the user-interface, typical estimates of user-interface code vary between 30% and 60% of application code [Sutton & Sprague, 1978, and Myers & Rosson, 1992], the productivity of human-computer interface designers has become an important research topic. Investigators in the computer science community responded to this need by designing the User Interface Management System (UIMS) to assist in the development of successful interactive graphics systems [Olsen et. al., 1984].
2.2.1 UIMS Concepts

The term User Interface Management System, or UIMS for short, was coined by Kasik in 1982 after some preliminary research on how graphical input could be used to broaden the scope of human-computer interaction [Kasik, 1982], and was first used by Jim Foley and Jim Thomas [Löwgren, 1988]. It is used to describe software tools that enable designers to create a complete and working user interface without having to program in a traditional programming language. The user may have to use a programming language to implement additional functions such as database search, network communication, or scientific computation. But the user interface can be created, revised, and maintained in a high-level language, often with menus, forms, and manipulation actions [Olsen, 1987 and Myers, 1988 & 1989, Shneiderman, 1992].

What is a UIMS in reality? After comparison of several UIMS definitions [Hill, 1987, Norman et. al., 1986, Betts et. al., 1987], Löwgren summarises that Hill's general definition, although not very informative, is still a good UIMS definition [Löwgren, 1988]. Hill's general definition is given below:

**A UIMS comprises special tools and techniques for implementing user interfaces.**

The key idea in UIMSs is the *separation* between the semantics of the application and the user interface that is provided for the user to make use of that semantics [Dix et. al. 1993 & Shneiderman, 1992], that is to separate the user interface aspects from the underlying function code into a distinct sub-system [Edmonds, 1982]. Ideally, the two parts, the application and the user interface, might be separated in such a way that changes to one component cause no changes to the other [Cockton, 1987 and Hartson & Hix, 1989]. It is, however, believed that this ideal goal is probably difficult to achieve completely [Cockton, 1987 and Edmonds & Hagiwara, 1990]. On the other hand, it is problematic where this division should be made [Cockton, 1986b]. Without doubt, the first acknowledged instance of a development system that supported this application-presentation separation was in 1968 with Newman's reaction handler [Newman, 1968]. Researchers in the field such as Edmonds [Edmonds, 1982], Kasik
[Kasik, 1982], and Green [Green, 1985] have gone further and suggested that additional improvements could be made by separating out more functionality of the user interface. The notion of separable user interface systems has been widely discussed by Edmonds [Edmonds, 1992].

The advantages of this separation have been surveyed by Cockton [Cockton, 1986a]. We summarise them below:

Porability: to allow the same application to be used on different systems it is best to consider its development separation from its device-dependent interface.

Reusability: separation increases the likelihood that components can be reused in order to cut the development costs.

Multiple interfaces: to enhance the interactive flexibility of an application, several different interfaces can be developed to access the same functionality.

Customisation: the user interface can be customised by both the designer and the user to increase its effectiveness without having to change the underlying application code.

Once the separation between the application and the presentation is allowed, how those two components communicate must be considered. This role of communications is referred to as dialogue control. Conceptually, this provides us with the three major components of an interactive system: application, presentation and dialogue control. In terms of the actual implementation, this separation may not be so clear [Dix et. al. 1993]. The three components are detailed in the following section.

2.2.2 The Seeheim Model

Several models have been proposed for user interface management systems and all of them attempt to separate the user interface from the application itself. In order to provide a common base for comparing various UIMS implementations, a logical model of a UIMS was developed at the Workshop on User Interface Management Systems in
Seeheim, West Germany, November 1-3, 1983 [Pfaff, 1985]. Although this model is only logical and does not necessarily describe an actual implementation structure for a UIMS, it does provide a conceptual framework for a wide range of systems. This model has become widely known as the Seeheim model as shown in Figure 2.1, which shows the three logical components that "must appear in a UIMS" [Green, 1985]. The Logical components of a UIMS are identified below:

![Figure 2.1 The Seeheim Model of a UIMS](image)

**Presentation**: the component of the model is responsible for the external appearance of the user interface, including what output and input are available to the user. Communication with other components can be specified using some abstract form of input and output language. The presentation component transforms the physical input commands into this abstract language and interprets the abstract display operations in terms of the physical display capabilities.

**Dialogue Control**: this component defines the structure of the dialogue between the user and the application program. The user actions are the tokens produced by the input in the presentation component, which then drives the dialogue to produce the appropriate information to be sent to the application. Similarly, the dialogue control receives information back from the application, which may change its interaction state and then be sent to the presentation component for the display.

**Application Interface Model**: this component is a representation of the application from the viewpoint of the user interface. It is responsible for representation application data, making them available to the interface, and, conversely, providing access to the interface by the application. Output from the application is received at the application
interface, where it may be transformed before being passed on to the dialogue control. Information held in this component may include such items as the names of application routines, their parameters and results, etc.

In order to deal with larger amounts of data produced by the application, such as dynamic images, a direct path to the presentation component is proposed in this model. In this case the only function of the dialogue control is to establish the pipeline between the application and the presentation component. Once the path has been opened, dialogue control does not take part in the information transfer. This approach is particularly effective when large amounts of data are to be transferred to the screen [Green, 1985]. This model is often ascribed to Green, but a similar model was discussed by Edmonds as early as in 1982 [Löwgren, 1988].

The Seeheim model seemed at the time to be perfect outline for a UIMS, with the lexical, syntactical and semantical aspects of the user interface represented in the presentation, dialogue control and application interface modules respectively [Green, 1985]. However, with the introduction of the direct manipulation interfaces [Shneiderman, 1983] which require a lot of semantic feedback, the problems started. The Seeheim model, which was aimed at separating all user interface aspects from the application, provides poor semantic feedback and performance [Dance et al, 1987].

2.2.3 Post-Seeheim Models

Since the Seeheim Workshop, many papers have been written concerning modification to this model. For example, Dance et al's [Dance et al, 1987] UIMS model (see Figure 2.2) allows for a higher bandwidth connection to the application in order to provide the semantic feedback often used in direct manipulation interfaces.

As shown in Figure 2.2, Dance et al's model consists of a workstation agent, a dialogue manager, a semantic support component and the application. The workstation agent and possibly some of the dialogue manager correspond to the presentation component of the Seeheim model. The dialogue manager and semantic support
components are similar to the dialogue control and application interface model respectively. The main purpose of the dialogue manager is to provide a higher level of the abstraction for interaction services. The purpose of the semantic support component is among other things to provide application for semantic operations such as feedback, default values and error checking and to provide application-specific help information.

![Diagram of Dialogue Manager](image)

**Figure 2.2 Dance et al's UIMS Model**

A number of multi-agent models, such as the Model-View-Controller (MVC) model [Krasner & Pope, 1988] and its modern successor: the Presentation-Abstraction-Control (PAC) model [Coutaz, 1987], have been developed along the lines of the object-oriented and event-processing paradigms. In the multi-agent approach, the state of an interaction is distributed among a collection of co-operating units (also known as agents or objects) which react to a given set of external phenomena (stimuli) and which in turn produce new stimuli. An agent is a complete information processing system, which may receive input from the user, and send to, and receive messages from, other objects. It may also communicate with a workstation agent to replace, update or delete a previously displayed entity, or create new display entities.

The basic idea behind MVC is the separation of a graphical interactive application into two parts: the abstract application (or model) which can perform the necessary computations without reference to any form of I/O, and the user interface part, which has the responsibilities for all I/O functions. The user interface part is split into two parts: the view, which handles all display (output) operations; and the controller, which handles user input. The view and controller can communicate with each other without interacting with the model. Communications from controller to model cause a change of state in the application in some way, while communications from controller to view
cause a change in the visible representation without affecting the state of the model. The separation of model and view/controller fits nicely into the object-oriented programming model, as the interface between them is defined in terms of messages understood by the model and the answers returned to the view model is hidden from the view and the controller [Hopkins & Horan, 1995].

Figure 2.3a Coutaz’s PAC Model

Coutaz’s PAC model is based on a collection of agents. An agent defines a competence at some level of abstraction. It is a three-facet logical cluster that includes a presentation, an abstraction and a control (Figure 2.3a). At any level of abstraction: 1) a presentation is a perceivable behaviour; 2) an abstraction is the functional core, which implements the functions that the application is able to perform; and 3) a control links an abstraction to a presentation, controls the behaviours of the two perspectives it serves, remembers a local state for supporting multi-thread dialogue, and maintains relationship with other agents [Bass & Coutaz, 1991]. Coutaz’s PAC architecture followed the Seeheim model at the top level and within each agent or interactive object, and recursively constructed an interactive system in a hierarchical manner (Figure 2.3b).

PAC differs from the older Model-View-Controller (MVC) model in the way it divides up various functionalities. A model in the MVC model corresponds to a PAC abstraction: both represent a competence that is media independent. In MVC a view is responsible for outputs and a control handles inputs. Thus, the combination of an MVC view and an MVC control corresponds to a PAC presentation. The PAC control has no
direct correspondence in MVC. It can, however, be considered as a particular class of model [Bass & Coutaz, 1991].

As Edmonds points out, however, most of the models are variations on the Seeheim view, rather than radical departures from it [Edmonds, 1992]. In fact, a re-assessment has confirmed most of those early ideas. The most significant innovation in the Slinky model [Bass et. al., 1992], proposed in that review, is the notion that the division of labour between the interface components might be flexible and depend, for example, on the maturity of a particular function [Edmonds, 1992]. However, the Seeheim concept is still at the centre of the Slinky model.

In the Slinky model (Figure 2.4), the Dialogue Component corresponds to the Seeheim Dialogue Controller. In the Seeheim model, however, the role of this component was limited to some obscure syntactic sequential processing such as the combination of lexical inputs into command level abstractions. The Dialogue Component in the Slinky model has responsibility for task-level sequencing, both for the user and for the portion of the application domain sequencing that depends upon the user; for providing multiple
view consistency; and for mapping back and forth between domain-specific formalisms and user-interface-specific formalisms.

The Domain-Specific Component and the Domain Adaptor Component are different terms for denoting the notions of Application and Application Interface introduced in Seeheim. The Domain-Specific Component controls, manipulates and retrieves domain data and performs other domain-related functions. The Domain-Adaptor Component serves as a mediator between the Dialogue and the Domain-Specific Components. It triggers domain-initiated dialogue tasks, re-organises domain data, and detects and reports semantic errors.

The Slinky model segments the Seeheim Presentation into two levels of abstraction: the Presentation Component and the Interaction Toolkit Component. The Interaction Toolkit Component implements the physical interaction with the end-user. The Presentation Component serves as a mediator between the Dialogue and the Interaction Toolkit Component. It provides a set of toolkit-independent objects for use by the Dialogue Component. Decisions about the representation of media object are made in the Presentation Component.
On the other hand, user interface management systems exploit the concept of separation of concerns introduced earlier with database management systems. However, these systems have not normally contained knowledge-based user support systems and they have not specifically addressed the general problems of front-ending existing software [Edmonds & McDaid, 1990].

Edmonds and McDaid [Edmonds & McDaid, 1990] developed an architecture for knowledge-based front ends (KBFEs) by extending knowledge-based modules, which provide the user with guidance and assistance, into the Seeheim model. The other additional module is a so-called Back-End Manager (BEM) [Pratt, 1990], which enables the interfacing with applications in a way that provides a clean and clear separation between the rest of the interface and the applications. This architecture is discussed in detail in the following section.

### 2.3 FOCUS KBFE Architecture

The FOCUS KBFE architecture, which was developed within the FOCUS project, is an extension of the Seeheim model. The architecture consists of several modules running as separate, concurrent processes that need not be physically located on the same machine and can communicate with one another over networks if necessary. The different software processes communicate with one another, using a single message-passing convention, through the module known as the Harness. As shown in Figure 2.5, the Seeheim model, which consists of a presentation layer, a dialogue control layer and an application-interface layer, is subsumed in the Harness. The application interface layer contains an explicit underlying application model in order to provide a clear and clean interface between the front and back ends. Although this approach has been widely adopted in spirit, the key issue of specifying a complete and consistent notation in which this interface can be expressed remains unresolved. The messages employed in FOCUS are expressed in relation to Abstract Interaction Objects (AIOs) and are introduced as an approach to the interface notation problem [Edmonds & McDaid, 1990].
As was understood in the Seeheim model, some communications can not be reduced to the bandwidth implied by the message type structure, for example dynamic 3-D images generated by a back-end. The Seeheim solution has been adopted in the FOCUS KBFE architecture. Thus, a high bandwidth route is provided from the back-ends to the presentation layer of the interface. It passes through a switch, which directs the data, under the direction of Dialogue Control without it having to pass through the various Harness components.

![FOCUS KBFE Architecture Diagram]

In addition to the three Seeheim layers, important extensions have been made to support the KBFE solution. Firstly, the KBFE contains an unspecified number of Knowledge-Based Modules (KBMs), which provide the user with the necessary guidance and assistance. The other additional module in the FOCUS KBFE architecture is the Back-End Manager (BEM), which enables the interfacing with applications in a way that provides a clear and clean separation between the rest of the interfaces and applications. The number of KBMs and back-end applications is not limited by the KBFE architecture and its modular design enables a range of applications to be integrated to support users performing their tasks [Edmonds et al, 1992]. The three main components: the Harness, Knowledge-Based Modules and the Back-End Manager are described in the following sections.
2.3.1 The Harness

The core module in the FOCUS KBFE architecture is the Harness. In effect, the Harness is the KBFE's operating system [Edmonds & McDaid, 1990]. It has a range of responsibilities that involve communication and control. It is responsible for all communications within the system, control of the dialogue with the user and the interaction presentation issues. The various modules in the system communicate with the user through the Harness. In providing this facility, the Harness maps between the Abstract Interaction Objects (AIOs) of interest to the modules and the physical realisations of them that are meaningful to the user. Thus, for example, the selection of an item from a list is mapped into a screen event such as a specific pull-down menu. The various modules can also communicate with one another, using the same message structures. In this case, the Harness simply routes the message and looks after the related housekeeping. The full Harness architecture and its components are discussed in section 2.4.

2.3.2 Knowledge-Based Modules

It is the Knowledge-Based Modules (KBMs) which provide the user with co-operative support and guidance. They can incorporate knowledge about the functions supplied in the application software as well as the ways in which they can and cannot be used in relation to particular problems and so on. They can also contain the information about the problem domain, the user's environment and the characteristics of the end users themselves. The choices rest with the KBFE developer. However, the key point is that, whatever facilities the developer is able and willing to supply, the architecture allows for them to be appropriately modularised and run as concurrent processes. The number and roles of these modules will vary depending on the particular application. Typically, they will provide the user with help on the use of the system, advice on the selection of alternative facilities or approaches and explanation. Thus they can provide strategic as well as syntactic help.
2.3.3 The Back-End Manager

In order to maintain the concept of interface separability within the KBFE architecture, while the Harness and the Knowledge-Based Modules (KBMs) are concerned with the 'whys' and 'wherefores' of what the end user is or should be doing (i.e. the end user's task), it is the Back-End Manager (BEM) that takes care of the 'how' in relation to the software systems available. The BEM's prime responsibility, therefore, is that of mapping an application-independent specification of the 'what' (task) into an executable specification of the 'how' with respect to a particular application or package (i.e. mapping from a semantic to a syntactic specification), of controlling the actual execution of the application, and of extracting the relevant information from the output of that application and passing that information back to the appropriate KBFE component.

In order to carry out its role, the BEM must contain the knowledge about the syntax, semantics, functionality and environment of the application software. This is stored in the form of specification objects (tasks and actions), back-end I/O objects (templates and extractions) and an environment knowledge base. Figure 2.6 illustrates the structure of the BEM showing its relationship to the rest of the KBFE architecture. The
brief description of the various components in the BEM is given as follows. Refer to [Pratt et. al., 1990 & Galmes et. al., 1991] for the detail of the Back-End Manager.

**KBMs:** one of the relevant KBM's responsibilities is, through the Harness, to send relevant back-end tasks in the FOCUS message structure to the BEM for processing and to receive reply messages from the back-end via the BEM with the output parameters instantiated.

**Task Manager:** on receiving a task message from a relevant KBM, the Task Manager, based on the knowledge available in the environment knowledge base, makes a decision as to which is the appropriate back-end to use and then, based on the knowledge in the task knowledge base, decides an ordered sequence of one or more back-end specific actions which are used to complete the task and makes appropriate calls to the Action Manager. On completion of the task, a message is sent back via the Harness to the calling KBM with the appropriate task output. The back-end expert must provide the knowledge of the tasks of the back-ends within the task knowledge base, which maps a task to an ordered sequence of one or more back-end actions.

**Action Manager:** on receiving a request to execute an action from the Task Manager, the Action manager, based on the knowledge in the action knowledge base, invokes the Template Engine to mix the action input parameter with the template, executes the back-end and finally invokes the Extraction Engine which interprets the back-end output via the extraction definition and action output parameters. The back-end expert must define back-end actions in the action knowledge base. An action is one object which creates a particular sequence of executable commands to a back-end depending on the action input parameters, processes the resulting output from the back-end and extracts the required output parameters.

**Template Engine:** at run-time, the Template Engine mixes the action-input parameters with the template to create the correct sequence of executable back-end commands. The
templates are built by a back-end expert using the TPL (TemPlate Language) [Galmes et. al., 1991], one of the BEM tools.

*Extraction Engine:* at run-time the Extraction Engine uses the extraction definition to extract the appropriate action output parameters from the output of the back-end which has resulted from the input generated by the Template Engine. The extraction programs (definitions) are built by a back-end expert using the XTL (eXTraction Language) [Galmes et. al., 1991] for extracting various data objects such as vectors, matrices, text, etc. from the output ASCII files.

The procedures to execute the back-end application are summarised as follows:

1. the BEM receives a message (from a KBM) containing a description of a back-end 'task' to be performed.
2. the Task Manager decides on the appropriate application with which to implement the task.
3. the Action Manager decomposes the 'task' into one or more back-end 'actions'.
4. the Template Engine maps the actions into back-ends commands and executes the back-end application.
5. the Extraction Engine extracts the 'relevant information' from the output of the application.
6. the Task Manager sends a message back to the calling KBM with the 'relevant information', which then may be conveyed to the user or used as a basis for further reasoning by the KBMs.

It is clear that the BEM requires a considerable amount of knowledge and that much of that knowledge is similar to that required by the Knowledge-Base Modules. There is, however, a significant distinguishing feature. When a task is given to the BEM, it should be fully specified, so that the knowledge available must be complete in the sense of being sufficient to execute that task. On the other hand, Knowledge-Based Modules will typically be concerned with advising and helping the user to make decisions, so
that they would normally operate in conjunction with the user. Thus, the knowledge available to the KBMs may be incomplete or tentative while the knowledge available to the BEM must be clear and complete.

2.4 Harness Architecture

Figure 2.7 shows the Harness components in more detail. The three layers presentation, dialogue control and application interface in the Seeheim model correspond to the Dynamic Presentation Manager (DPM) and the Physical Presentation Layer (PPL), the Dynamic Dialogue Manager (DDM), and the Harness Communications Manager (HCM) and the Application Communications Manager (ACM) in the FOCUS Harness respectively. In addition to the Harness Layers, there are two relevant knowledge bases: the Configuration Knowledge Base and the Presentation Knowledge Base.

![Figure 2.7 The Harness Architecture](image-url)
2.4.1 Communications Manager

The Communications Manager, or the Application Interface Model in Seeheim terms, is divided into two components: the Harness Communications Manager (HCM) and the Application Communications Manager. Its key function is to allow the various components of the system to exchange data. In fact, the FOCUS KBFE architecture employs a star topology, with the HCM as the centre of the star and the various ACMs as its satellites (Figure 2.8). The FOCUS applications - the Knowledge-Based Modules and the Back-End Manager - rely on the ACM to provide communication services. The ACM relies on the HCM to perform message routing in most cases (except that if the destination application is in the same process as the sending application, the ACM delivers the message directly). The second Communications Manager function is centralised control. It is important to realise that the FOCUS KBFE is a distributed system. Control of this distributed system is centralised within the HCM. The HCM maintains the status of all the applications.

![Diagram](image)

**Figure 2.8 The FOCUS KBFE's Star-Topology Architecture**

**Harness Communications Manager (HCM):** the HCM provides the communication support for the Harness. It controls the starting and stopping of all KBFE components including other components of the Harness and the Back-End Manager. It takes all the information necessary to start a particular KBFE from the Configuration Knowledge Base. Once the KBFE has started, the HCM handles the routing of all communications messages, and detects and deals with communications failure. The HCM is the only
Harness component, which understands 'where' a KBM or back-end application resides and how messages are to be delivered to it.

*Configuration Knowledge Base:* all the information necessary to configure a particular KBFE is stored declaratively in the Configuration Knowledge Base. This information includes knowledge about all the host machines which may be used in the KBFE, the physical Unix processes which may run, the KBMs which may be grouped in each physical process, and the resources and tools which KBMs or the Back-End Manager may use. The Configuration Knowledge Base also holds information about the KBFE components that should be started when the Harness itself is started. The HCM consults the Configuration Knowledge Base at the KBFE initialisation.

*Application Communications Manager (ACM):* the ACM is responsible for managing the communications between an application process and the Harness. It is that part of the Communications Manager, which resides with the application software. Each time a physical process is started within the KBFE, the HCM supplies it with a copy of the ACM. The ACM may then be used by any of the logical processes running within that physical process. The ACM provides all necessary message transmission and reception services for the KBMs and/or tools that have been bundled into the process. The ACM sends and receives messages to/from a physical process through the HCM, while messages that are destined for applications co-resident in the ACM's process are routed directly - that is, not through the HCM. Besides sending and receiving messages, the ACM provides a process synchronisation service for Prolog KBMs - i.e. the ability to suspend processing until a particular message is received from another component. The ACM also provides a channel instrumentation service and supports the registration of external event streams that do not use the FOCUS protocol.

2.4.2 Dynamic Dialogue Manager

The Dynamic Dialogue Manager (DDM) is the extension of the concept of the Dialogue Control component in the Seeheim model. It maintains the interfaces with the ACM and the Dynamic Presentation Manager (DPM). It is responsible for the logical management of all communications with the end user, for negotiating conflict resolution at the end user interface and for maintaining the logical link between the state of the KBFE and the state of the interface. It decides whether the contents of Abstract Interaction Objects (AIOs) in a message to the end user should be 'safely' realised on screen. If so, it will forward the message to the DPM. Otherwise, it will deny access to the end user - e.g. by queuing or discarding the message - and/or report this action to the message sending KBM. This is important because, with the potentially large number of diverse KBFE components which may interact with the end user through the Harness and the possibly incomplete knowledge about the behaviour of particular application systems, it is sometimes necessary to hold back requests for interactions with the user. This could be, for example, because the screen is already very full and there is a clear risk of overloading the end user. In such cases the DDM might have to negotiate with the module requesting interaction in order to resolve the situation.

2.4.3 Presentation

The Presentation function is divided between two components: the Dynamic Presentation Manager (DPM), which is independent of any software environment, and the Physical Presentation Layer (PPL), which may contain material which is specific to an operating system, window manager or widget set. The Presentation Knowledge Base (PKB) contains definitions of all available AIO prototypes and the DPM instantiates these prototypes using the contents of messages. The PPL is responsible for directly mapping instantiated interaction objects to a particular window management system and/or widget set.
Presentation Knowledge Base (PKB): the PKB contains definitions of all AIO prototypes that may be used by the KBFE. Since an AIO is defined as a class of interaction objects in which the presentation details are not specified, an AIO prototype consists of a logical description of the AIO - i.e. the contents and attributes it may have - and a mapping of contents and attributes which must be realised on screen to interaction primitives. The KBM developer uses the message to fill in the contents of the AIO and to set its attributes. The Harness provides a set of 'ready-made' core or standard AIOs: inform, query, question, panel, hypertext, matrix and selection that form the PKB i.e. the static AIO library which is contained in the DPM code.

Dynamic Presentation Manager (DPM): the DPM is responsible for realising AIOs. It maintains data structures that represent AIOs in the abstract. Thus, the DPM determines the values to place into the attribute slots of each AIO. It is responsible for the logical creation, modification, destruction and management of AIOs. It relies on the PPL for the actual visual realisation. In this way, a clear separation is achieved between the logical description of dialogue components and the physical realisation of them.

Messages to create new interaction objects are processed as follows:

- DPM establishes the type of each AIO within the message
- DPM consults the relevant AIO prototype definitions from the static AIO library (i.e. the PKB) and checks the AIO syntax, which is represented in a Backus-Naur Form (BNF) grammar. If the definition cannot be found or a syntax error is found, this fact is reported to the DDM
- DPM instantiates the AIO prototype definitions as necessary from the contents of the message, overwriting the default values
- The result is passed to the PPL for the actual visualisation

A message containing the identifier (Id) of an AIO, which has already been visualised is automatically treated as an 'update' message. It may:

- Add one or more new sub-AIOs to an existing container AIO
• Change the contents of one or more AIOs
• Change any of the attributes of an AIO, including controls

A message may also request the destruction of one or more AIOs. The destruction of a container AIO will cause the destruction of all standard AIOs contained within it.

When an event occurs at the end user interface - e.g. a control is activated - the PPL informs the DPM. The DPM is then responsible for sending an appropriate return message to the KBM. A return message can be activated in three ways:

• Event: when some pre-specified event occurs - e.g. the user activates a control
• Request: when the KBM issues a request for the state of one or more AIOs to be returned
• Sample: when the Harness repeatedly returns the state of one or more AIOs at specified intervals on the KBM's requirement

Physical Presentation Layer (PPL): the PPL is responsible for the visual realisation of AIOs on screen and deals directly with the window manager. The PPL communicates with the rest of the Harness via the DPM. When the DPM has parsed the input message, it creates or updates a data structure representing the instantiation of the AIO. Each element of the structure has been flagged during the parsing as to whether the corresponding visualisation (e.g. widget in X-Window) must be created, updated, or left untouched. The PPL receives AIOs for realisation in the form of C data structures. When the end user interaction is complete, the PPL revises the data structures to reflect the interaction and return them to the DPM.

The design of the PPL component is such that only the calls to the window manager should need to be changed to accommodate other windowing environments - e.g. MS-Windows. The FOCUS Harness was implemented under the X-window system [Gettrs and Scheifler, 1986] and the OSF/Motif toolkit [OSF, 1990]. The Macintosh version of the Harness called MacHarness [Huang et. al. 1992] has been implemented under the Macintosh window environment in the MUMS project (IED4/1/1256: "Multimedia User
Modelling Systems\textsuperscript{a}). The implementation of the MacHarness takes the advantage of the separation between the DPM and the PPL. In fact, only the PPL code needs to be re-written in the MacHarness.

2.5 Messages, AIOs and User Interface

2.5.1 Messages

As mentioned before, the central communications concept is based upon the simple notion of sending and receiving messages. A message is the way in which a KBM can communicate with other KBMs, with the Harness itself, with the BEM or with the end user. In the FOCUS KBFE architecture, this communication takes place using a clearly defined abstract message structure. The basic format of a message is:

\begin{quote}
message(identifier, content, sender : receiver)
\end{quote}

There are three types of messages: control messages, task messages and user interaction messages. The control messages, usually sent by a KBM such as a request to activate a new KBM process, are dealt with by the Harness Communications Manager (HCM). These messages include 'instigate process', 'terminate process' and 'end of session'. For example, to start a KBM called 'help_manager', the following message should be sent to the HCM:

\begin{quote}
message( m1, instigate( help_manager ), mykbm : feh ).
\end{quote}

The task messages, usually transferred between KBMs and BEM such as a request to execute a back-end task, are dealt with by the relevant KBM or BEM. The HCM only passes them to the destination. For example, to get the raster maps available in the current GRASS (Geographical Resources Analysis Support System) [Westervelt, 1991] database, the KBM sends a task message as follows to the Back-End Manager (BEM):

\begin{quote}
message( task1, goal( list, get_map_list( raster, MapList ) ), mykbm : bem ).
\end{quote}
A message will be returned by the BEM with the variable 'MapList' instantiated. Chapter 7 describes the GRASS KBFE in detail.

The user interaction messages are, typically, to or from the user. In the former case, for example, they are passed by the Communications Managers via the DDM to the DPM. The message content is expressed in the terms of operations on the Abstraction Interaction Objects (AIOs).

### 2.5.2 AIOs and User Interface

As we know that an Abstract Interaction Object (AIO) is a prototype object in which the details of presentation are not specified - i.e. each prototype simply defines the 'contents' and 'attributes' that an object may have, but says nothing about the appearance of the object on screen. The interface designer must use a message to fill in the contents of AIO and set its attributes. A logical user interface is the meaningful combination of some AIOs with their attributes partly or fully instantiated. If an AIO's attributes are uninstantiated, its default attributes will be used. The Harness maps this information to the window manager which is being used and realised the completed object on screen. There are two types of AIOs:

- Standard AIOs, which include 'choice', 'inform', 'input', 'hypertext', 'question', 'selection' and 'text', are designed to be the basic components of the user interface, and
- Container AIOs, which include 'group', 'panel' and 'query', are used to group standard AIOs logically and apply attributes - e.g. controls - to the whole group.

The standard AIOs must be contained within container AIOs. A user interaction object shown in Figure 2.9 will be generated from the message below:

```message (setup_main_menu,
    ( destroy,
    tell,
```
group ( grass_task_menu, 

selection ( main_selection, 

[ start ( external : 'Start Graphics Driver' ), 
  stop ( external : 'Stop Graphics Driver' ), 
  display ( external : 'Display Raster Maps' ), 
  routing ( external : 'Optimal Routing' ), 
  exit ( external : 'Quit System')], 
[ label : 'Select one Task Please:', 
  limits : (0, 1) ]),

[ title : 'Grass Main Window', 
  controls : [ ok(external : 'Execute Task'), 
              quit(external : 'Quit') ] ]), 

grass_kbm : user ).

Figure 2.9 A User Interaction Object Example

The bold words are the Harness' reserved words. The abstract nature of the message and the hierarchical organisation of syntax are clearly indicated. Refer to "Front End Harness Version 2: Specification" and "Harness V3.0: FOCUS Message User Guide" [Heggie et. al. 1991 & 1992] for the syntax definition of AIOs. More complex interaction objects can be created with combinations of AIOs and sub-groupings of
AIOs. It can be seen that the combination of the message and AIO approach produces a flexible solution to the problem of interface and dialogue specification.

The user causes messages to be sent to the sending KBM, when the user interacts with the interaction objects. For example, when the user selects the item 'Start Graphics Driver' and presses the button 'Execute Task', the DPM generates the following message which presents the current state of the user interface shown in Figure 2.9 and, via the ACM and HCM, sends this message to the sending KBM:

\[
\text{message (setup_main_menu,}
\]
\[
\quad \text{return ([grass_task_menu], control (ok), [main_selection = [start]]),}
\]
\[
\quad \text{user : grass_kbm).}
\]

When it receives the return message, the KBM may make some inferences and send messages to the user, BEM, or other KBMs for the further interaction.

Thus the creation of the user interface within the system can be seen to occur at three distinct levels. Firstly, the message syntax provides a formal, highly abstract, notation for describing the user interface at the requisite complexity. Secondly, the AIO prototype provides a functional realisation of the notational description of the user interface, but still in an abstract form. Finally, the actual physical implementation of the user interface is realised as the particular interaction objects that are created at the PPL.

The abstract nature of the interface specification ensures that the quality of presentation in resulting interface is only dependant on the realisation of the particular interaction objects. Thus, the presentation quality is more easily controlled because it is localised in one place, i.e. the Physical Presentation Layer.

### 2.6 Summary

The concepts of the UIMS are discussed briefly at the beginning of this chapter. The motivation for UIMS is simple: to reduce the repetition of the large percentage of code that is written to implement the user interface. A UIMS intends to be a flexible, re-
usable interface module that communicates with the application software on one side and with the end user on the other side. The key idea in UIMS is to separate the user interface from the applications. A well-known abstract UIMS model, the Seeheim model, which consists of three logical components - presentation, dialogue control and application interface - is described in general. Three examples of post-Seeheim models - Dance et al's, Coutaz's PAC and the Slinky - are given following the description of the Seeheim model.

Then, the FOCUS KBFE architecture, which is an extension of the Seeheim model, is presented. The FOCUS emphasises the front-ending of existing software applications by extending the Knowledge-Based Modules, which provide the user with guidance and assistance, and the Back-End Manager which enables the interfacing with the integrated applications in a clean and clear separation between the rest of the interface and the applications. Thus, the FOCUS KBFE consists of three components - the Harness, KBMs and BEM. The functions provided by the three FOCUS components are discussed in detail.

The core component in the FOCUS KBFE, the Harness, is discussed in greater detail in section 2.4. The Harness, in effect, is the FOCUS KBFE's operating system and is actually a UIMS. The Harness is divided into five components - HCM, ACM, DDM, DPM and PPL. The Communications Managers - HCM and ACM - control all of the communications in the FOCUS KBFE. This communication takes place using a clearly defined abstract message structure. DDM maintains the logical link between the state of the KBFE and the state of the interface. DPM maintains the logical state of the interface, while PPL maintains the physical state of the interface.

In the last part of this chapter, FOCUS messages and AIOs are discussed. There are three kinds of messages - control, task and user interaction messages. The concept of AIO is introduced into the Harness. An AIO is a prototype object in which the details of presentation are not specified. A user interface is the meaningful combination of the standard AIOs with attributes partly or fully instantiated. The standard AIOs must be
CHAPTER 2 FOCUS ARCHITECTURE

contained within the container AIOs. To create a user interface, a KBM must send the Harness a message that contains the specification of AIOs. An example of user interaction messages is illustrated in section 2.5.2. This chapter intends to provide the necessary background knowledge for the full understanding of the following chapters.
CHAPTER 3

INVESTIGATIVE STUDY OF USER INTERFACE TOOLS

3.1 Introduction

Many tools have been created to make user interfaces cheaper and easier to design and implement; some have been very successful. For example, Apple's MacApp has been reported to reduce development time by a factor of four or five [Shmucker, 1986]. User interface tools come in two general forms: user interface toolkits (UITs) and user interface development systems (UIDSs) [Myers, 1989]. The latter are usually regarded as UIMSs, which have been described in chapter 2. The user interface development systems can be classified into four groups: language-based user interface specification, direct graphical user interface specification or graphical user interface builders, automated user interface generation and model-based user interface development systems. This chapter intends to survey the state of the art. First the toolkit approach to create user interface is introduced. The UIMS approaches - language-based specification, direct graphical interface specification, automated interface generation and model-based user interface development systems - are described in the following sections.
3.2 User Interface Toolkit Approaches

3.2.1 Toolkits

The most common tool for user interface creation is a user interface toolkit. A user interface toolkit is a library of interaction techniques, where an interaction technique is a way of using a physical input device (such as a mouse, keyboard, table, or rotary knob) to input a value (such as a command, number, percent, location, or name), along with the feedback that appears on the screen [Myers, 1989]. Examples of interaction techniques are windows, scroll bars, pull-down or pop-up menus, data-entry fields, buttons, dialogue boxes, etc. A toolkit provides the programmer with a set of ready-made interaction techniques - alternatively called interaction objects, gadgets or widgets - which can be used to create a user interface for an application. The interaction objects have a predefined behaviour, such as that described for the button, that comes free without any further programming effort. Toolkits exist for all windowing environments (e.g. OSF/Motif and XView for the X-Window system, the Apple Macintosh Toolbox and the Software Development Kit for Microsoft Windows). One of the motivations for the approach is that the final user interface will have a "look and feel" (LAF) consistent with others created using the same toolkits.

There are two kinds of toolkits. The most conventional is a collection of program routines that can be called by the application programs which are usually written in a conventional programming language (such as C, Pascal, BASIC, COBOL, Ada, etc.). Examples are the Microsoft Windows Software Development Kit (SDK) and the Apple Macintosh Toolbox. Conventional programming languages with accompanying toolkits are familiar to experienced programmers and afford great flexibility.

The following example program called "msbutton.c" illustrates the use of the Microsoft Windows SDK. This program displays a push-button, and when the user presses the mouse on the push-button, the program displays the message "PushButton selected", then terminates:
/**-------------------------------------------------------------------------------
*** File: msbutton.c:
*** Description: An Example for Microsoft Windows SDK
***-------------------------------------------------------------------------------
/* include files */
#include <windows.h>
#define IDC_PUSH_BUTTON 256 // define button Id
long FAR PASCAL _export WndProc (HWND, UINT, UINT, LONG);

int PASCAL WinMain (HANDLE hlnstance, HANDLE hPrevlnstance,
LPSTR lpszCmdParam, int nandShow)
{

 static HWND hwnd ;
 static HWND hwndButton;
 static MSG msg ;
 static HWND wndclass ;

 if (!hPrevlnstance)

 wndclass.style = CS_HREDRAW | CS_VREDRAW ;
 wndclass.lpfnWndProc = WndProc ;
 wndclass.cbClsExtra = 0 ;
 wndclass.cbWndExtra = 0 ;
 wndclass.hInstance = hlnstance ;
 wndclass.hIcon = LoadIcon (NULL, IDI_APPLICATION) ;
 wndclass.hCursor = LoadCursor (NULL, IDC_ARROW) ;
 wndclass.hbrBackground = GetStockObject (WHITE_BRUSH) ;
 wndclass.lpszMenuName = NULL ;
 wndclass.lpszClassName = szAppName ;
 RegisterClass (&wndclass);
 hwnd = CreateWindow (szAppName, "MS BUTTON" ,
 WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
 hwnd, NULL, NULL, NULL, hwnd) ;

 hwndButton = GetDlgItem (hwnd, IDC_PUSH_BUTTON) ;
 GetClientRect (hwnd, &rect) ;
 MoveWindow (hwndButton, 0, 0, rect.right, rect.bottom, TRUE);

 TranslateMessage (&msg) ;
 DispatchMessage (&msg) ;

 return msg.wParam ;
}

/* window procecedure */
long FAR PASCAL _export WndProc (HWND hwnd, UINT message, UINT wParam, LONG lParam)
{

 switch (message)
 {
 case WM_CREATE:
 GetClientRect (hwnd, &rect) ;
 hwndButton = CreateWindow ("button", "Push Here",
 WS_CHILD | WS_VISIBLE, 0, 0, rect.right, rect.bottom,
 hwnd, IDC_PUSH_BUTTON, (LPCREATESTRUCT) lParam)->hInstance, NULL) ;
 return 0 ;
 case WM_SIZE:
 hwndButton = GetDlgItem (hwnd, IDC_PUSH_BUTTON) ;
 GetClientRect (hwnd, &rect) ;
 MoveWindow (hwndButton, 0, 0, rect.right, rect.bottom, TRUE);
 return 0 ;
 case WM_COMMAND:

CHAPTER 3 INVESTIGATIVE STUDY OF USER INTERFACE TOOLS

switch (wParam)
{
    case IDC_PUSH_BUTTON:
        MessageBox(hwnd, "PushButton selected.", "msButton", MB_ICONINFORMATION);
        PostQuitMessage (0);
        break;
    default:
        break;
}
return 0;

The other kind of toolkits adds features associated with object-oriented programming (OOP), such as inheritance, to make it easier for the designer to customise the interaction techniques. An example is the X-Window Toolkit (Xtk). Other toolkits, such as Grow [Barth, 1986] and Coral [Szekely and Myers, 1988] add constraints to the object-oriented approach. The constraints let the designer specify relationships among objects and have the relationships maintained by the system. For example, the designer can specify that a line is connected to two rectangles, and the system will automatically move the line whenever either rectangle is moved.

The OSF's [OSF, 1990b] Motif example program called "xmbutton.c", which has a similar function to the Microsoft SDK example shown above, is listed below:

```c
/**-------------------------------------------------------------------------------
***
file: xmbutton.c
project: Motif Widgets example programs
*** description: This program creates a PushButton widget.
*** (c) Copyright 1989 by Open Software Foundation, Inc.
*** (c) Copyright 1989 by Hewlett-Packard Company.
**-----------------------------------------------------------------------------*/

#include <Xll/Intrinsic.h>
#include <Xm/Xm.h>
#include <Xm/PushB.h>

/* Functions defined in this program */
void main ();
void activateCB (); /* Callback for the PushButton */
/* global variables */
char *btn_text; /* button label pointer for compound string */
/** main - main logic for demo1 program */
void main (argc,argv)
unsigned int argc;
char **argv;
{
    Widget toplevel; /* Shell widget */
```
CHAPTER 3 INVESTIGATIVE STUDY OF USER INTERFACE TOOLS

3.2.2 Virtual Toolkits

Although there are many small differences between the various toolkits, much remains the same. For example, all have some type of menu, button, scroll bar, text input field, etc. Although there are fewer windowing systems and toolkits than there were 5 years ago, people are still finding that they have to do a lot of work to convert their software from Motif to OpenLook to the Macintosh and to MS-Windows [Myers, 1993].

Therefore, a number of systems are being developed that try to hide the differences between the various toolkits, by providing virtual widgets which can be mapped into the widgets of each toolkit. The user interface code can then be linked to different actual toolkits, and will run without change. XVT (Extensible Virtual Toolkit) [XVT, 1991] is such a system that provides a C or C++ interface that hides the differences of the
Motif, OpenLook, Macintosh, MS-Windows, and OS/2-PM, and also the character-based windows (Figure 3.1).

![Diagram](image)

**Figure 3.1 XVT Supports Application Development on Five Window Systems**

The advantage of the XVT approach is that on each platform the application has the same look and feel as other applications built locally on that platform. However, there are two major drawbacks to this approach. First, XVT is forced to provide only those functions common to all platforms - the lowest common denominator solution. Second, the designer must use different support tools (e.g. layout editors) on each platform [Pausch, et al., 1991]. A different approach to portability is taken by SUIT (Simple User Interface Toolkits) [Pausch, et al., 1991].

The innovative SUIT (Simple User Interface Toolkits), built for education purposes, allows user-interface design students to create applications that will run transparently across Macintosh, IBM PC, Unix/X, and Silicon Graphics platforms. Pascal succeeded for two basic reasons. Firstly, it presented a small, consistent language with simple semantics. Secondly, the language was designed so that compiler could easily be implemented for a variety of hardware platforms. In the same spirit as Pascal, SUIT was developed to offer the same two advantages as Pascal: it is easy to learn, and it can
be easily ported to different platforms [Pausch, et al., 1991]. Figure 3.2 shows the software layering, which makes SUIT portable.

<table>
<thead>
<tr>
<th>SUIT (Simple User Interface Toolkit)</th>
<th>SRGP (Simple Raster Graphics Package)</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Mac Toolbox</td>
</tr>
<tr>
<td>UNIX machine</td>
<td>Macintosh</td>
</tr>
<tr>
<td>BGI driver</td>
<td>DOS PC</td>
</tr>
</tbody>
</table>

Figure 3.2 SUIT Layered Software for Portability

3.2.3 Disadvantages with the Toolkit Approaches

As what we have seen in the above examples, with all of the user interface toolkits, the interface designer writes programs in a conventional programming language. The advantage is that the programmer has extensive control and great flexibility in creating the user interface. However, user interface toolkits can become complex, and the programming environments for those such as Microsoft Windows Software Development Kit (SDK), Apple Macintosh Toolbox, and Unix X-Window Toolkit (Xtk) require months of learning for programmers to gain proficiency. The effort to build user interfaces with toolkits is great, the potential for errors or lack of standardisation is high, revisions are time-consuming, and maintenance is difficult.

Toolkits have become popular with programmers, but they provide only partial support for consistency, and designers and managers must still depend heavily on experienced programmers [Shneiderman, 1992]. A toolkit typically includes hundreds of procedures that implement many interaction techniques, and it is often not clear how to use the procedures to create a desired interface [Myers, 1989].

As shown in the example of the Motif widget set, most toolkits today require the programmer to specify call-backs for almost every widget in the interface, and some
widgets even take more than one call-back. For example, the slider widget in Motif has two call-backs, one for when the indicator is dragged and one for when it is released. A typical user interface for moderately complex program will contain hundreds or even thousands of widgets. Many application programs contain over 2000 widgets [Myers, 1991; Hashimoto & Myers, 1992]. This means that the programmer must provide many call-back procedures. To add to the complexity, each type of widget may have its own protocol for what parameters are passed to call-back procedures, and how the procedures access data from the widget. The use of all of these call-backs means that the user interface code and the application code are not well separated or modularised [Myers, 1991]. Thus, changing the user interface can require major changes to the application code because there is not a clean separation between system functions and dialogue tasks. The user interface is not independent of the application but is embedded within it. Therefore, toolkits do not enforce a software architecture suitable for iterative design [Coutaz, 1987].

In addition, it is difficult for a novice to build user interfaces because of the sequential nature of underlying programming [Tatsukawa, 1991]. Toolkits provide only a limited range of interaction objects, limiting the kinds of interactive behaviour allowed between the user and the system. Toolkits are expensive to create and are still very difficult to use by non-programmers. Even experienced programmers will have difficulty using them to produce an interface that is predictably usable [Dix, et al., 1993]. Therefore, there is a need for additional support for programmers in the design and use of toolkits to overcome their deficiencies. The set of programming and design techniques which are supposed to provide another level of services for interface design beyond the toolkits are User Interface Management Systems (UIMSs). The UIMS approaches are surveyed in the following sections.

3.3 Language-Based User Interface Specification

In most UIMSs, the designer specifies the user interface with a special-purpose language; this language may have many forms: menu networks, state transition
networks, context-free grammars, event languages, declarative languages, or object-oriented languages [Myers, 1989]. Many researchers in this area have provided thoughtful reviews of language-based interface specification methods on various UIMSS [Green, 1986; Myers, 1989 & 1993; Hartson & Hix, 1989; Jones, 1991a & 1991b; and Shneiderman, 1992].

3.3.1 Menu Networks

One of the simplest, the most widely used interaction technique, is the menu hierarchy or network. In the menu-based system, the communication between application and presentation is modelled as a hierarchy or a network of menus or sub-menus [Dix et al., 1993]. Menu systems are straightforward to implement. All that is needed is to list the options available and then ask the user to choose from among them [Jones, 1991b]. When the user selects an item on one menu, another menu appears or an action is executed. To control the interaction, the programmer must simply encode the levels of menus and the connections between one menu and the next sub-menu or an action. The menu is used to embody all possible user inputs at any one point in time. Links between menu items and the next displayed menu model the application response to the previous input.

There are two kinds of menus. One is explicit menus or conventional menus, in which there is an orderly enumeration of the menu items with little extraneous information such as simple menus, pop-up menus, pull-down menus, etc. The other is embedded menus, in which the menu items might be embedded in text or graphics and still be selectable [Shneiderman, 1992]. More complex examples are Hypertext [Conklin, 1987] and Hypermedia [Woodhead, 1990]. Embedded menus permit items to be viewed in context and eliminate the need for a distracting and screen-wasting enumeration of items. Contextual display helps keep the user focused on the tasks and on the objects of interest. Items rewritten in a list form (explicit menu) may require longer descriptions (of the items) and may increase the difficulty of making selections.
because of confusion arising from cross-referencing between the menu and the context [Shneiderman, 1992].

The examples of UIMSs which support menu specification are ZOG [Robertson et al., 1981], TIGER/UIMS [Kasik, 1982], etc. ZOG, which is a rapid-response, large-network, menu selection system developed at Carnegie-Mellon University, USA, was one of the best-known hypertext systems of the 1970s. Hierarchically linked, ZOG’s screen-sized frames of information and menu options provided a consistent interface. In the early 1980s, ZOG was networked on the state-of-the-art nuclear aircraft carrier USS Carl-Vinson as a ship-wide information and management system [Akscyn and McCracken, 1984]. The commercial version, KMS (Knowledge Management System), is a distributed hypermedia system [Akscyn et al., 1988]. ZOG has been used as an interface for a command language system, a database retrieval system, a CAI system, a guidance system, an interrogation system and a question-answering system [Jones, 1991 b]. ZOG is based on a hierarchy of sub-nets. Each sub-net is a tree of frames in the form of a database. The system displays the frames to the user. The frames are built using a frame editor, ZED, which can be used at any time by the developer, including when using the system. ZOG is really a hierarchical menu system together with a generalised mechanism for the display of information and the triggering of actions.

The Interactive Graphic Engineering Resource, or TIGER UIMS developed at Boeing Computer Services [Kasik, 1982], is part of large toolkit for the development of highly interactive graphics-based applications [Kasik, 1985]. TIGER supports a sophisticated menu network that supports skipping levels and aborting. This UIMS strictly separates dialogue components from the application by formatting all dialogue sequences for display, managing all end-user defaults within and across sessions, accepting all end-user inputs and handling operating system exceptions [Hartson & Hix, 1989]. The information in the user interface specification is described in a specialised textual programming language called TICCL (Tiger Interactive Command and Control Language), which is designed as an extension of Pascal and is precompiled. TICCL
provides a context that contains links to the application via strict procedure interface. Dialogue specification occurs through a text editor, and an infamous 'copy-and-conquer' technique is used where existing dialogues are modified for new functions [Olsen et al., 1984]. TIGER UIMS is being used in a wide variety of applications, including 3-D geometry construction and manipulation for points, curves, surfaces, and volumes; finite element modelling; drafting and documentation; hierarchical design charts; space station analysis; oil well log history analysis; and interactive design [Hartson & Hix, 1989].

Many hypertext systems, such as Apple's HyperCard program, could also be considered UIMSs that manage menu networks [Myers, 1989].

3.3.2 State Transition Networks

State transition networks (STNs), or state diagrams, provide a graphical means of dialogue specification. A transition network is a set of states. It consists of nodes that correspond to a state and directed arcs that connect nodes. Arcs out of each state are labelled with the input token that will cause a transition to the state at the other end of the arc. In addition to input tokens, the arcs in some systems are labelled with application procedures to be called and with output to be displayed. The earliest use of state transition network for interaction specification is Newman's Reaction Handler in 1968 [Newman, 1968]. It used finite state machines (FSMs), the simplest forms of transition networks, to implement a simple UIMS that handles textual input only.

A large system may have too many nodes to be readily understood and to be manageable, and, furthermore, some parts of the dialogue may be repeated. In order to manage this complexity, further representational power is introduced to transition networks by allowing recursive calls. They are called Recursive Transition Networks (RTNs). RTNs make it possible to divide complex networks into more manageable pieces: sub-networks. The successful completion of a sub-network may be used as a complex conditional on the arcs of another network [Denert, 1977]. Sub-networks
facilitate the design and understandability of larger systems and allow re-use of common dialogue fragments, and with recursion transition networks can handle more complex dialogues [Jones, 1991a].

*Augmented Transition Networks* (ATNs) is another enhancement form of STNs. The ATNs have been used to analyse natural language structure [Woods, 1970]. Current research has progressed far beyond this point, but these ideas were quite novel when first proposed [Hartson & Hix, 1989]. In the ATNs, the system is assumed to hold a set of registers: global data structures that the transition network can manipulate. In the STNs, the arcs are labelled with the action that causes the transition and the system response. In the ATNs, however, the arcs have a condition as well as the action. The condition can refer to the system's global data structures and the arc is only followed if the condition is satisfied and the action occurs. The system response is augmented to include not only feedback and display, but also the setting of the global data structures. These data structures can be used simply to describe more complex dialogues and can also be used to communicate with the application [Dix et al, 1993].

Jacob's State Diagram Specification Interpreter [Jacob, 1985] is based greatly on semantic, syntactic, and lexical levels, using separate diagrams for each level. State transitions are associated with either an input or an output token, but not both simultaneously. That is, output is treated as a separate token, rather than as a special action, allowing representation of output dynamics. However, this can increase the number of states in a diagram making it more difficult to read. In Jacob's system the arcs can have recursive calls to other sub-networks or sub-diagrams. Those sub-diagrams are used to control complexity by providing modularity and an ability to decompose designs into levels of abstraction [Hartson & Hix, 1989]. Figure 3.3 shows a transition network for a simple desk calculator created with Jacob's State Diagram Specification Interpreter system.

system that is similar to Jacob’s except that it has more powerful output primitives [Myers, 1989]. It is a system for representing not only the user interface but also an entire interactive information system. It is based on graphical specification of required dialogue using transition networks or sub-networks and can rapidly produce either a menu or a form-filling dialogue. Transition diagrams can be produced, using a graphical Transition Diagram Editor (TDE), and then be automatically converted to the skeleton of the textual code, or textually by using a special language entered with an ordinary text editor. Then a Transition Diagram Interpreter (TDI) can be used on the textual code to demonstrate and evaluate the prototype system. Figure 3.4 shows a transition network for a simple menu system from Wasserman’s Rapid/USE [Wasserman & Shewmake, 1985]. Some of its corresponding textual encoding is shown in Figure 3.5.

An interesting extension of transition networks is Cockton’s Generative Transition Networks (GTNs) [Cockton, 1988 & 1990]. The GTNs are equivalent in power to the Augmented Transition Networks (ATNs), but superior in their economy [Cockton, 1990]. The key contribution of GTNs is to simplify the specification of responses to globally enabled events within a sequential network formalism. Cockton [Cockton, 1990] claims that without GTNs, the designer is forced to choose between abstractions which are good at sequence, but not interleaving, or vice-versa; with GTNs, designers can have both. This is because GTNs encapsulate regularity rather than enumerate it. The generative property of a GTN is reflected in descriptions, which generate arcs rather than describe each of them individually. For example, the following statement:

\[
\text{all : hit\_help} \Rightarrow \text{do\_help};
\]
\[
\Rightarrow \text{same}
\]

declares that at all nodes, if there is a hit\_help event, the response is the do\_help procedure. The traverser is to stay at the same node.

Similarly, the following statement:
all - {a, b, c} : hit_quit => traverse_safe_quit; => same

declares that for all nodes, except a, b, and c, if the hit_quit proposition is true, the response is the traverse_safe_quit actions. The traverser is to stay at the same node.

Figure 3.3 A Transition Network from Jacob's State Diagram Specification Interpreter

Figure 3.4 A Transition Network from Wasserman's Rapid/USE
Figure 3.5 Textual Code for Figure 3.4

State transition networks are also used in Edmonds' SYNICS [Edmonds, 1981] to represent dialogue and global control. Alty's CONNECT [Alty, 1984a] is a front-end to the CP/M and MS-DOS operating systems and is based on transition networks together with a production rule system which enables the network connectivity to be altered to
provide an adaptable interface. Many others [Denert, 1977, Dwyer, 1981 & Green, 1981] have also used transition networks to represent the human-computer interface.

Transition networks are well suited to the sequences of simple dialogues [Cockton, 1985]; and state transition UIMSs are most useful when the interface must do a lot of syntactic parsing or has many modes (each state is really a mode) [Myers, 1989]. However, the transition network approach has not been successful for those highly interactive systems that are largely mode-free, and where the user has many choices at every point. This requires many arcs out of each state. Another problem with this approach is that state transition networks cannot handle the interfaces that let the user operate on multiple objects concurrently (possibly using multiple input devices). Also, the networks become awkward when used for large interfaces, but Alty [Alty, 1984b] has proposed the use of graph theoretic analysis techniques (path algebra) to overcome this problem.

Jacob's Interaction Object System [Jacob, 1986] addresses the above problems, but tries to retain the clarity of state transition networks. The system combines state transition networks with a form of event language. It lets multiple diagrams be active at the same time and transfers the control flow from one to another in a coroutine fashion. Jacob's system also provides simple inheritance to reduce repetition in specification of common aspects of hierarchically related objects. In particular, this system can be used to create direct manipulation interfaces.

Additionally, statecharts [Harel, 1988] are a hierarchical extension of conventional state transition networks and are well suited for specification of complex user interfaces. Statemaster, which was developed by Wellner [Wellner, 1989] is a statechart-based UIMS. The three fundamental components of these statecharts adapted in Statemaster are states, events, and actions. Behaviours are made up of event-action pairs. The most important feature that statecharts extended to the transition networks is the introduction of two new kinds of states to represent the concepts of grouping and concurrency.
In statecharts, several states, which have identical transitions triggered by the same event, can be grouped together to form a single transition. A grouper state is a XorGrouper state, which means that when it is active, one and only one of its children is active, or an AndGrouper state, which means that when it becomes active, all of its children become active. Statecharts use the AndGrouper states to specify concurrency. XorGroupers effectively overcome the explosion of transitions that occurs in conventional state networks, while AndGroupers make a statechart representation grow linearly when a conventional representation grows exponentially [Wellner, 1989].

The statechart dialogues interact with the outside world entirely through events (for input) and actions (for output). Events can be generated by the I/O drivers, interaction objects, application software, or by the broadcast action, which is used to specify the communications between disconnected dialogues. Actions implement all side effects of state transitions. Any state can have optional entry actions that are executed whenever the state is entered and exit actions that are executed whenever the state is exited. The addition of entry and exit actions is the most significant difference between Statemaster's and Harel's statecharts. Conditional actions are used to check the values of specific variables or the active status of specific states to determine which action to execute, and to guard the entrance of a particular state, impose constraints on the dialogue and to handle exceptions. Every state can also have behaviours. A behaviour is simply an event-action pair that specifies an action to be executed when a specific event occurs.

### 3.3.3 Context-Free Grammars

A formal grammar is a tool that allows the structure of the human-computer interaction to be described and analysed as a language [Jones, 1991a]. The formal grammar is specified as a set of production rules that define non-terminal symbols in high-level components and specific symbols as terminals. Whenever a non-terminal symbol appears, it may be replaced by substituting the contents of its definition. The specification is normally based on Backus-Naur Form, or Backus Normal Form.
CHAPTER 3 INVESTIGATIVE STUDY OF USER INTERFACE TOOLS

(BNF), but pure BNF needs to be modified so that an action is associated with each production rule to cope with the human-computer interaction [Lawson et al., 1978, Reisner, 1981 & Shneiderman, 1982]. Such actions might include prompts and other feedback to the user, as well as to the application. The grammars in this class are usually context-free, so it is common to use context-free grammar instead of BNF.

Most grammar-based UIMSs are based on parser generators. For example, the designer might specify the interface syntax in BNF. Grammar-based systems are good for describing command-based interfaces, but are not so good for more graphically based interfaces [Myers, 1989]. Also it has proved harder to use than state transition networks [Guest, 1982], and has difficulty in providing control of interaction [Jacob, 1984]. Although these two techniques are equivalent in power, designers seem to prefer state transition approaches [Guest, 1982]. The reason is that STNs provide more comprehensible language representations because they show the time sequencing and the surface structure of user interface more directly than BNF does [Jacob, 1983].

SYNGRAPH (SYNtax directed GRAPHics) [Olsen and Dempsey, 1983] generates user interface for interactive graphics applications. Input to SYNGRAPH is a description written in a formal grammar using an extended BNF. From this description, a parser automaton and a standard interface presentation (with menus, textual input, and a few predefined interaction devices: locator, valuador, pick, etc.) can be generated in Pascal code, and subsequently refined by the designer to create the final interactive program. SYNGRAPH can deal with semantic error recovery, "Cancel" and "Undo" at the semantic level. However, it does not provide semantic feedback or defaults because there is no way for application routines to affect the parsing [Myers, 1989].

Edmonds and Guest [Edmonds & Guest, 1978] have devised a notation for representing translation of user-computer interfaces. The dialogue is specified in an extended BNF-like, or context-free, style. The significant feature that extends the range beyond context-free grammars is an "if...then...else" construct. In their SYNICS (SYNtax and semantICS) system [Edmonds, 1981, Guest, 1982], the initial idea was
to use their BNF-like language to specify the dialogue. The system produced a table-driven, top-down recogniser that accepts strings of text from the user. Subsequently this was augmented with a transition network front-end (DDL: Dialogue Definition Language). In adding the transition network front-end the BNF is now associated with arc conditions and therefore combines the expressive power of the string matching of BNF with the more attractive features of transition networks [Jones, 1991b]. Further work has produced SYNICS2 [Edmonds & Guest, 1984].

Traditional grammar only describes the user's input. To accommodate the richness of interactive software, it should also describe the output from computer [Shneiderman, 1982]. Shneiderman's *multiparty grammars*, one variant of BNF, adapted specifically to represent interaction languages rather than static ones. The significant extension from the standard BNF is that multiparty grammars have non-terminals labelled with a party (typically the user, labelled by "U:.", or the computer, labelled by "C:"). Non-terminals acquire values during parsing for use by other parties, and therefore error-handling rules can be included easily [Shneiderman, 1992]. In the grammar, unlabelled non-terminals describe complex actions involving one or more parties. For example, the following grammar describes the opening steps in a log-in process:

\[
\begin{align*}
\langle \text{Session} \rangle &::= \langle \text{U: Opening} \rangle \ \langle \text{C: Responding} \rangle \\
\langle \text{U: Opening} \rangle &::= \text{LOGIN} \ \langle \text{U: Name} \rangle \\
\langle \text{U: Name} \rangle &::= \langle \text{U: string} \rangle \\
\langle \text{C: Responding} \rangle &::= \text{HELLO} \ \langle \text{U:Name} \rangle 
\end{align*}
\]

Multiparty grammars are quite effective for text-oriented command sequences that have repeated exchanges, such as a bank terminal. They can be used with compiler-compilers to generate quickly a prototype and a working system. Unfortunately, two-dimensional styles such as form-filling or direct manipulation and graphical layouts are more difficult to describe with multiparty grammars [Shneiderman, 1992].

In addition, ordinary context-free grammars are inadequate for the interactive, multi-threaded dialogues (interfaces that use multiple input devices concurrently) of modern
interfaces [Scott & Yap, 1988]. Scott and Yap propose a notation for specifying dialogues based on context-free attributed grammars with two extensions: fork operators for specifying sub-dialogues and context attributes for dispatching tokens. For concurrent conversations, the system allows productions to fork off sub-parsers that continue in parallel. For multiple contexts they introduce a special synthetic attribute for tokens that can assist in driving the parse.

3.3.4 Event-Based Systems

In Green's survey [Green, 1986] of three formal classes of techniques for describing dialogue in UIMSs based generally on transition networks, grammars, and events, he concluded that events had the greatest descriptive power, a conclusion consistent with the fact that an event-based mechanism can be used for both sequential and asynchronous dialogue.

In event languages, the user interface is viewed as a collection of events and event handlers [Green, 1985]. Input tokens are considered to be events that are sent immediately to event handlers. These handlers can cause output events, change the internal state of the system, which might enable other event handlers, and execute application routines [Myers, 1989].

The UofA (University of Alberta) UIMS [Green, 1985b] is based on event handlers described in an event language similar to C. The description of a program written in the event language contains at least one event handler definition. From these descriptions, executable event-handling routines are generated automatically in C. Each event handler has its own local variables and procedures for processing events. When an event is received, an event handler executes the associated procedure which may perform calculations, send events to other event handlers and send tokens to the presentation component and application interface model. The skeleton for an event handler description is shown below:

```c
EVENT_HANDLER event_handler_name;
```
CHAPTER 3 INVESTIGATIVE STUDY OF USER INTERFACE TOOLS

TOKEN

token_name event_name;
.
.

VAR

type variable_name = initial_value;
.
.
EVENT event_name:type {
    statements;
}
.
.
EVENT event_name:type {
    statements;
}

INIT statements;
END event_handler_name;

The first part declares the input and output tokens which the event handler can process. The second section consists of declarations of the event handler's local variables. The third section consists of event declarations, each of which consists of one or more C statements that specify the processing to be performed on the events received by the event handler. The final part declares the statements to be executed when the event handler is initialised. The visual part of the interface is represented through use of an interactive layout facility built on a window-based graphics package.

Similarly to the event language described in Green's UofA, ALGAE, A Language for Generating Asynchronous Event handlers [Flecchi & Bergeron, 1987], has been designed for the Seeheim Model. It provides facilities for the specification of multi-threaded, event-driven dialogues. Event handlers reside in each component of the user interface. By supplying a general event-based, message-passing mechanism, ALGAE provides a communications interface for various components and supports concurrency. Information can only be passed between components through the use of
events, so components can be placed in separate processes on different machines. Only the ALGAE runtime queuing system handles the message passing. Figure 3.6 represents a generic user interface constructed using ALGAE [Flecchi & Bergeron, 1987].

Figure 3.6 Structure of An ALGAE-Based User Interface

The Event Response System (ERS) [Hill, 1987b] uses a similar idea but with an entirely different syntax. It provides a model and corresponding language called Event Response Language (ERL) for representing event-based and concurrent dialogues. User-generated events result in specified system response. Complex dialogues can be represented as a set of simpler dialogues running in parallel, using a compact and structured representation technique based on production rules. The ERL has been incorporated into the Sassafras UIMS [Hill, 1986]. In the ERL, local variables called flags are used to help specify control flow. Flags can also be used to store information from input devices and application routines. Events, which are similar to flags, allow communication with output devices and the application.

Squeak [Cardelli & Pike, 1985], a textual language for programming mouse-based interfaces, exploits concurrency. Squeak’s processes are similar to events handlers and the messages sent its processes are similar to events.

In addition, CSP (Communicating Sequential Processes) [Hoare, 1985], like Petri nets [Thiagarajan, 1985] and CCS [Milner, 1985], is a mathematical notation designed for expressing concurrent activities and which can therefore be applied to modern user interfaces. However, it lends itself better to execution than CCS and is more succinct.
than Petri nets [Alexander, 1990]. Using CSP, a system is described by one or more processes. A process is made up of events, where events are atomic and represent some incident or activity in the system. CSP makes no distinction between events caused by a user and events caused by the system.

EventCSP [Alexander, 1990], a subset of CSP, is specially for dialogue design. Unlike "Squeak", which treats the CSP as a programming language and executes the CSP directly, Alexander's system provides a simulator that takes an eventCSP specification and simulates the behaviour of the system that was specified. This eventCSP simulator is the central component of the dialogue design tools, which are collectively known as SPI (Specifying and Prototyping Interaction). The designer first can use the SPI Outliner to "sketch" a plan of a proposed dialogue and the Outliner allows these outlines, written in eventCSP, to be executed on the simulator. Then the effects of events can be made more explicit by the designer using a SPI Scenarios tool to create "scenarios" (which can be thought of as "slide shows") of what the screen will look like to the user. At this stage, there is no need to link the dialogue or screen layout to the application. All that is required is that a picture of the screen be associated with each event in the eventCSP specification. These specifications can be tested using the SPI Scenarios tool.

EventCSP offers a formal, expressive and powerful language for dialogue structure, capable of specifying the concurrent dialogues that are increasingly a part of modern user interfaces. However, eventCSP has certain drawbacks. There is no mechanism for ensuring that scenarios or event specifications are consistent with their associated CSP specification. Its inability to express the screen presentation may be a significant flaw [Alexander, 1990].

Event-based systems are explicitly designed to handle multiple processes. People can be more effective when they operate multiple input devices concurrently [Buxton & Myers, 1986]. It is easier to use multiple processes to program multiple interactions where the user can choose which interaction to use. The disadvantage of event-based
systems is that it is often very difficult to create the correct code because the control flow is not localised. Small changes in one part of the program can affect many other parts, making such systems difficulty to maintain [Myers, 1989].

3.3.5 Declarative Approaches

The declarative approach tries to define a language that is declarative (stating what should happen) rather than procedural (how to make it happen). COUSIN [Hayes et al., 1985] allows the designer to specify user interfaces in this manner. The user interfaces supported are basically forms where fields can be text-typed by the user, or options selected using menus or buttons. There are also graphic output areas for use by applications. The application program is connected to the user interface through global data (slots), which can be set and accessed by both the application and the interface.

3.3.6 Object-Oriented Approaches

The common feature of the object-oriented systems is that they provide an object-oriented framework in which the user interface is programmed. There are high-level classes that handle the default behaviour and the user interface designer specialises these classes to deal with behaviour specific to the interface [Myers, 1989]. The characteristics of object-oriented programming - encapsulation (data abstraction and information hiding), dynamic binding, inheritance of attributes and procedures, and message passing - make it a likely platform for direct-manipulation interface and asynchronous control [Hartson, 1989]. These systems can handle highly interactive, direct-manipulation interfaces because there is a computational link between the input and the output that the application can modify to provide semantic processing [Myers, 1989].

The advantages of object-oriented systems include higher productivity because the code can be shared and reused; decoupling of representation from implementation; reliability, consistency, and locality of definition from inheritance; and low code bulk [Hartson & Hix, 1989; Hartson, 1989]. Although these systems make it much easier to create user
interfaces, they are programming environments and therefore are inaccessible to nonprogrammers [Myers, 1989]. Other disadvantages include a steep learning curve for programming and a high performance penalty for systems because they are interpretative and use dynamic binding and message passing [Hartson & Hix, 1989; Hartson, 1989].

MacApp [Shmucker, 1986] is a software system that provides an overall application framework and uses the object-oriented language Object Pascal. Classes are provided for the important parts of an application, such as the main windows, the commands, etc., and the programmer specialises these classes to provide the application-specific details, such as what is actually drawn in the windows and which commands are provided. MacApp has been very successful at simplifying the writing of Macintosh applications [Myers, 1993].

Unidraw [Vlissides & Linton, 1990] uses a similar approach, but it is more specialised for graphical editors. This means that it can provide even more support. Unidraw uses the C++ object-oriented language and is part of the InterViews [Linton et al., 1989] system. Unidraw has been used to create a drawing editor, a user interface builder (Ibuild), and a simple schematic capture system [Vlissides & Tang, 1991; Linton, 1993].

GWUIMS (George Washington University UIMS) [Sibert et al., 1986 & 1988] is an object-oriented UIMS that uses object-oriented Lisp and provides a classification of interface operations and objects that fit into each class. GWUIMS extends the object-oriented paradigm so that the developer's view, if not the user's view, is also object-oriented. It uses a hybrid approach in which object orientation is strongest at the top levels and the lowest level objects are coded in an efficient conventional programming language [Hartson, 1989]. Higgens [Hudson & King, 1986] adds a structured-data description that supports Undo and Redo and lets the UIMS automatically manage the recalculation and redisplay of objects intelligently.
3.4 Direct Graphical User Interface Specification

The current direction in applications and in UIMS tools is certainly toward graphics user interfaces with direct manipulation [Shneiderman, 1983]. In recent years, the new technique, *direct graphical specification*, has emerged for constructing the user interfaces. In direct graphical specification, as it is known, the designer defines the user interface, at least partially, by placing objects on the screen interactively using a pointing device (usually a mouse) [Myers, 1989 & 1993]. The motivation for this approach is that because the visual presentation of the user interface is so important, a graphical tool is the most appropriate way of specifying it. Therefore, this technique seeks to lighten what Cardelli [Cardelli, 1988] calls the *artistic, polishing* and *programming* burdens. The advantage of this approach is that it is usually much easier for the designer to use. Many of these systems can be used by nonprogrammers [Myers, 1989 & 1993].

3.4.1 Creating User Interface by Direct Manipulation

One group of graphical specification systems, usually called *graphical interface builders*, lets the designer create dialogue boxes that are to be part of a larger user interface. They allow the designer to select from a predefined library of interaction techniques, or widgets in modern windowing systems, and place them on the screen in direct manipulation manner. Other properties of the widgets can be set using property dialogue boxes. They may also support for sequencing, for example by bring up sub-dialogues when a particular button is pressed.

A number of research UIMSs, such as DialogEditor [Cardelli, 1988], vu [Singh & Green, 1988] and Gilt [Myers, 1991] provide graphical facilities for creating the presentation of graphical interfaces. Today, modern UIMSs for windowing systems usually provide graphical interface builders. Examples are the NeXT's User Interface Builder [Webster, 1990], Sun Microsystems' Developer's Guide (Graphic User Interface Design Editor) [Sun Microsystems, 1990], SUIT's (Simple User Interface
CHAPTER 3 INVESTIGATIVE STUDY OF USER INTERFACE TOOLS


DialogEditor [Cardelli, 1988], one of the earliest user interface builders, can be used to create interface by direct graphical interaction. In DialogEditor, user interfaces are organised around the dialogue paradigm. Each dialogue is a collection of interactors (buttons, menus, scroll bars, text areas, text lines, etc.). Applications can use one dialogue for basic interface and expose additional dialogues as needed. In DialogEditor, starting with an empty dialogue, one can add instances of various kinds of interactors, move them around and change their size, individually or in groups. For each interactor class there are associated dialogues used to set the attributes (Active, Dormant, ReadOnly, and Passive for text line), and the appearance (fonts, colours, borders, etc.) of interactors of that class. Test and debug modes are provided to test a dialogue's 'look' and 'feel' without embedding the dialogue into an application. In the application code, related procedures have to be attached to the interactor events.

A similar approach has been used in vu (visual user-interface design workshop) [Singh & Green, 1988]. Vu provides a collection of objects which are customised by using direct manipulation techniques and glued together to form a complete interface. A simulation sub-system is provided to have a 'feel' and a 'look' without delay between editing the design and testing it. Vu can, at all times, provide an up-to-date description of the interface and the objects used in the interface. The interactors in DialogEditor are incapable of such self-description.

Like DialogEditor and vu, Gilt (Garnet Interface Layout Tool) Interface Builder [Myers, 1991] allows dialogue boxes and similar user interface windows to be created by selecting widgets from a palette and laying them out using a mouse. More interestingly, Gilt provides a variety of mechanisms to reduce the number of call-back
procedures that are necessary in graphical interfaces. For example, a uniform framework is used for all value filtering from widgets.

The NeXT User Interface Builder's [Webster, 1990] palette of controls (widgets) includes sliders, radio buttons, check boxes, text entry fields, labels, scrolling lists, and buttons. Once the controls have been selected and dragged into place, the designer can choose attributes for each control from an Inspector Panel. The simple actions for sizing, moving, and copying controls enables many kinds of complex user interfaces to be built quickly. Some linkages between controls can be specified by direct manipulation, but usually code must be written in Object-C to complete the system. The majority of commercial interface builders for modern windowing systems are using a similar approach to create dialogues or windows. The descriptions of the created dialogues or windows can be produced in resource script (Macintosh Toolbox, MS-Windows, etc.) or in UIL (User Interface Language for OSF/Motif, etc.) format. Furthermore, many of them provide facilities (for example, Prototyper for Macintosh and the ClassWizard in MS-Visual C++'s AppStudio for MS-Windows) to generate the skeleton code for the created dialogues or windows. However, the designer has to fill the "call-back" code into the skeleton.

Although Direct Manipulation Interface Builders make laying out the dialogue boxes and the menus easier, this is only part of the user interface design problem. For any kind of application that has a graphics area (such as drawing programs, CAD, visual language editors, etc.), Interface Builders do not help with the contents of the graphics pane [Myers, 1993]. An advanced method, demonstrational interfaces, has emerged to overcome these problems.

3.4.2 Creating User Interface by Demonstration

Demonstrational user interfaces provide concrete examples on which the user operates, rather than requiring the user to deal with abstractions such as variables and control structures. A "demonstrational interface" watches while the user executes conventional
direct manipulation actions, but creates a general abstraction from the specific example. For instance, the user might drag a file named "foo.ps" to the trash, and then a file named "bar.ps", and a demonstrational system might automatically create a macro to delete all files that end in ".ps" [Myers, et al., 1991].

An interface builder is called demonstrational, if this technique is used in creating direct manipulation user interfaces. Demonstrational interface builders allow the creation of the complete user interface in direct manipulation and demonstrational manner without programming. The designers create a layout (presentation) by direct manipulation of graphical objects and demonstrate to the interface builder how the end user might interact with the interface by giving an example of the interface in action (behaviour). From this demonstration, the interface builder automatically infers the general operation of the interface and produces an implementation of it.


Peridot (Programming by Example for Real-time Interface Design Obviating Typing) [Myers, 1987, 1988 & 1990] is the system pioneered to support demonstrational interface creation. Peridot enables interface designers to create dynamic interaction techniques by demonstration. It allows new, custom widgets to be created without programming. It guessed (inferred) alignment of graphical objects using global rules. Peridot places the emphasis on providing low level control of graphics and devices to the designer. The primitives that the designer manipulates with the mouse are rectangles, circles, text, and lines. The system generalises from the designer's actions to create parameterised, object-oriented procedures like those that might be found in toolkits (such as menus, scroll bars, windows). Peridot can also be used by non-programmers.
Lapidary (Lisp-based Assistant for Prototyping Interface Designs Allowing Remarkable Yield) [Myers et al., 1989] extends the ideas of Peridot to allow application-specific graphical objects and behaviours to be specified by demonstration. For example, the designer can draw the nodes and arcs for a graph program.

Garnet (Generating an Amalgam of Real-time, Novel Editors and Toolkits) [Myers et al. 1990], refined from Peridot, is a comprehensive user interface development environment containing many high-level tools, including Gilt, the Lapidary interactive design tool, and others. Garnet also contains a complete toolkit, which uses a prototype-instance object model, constraints, and separation of the behaviours from the graphics. One important goal of Garnet is to allow user interface designers who are not programmers to design and implement the look and feel of user interfaces [Myers et al., 1993].

DEMO [Wolber & Fisher, 1991] uses the idea of demonstrating the end user's action that starts a behaviour (called the "stimulus") and of then demonstrating the response to that stimulus. DEMO II [Fisher et al., 1992] added sophisticated techniques for inferring constraints to control how objects are placed or moved.

Demonstrating, or "showing" in Lewis et al.'s terminology, forms the basis of OSU (Oregon Speedcode Universe) [Lewis et al., 1989]. OSU incorporates a number of domain-specific tools for automatically creating, manipulating, and "playing back" prototypes. The heart of OSU consists of three tools for graphically constructing prototypes: ResDez (the resource designer), the graphical sequencer, and the program generator. ResDez graphically creates and edits all user interface objects by direct manipulation. By the designer "showing" (demonstrating) rather than "telling" (giving instruction code via a programming language) the system what to do, the graphical sequencer creates the actions (or behaviours) for transforming elements of interface objects from one state to another. The program generator automatically writes "compilable" Pascal source code equivalent to the interface prototype defined by the ResDez and the graphical sequencer.
Druid (Demonstrational Rapid User Interface Development) [Singh et al., 1990] aims to help create interactive, graphical, direct manipulation user interfaces. Druid follows a demonstrational approach for specifying the presentation as well as the behaviour of the user interface. Similarly to Peridot, Druid actively helps the designer in crafting the presentation component by guessing (inferring) from his/her imprecise actions. When the designer adds a new widget in the presentation component, Druid immediately tries to find other widgets in the interface to which the new widget relates. Once the designer confirms Druid’s guess, Druid automatically executes the guessed relation. In the specification of the interface behaviour, Druid "watches" the demonstration given by the designer and implements the demonstrated behaviour when the end user interacts with the finished user interface. From the demonstration Druid determines the details of the user interface behaviour specification, such as initial set-up, argument requirements for commands, and argument entry sequences. Contrary to Peridot, Druid concentrates on providing high-level facilities for easily and rapidly creating the overall look and feel of the interface.

Marquise (Mostly Automated, Remarkably Quick User Interface Software Environment) [Myers et al., 1993], the newest addition to the Garnet environment, allows virtually all of the user interfaces of graphical editors to be created by demonstration without programming. It ties together all the previous tools, while supporting interactive specification of the entire user interface. In particular, Marquise goes beyond previous tools since it supports creating, editing, and deleting of objects at run time, and allows the overall graphical appearance of interface to be drawn, and the behaviour for object creation, selection and manipulation to be demonstrated. Marquise uses the stimulus-response idea from DEMO, here called "train" and "show", but concentrates on which high-level actions are appropriate and the context of the stimulus [Myers et al., 1993].

Although all of these systems are research prototypes, some of the ideas are beginning to appear in commercial systems [Myers, 1993].
3.4.3 Frame/Card-Based Systems

Another type of graphical specification system organises the interface as a sequence of mostly static pages called frames or cards. Each page might contain text, graphics, and interaction objects, and commands that cause transfer to other pages. Usually, there is a fixed set of widgets to choose from, which were coded by hand in a conventional programming language. Examples of this category of systems include Menulay [Buxton et al., 1983], Trillium [Henderson, 1986], HyperCard for Apple Macintosh.

Menulay [Buxton et al., 1983] allows the designer to place text, graphical potentiometers, buttons, and pictures (from a library or sketched by the designer) on the screen and see exactly what the end user will see when the application is run. Each active item in the display can be linked to a semantic routine, which is invoked when the user selects the item with a pointing device. It also allows shot programs (routines in a conventional programming language) to be typed-in interactively to specify selection behaviour. Menulay automatically generates tables and code of the graphical interface specified by the designer. However, its rigid table-driven structure limits the interaction between the semantic level and the interface, preventing all forms of semantic feedback.

Trillium [Henderson, 1986], which is very similar to Menulay, has been used to design the user panels for photocopiers. One strong advantage over Menulay is that Trillium's cards can be executed as they are designed, because the specification is interpreted rather than compiled. By separating the interaction behaviour from the graphical presentation, Trillium allows the designer to change both in isolation. However, Trillium has limited support for frame-to-frame transitions (since this is rarely needed for photocopiers).

One of the most famous card-based systems is HyperCard. The designer can easily create cards containing text fields, buttons, etc., along with various graphic
decorations. The buttons can transfer to other cards. HyperTalk, a scripting language provided by HyperCard, provides more functionality for buttons.

### 3.5 Automated User Interface Generation

A problem with all of the language-based tools and other UIMS approaches is that the designer must work at very low levels of lexical and syntactic detail about the placement, format, and design of the user interface. This tends to be very time-consuming and error-prone as the designer has to produce detailed interface specifications, often in cryptic specification languages [Singh & Green, 1989]. To solve this problem, a class of UIMS tries to create the user interface automatically from a high level specification of the application semantic description and then allows the designer to modify the generated interface to improve it. This development approach starts with the conceptual, and then analyses, structures, develops, and finally allows modification to the interface design. Examples of UIMS that automate some aspects of this process are MIKE [Olsen, 1986], Mickey [Olsen, 1989], Jade [Vander Zanden & Myers, 1990], UofA* [Singh & Green, 1989 & 1991], DON [Kim & Foley, 1990 & 1993], GENIUS [Janssen et al., 1993] and TRIDENT [Vanderdonckt & Bodart, 1993].

UofA* [Singh & Green, 1989 & 1991] and Jade [Vander Zanden & Myers, 1990] concentrate on creating menus and dialogue boxes. They allow the designer to use a graphical editor to modify the generated interface if it is not good enough. From a simple and high-level description of the commands and a description of the device on which the user interface will be implemented, the UofA* UIMS produces an initial lexical and syntactic design of the interface. The designer can influence the decisions made by the UofA* by providing the user's preferences and by changing the UIMS's defaults. The initial interface can be refined by the designer using the graphical editor vu [Singh & Green, 1988]. Jade (a Judgement-based Automatic Dialog Editor) automatically creates and lays out graphical input dialogues, such as menus, palettes, buttons, and dialogue boxes. It does this by combining a textual specification of the
dialogue's contents, written by an application programmer, with look-and-feel databases prepared by a graphic artist or style expert. Similar to UofA*, the dialogue can then be modified by a graphic artist using a direct manipulation, graphical editor.

MIKE (Menu Interaction Kontrol Environment) [Olsen, 1986] interfaces are described in terms of what they are supposed to do rather than in terms of how events are handled. By adopting a high-level description of the interface, MIKE is able to generate an initial interface from very early specifications of the interface design. The initial interface is menu-oriented and rather verbose, but the designer can change the menu structure, use icons for some commands, and even make some commands operate by direct manipulation. The designer uses a graphical editor to specify these changes. There are two problems in MIKE's model of user interfaces. The first is that the dialogue style is dictated by the UIMS to a large degree. The style could be adapted to a variety of applications but the fundamental style could not be changed. A second problem is that MIKE's view of user interfaces weighed heavily in favour of command-style interactions which the community in general is moving away from [Olsen, 1989].

The Mickey system [Olsen, 1989] was designed as an attempt to overcome these problems.

DON [Kim & Foley, 1990 & 1993] focuses on the automatic organisation and the development of menus and dialogue boxes using various types of knowledge as input and a set of design rules as an engine. DON provides high-level control and expert assistance in the user interface generation process. GENIUS (GENerator for user Interfaces Using Software ergonomic rules) is developed for the automatic generation of user interfaces from extended Entity Relationship (ER) data models and petri net based dialogue description. GENIUS approach is suitable for the generation of animated prototypes of database-oriented applications [Janssen et al., 1993]. Similar to GENIUS, based on an Entity-Relationship-Attribute (ERA) data model and a dialogue dynamic model which provides a Function Chaining Graph (FCG) specifying the data
flow during the task performance, TRIDENT [Vanderdonckt & Bodart, 1993] automatically generates a user interface for business-oriented applications.

One group of UIMS can be characterised as model-based or rule-based systems that automatically generate the user interface by building a declarative description (model) of their presentation and behaviour. They include UIDE [Foley et al., 1988, 1989 & Sukaviriya et al., 1993], HUMANOID [Szekely, 1990 & Szekely et al., 1993], ITS [Wiecha et al., 1989 & Wiecha & Boies, 1990]. We describe them in the following section.

3.6 Model-Based User Interface Development Systems

UIDE (the User Interface Design Environment) [Foley et al., 1988, 1989] emphasises a model-based approach to create, modify, and generate an interface to an application. An application model describes various details of an application interface including application semantics. The description includes the pre- and post-condition of the operations, and the system uses these to reason about the operations, and to automatically generate an interface.

UIDE's application model consists of three layers which describe interface functionality at different levels of abstraction: the highest level captures semantics of application actions, the mid-level captures actions which exist in various interface paradigms (but are independent of any specific application), and the lowest level describes mouse and keyboard interactions designed to capture lexical groupings which define related lexical input and output [Sukaviriya et al., 1993]. Through these layers, designers have different granularity of control over interface features that they can choose for an application. The run-time environment uses the application model to sequence user dialogues based on task descriptions. UIDE's application model allows the mapping of applications to interfaces to be changed without change application actions. The model also allows an application action to be mapped to multiple interface actions.
CHAPTER 3 INVESTIGATIVE STUDY OF USER INTERFACE TOOLS

One interesting part of UIDE is that the designer can apply "transformations" to the interface. These change the interface in various ways. For example, one transformation changes the interface so that it has a currently selected object instead of requiring an object to be selected for each operation. Another provides functions specialised for various types ("delete-square" and "delete-triangle"), rather than a general-purpose function ("delete"). UIDE applies the transformations and ensures that the resulting interface remains consistent.

Within recent years, UIDE's support capabilities of both design-time and run-time have been strengthened. The design assistance includes the automatic layout of menus and dialogue boxes, and transformations on interface styles using a compositional model of the user interface. The run-time support includes using the pre- and post-condition to control interface objects, automatic generation of animated help, and automatic generation of textual help. If the specifications in the knowledge base change, the generated interface and help at run-time will change accordingly [Sukaviriya et al., 1993].

HUMANOID (High-level UIMS for Manufacturing Applications Needing Organised Iterative Development) [Szekely, 1990] is a model-based interface design and construction system where interfaces are specified by building a declarative model of their presentation and behaviour. A standard run-time support module is included in every application to execute the model, that is, to construct the application displays and interpret input according to the information in the model. The declarative modelling language provided by HUMANOID factors the design of interfaces into five semi-independent dimensions. Application semantics: represents the objects and operations of an application. Presentation: defines the visual appearance of the interface. Behaviour: defines the input gestures (e.g. mouse clicks) that can be applied to presented objects, and their effects on the state of the application and the interface. Dialogue sequencing: defines the ordering constraints for executing commands and supplying inputs to commands. Action side-effects: defines actions executed
automatically when commands or command input change state (e.g., making a newly created object the current selection).

HUMANOID's application and dialogue sequencing models are similar to UIDE's, but UIDE's presentation model is relatively impoverished compared to HUMANOID's. UIDE does not have a sophisticated designer's interface, so its usability is low. UIDE emphasises model analysis tools such as consistency checkers and keystroke analysis. Since HUMANOID's and UIDE's models are similar, the complementary advantages of both systems could be integrated in a single system [Szekely et al., 1993].

ITS (Interactive Transaction Systems) [Wiecha et al., 1989 & Wiecha & Boies, 1990] can also be characterised as a model-based system. In ITS, interactive programs are defined using three kinds of abstraction: data definitions, to define the data manipulated by the program, content tree, to define the dialogues in an abstract way, and style rules, to map the content trees to graphical objects. ITS is being used to implement the visitor information system for EXPO 1992 [Wiecha & Boies, 1990]. Unlike the other systems mentioned above, the designer in EXPO has to write some of the rules used to generate the interface, rather than just writing a specification that the rules work on. In particular, the design philosophy of ITS is that all design decisions should be codified as rules so that they can be used by subsequent designers, which will hopefully mean that interface designs will get easier and better as more rules are entered. As a result, the designer should never use graphical editing to improve the design, since then the system cannot capture the reason why the generated design was not sufficient [Myers, 1993]. ITS's presentation model is similar to HUMANOID's, but its dialogue and application models are less expressive [Szekely et al., 1993].

While the idea of having the user interface generated automatically is appealing, this approach still needs to do further investigation, because the user interfaces that are generated are generally not good enough. A further problem is that the designer must often use special-purpose specification languages, which can be difficult to learn and use properly [Myers, 1993].
CHAPTER 3 INVESTIGATIVE STUDY OF USER INTERFACE TOOLS

3.7 Summary

This chapter gives the survey on user interface tools in two aspects: user interface toolkit approaches and UIMS approaches. First, the interface toolkits are surveyed. The examples of the most popular toolkits today are X-Window Toolkit (Xtk) for workstations, Toolbox for Apple Macintosh and Software Development Kit (SDK) for MS-Windows. One of the motivations for the toolkits is that the final interface will have a "look and feel" consistent with others created using the same toolkits. Although there are many similarities between the various toolkits, people still need to do a lot of work to convert their software from one windowing system to another. A number of virtual toolkits have been developed to try to hide the differences among the various toolkits and to overcome the problem. Two examples of virtual toolkits are described: XVT (Extensible Virtual Toolkit) and SUIT (Simple User Interface Toolkit).

The advantage with the toolkits is that the programmer has extensive control and great flexibility in creating the user interface. However, there are lots of disadvantages as described in this chapter: they are expensive to create, difficult to use and to maintain, complex, and there is a lack of standardisation and no well separated code for the user interface and the application, etc. Therefore, there is a need for additional support in the design and use of toolkits to overcome their deficiencies. Beyond the toolkits, UIMS approaches provide another level of services for the user interface design. Then, UIMS approaches are surveyed in great detail following the toolkit approaches.

UIMS approaches are classified into four groups: language-based specification, direct graphical interface specification, automated interface generation and model-based user interface development systems. Earlier systems are usually language-based. Systems such as Robertson et al's ZOG, Kasik's TIGER/UIMS, Newman's Reaction Handler, Jacob's State Diagram Specification Interpreter, Olsen's SYNGRAPH, Edmonds' SYNICS, Green's UofA, Flecchi's ALGAE, Hayes' COUSIN, etc are described in the section of language-based specification. Object-oriented approaches have become
the fashion in the user interface design and specification. One of the well-known systems that provide an overall application framework and use the object-oriented language is MacApp. The advantages of object-oriented systems are: higher productivity because code can be shared and reused; decoupling of representation from implementation; reliability, consistency, and locality of definition from inheritance; and low code bulk. The disadvantages include a steep learning curve for programming, a high performance penalty for systems due to the fact that they are interpretative and use dynamic binding and message passing, etc.

With the appearance of direct manipulation interfaces, direct graphical specification has emerged for constructing the user interface in direct manipulation manner. In direct graphical specification, the designer defines the user interface by placing an object on the screen interactively using a pointing device without programming. A number of research UIMS (such as DialogEditor, vu, Gilt) and commercial systems (such as UIMX, Prototyper, MS-Visual C++) provide graphical facilities called interface builders for creating the presentation or appearance of the user interface. However, the designer has to provide the code for the behaviours of the user interface. An advanced method - demonstrational interfaces - has emerged. Demonstrational interface builders allow the creation of the complete user interface in direct manipulation and demonstrational manner without programming. The system pioneered to support demonstrational interface creation is Myers' Peridot, which is refined in Garnet and Marquise. Other systems such as OSU, Druid and DEMON are also described in this chapter. Although all of these systems are research prototypes, some of the ideas are beginning to appear in commercial systems.

Sections 3.5 and 3.6 provide the survey for the approaches of automated user interface generation. This kind of UIMS tries to create the user interface automatically from a high level specification of the application semantic description and then allows the designer to modify the generated interface to improve it. Examples of UIMS that automate some aspects of this process are MIKE, Mickey, Jade, UofA*, DON,
GENIUS and TRIDENT. One group of UIMS, which automatically generate the user interface by building a model of the presentation and behaviour (such as UIDE, HUMANOID and ITS), are characterised as model-based systems and they are described in section 3.6. The approach of automated user interface generation still needs to do further research, because the user interfaces that have been generated are not good enough and the languages for the model specification are difficult to learn and use.
CHAPTER 4

THE ABSTRACT INTERACTION OBJECT EDITOR

4.1 Introduction

The Harness UIMS, which was described in chapter 2, has been developed for knowledge-based front-ends. This has been done as part of the ESPRIT project FOCUS (2620), in which an architecture and set of tools have been developed to support the front-ending of existing software systems. FOCUS front-ends typically contain knowledge-based modules, which offer user support in both task related matters and in selecting and using the appropriate existing packages. The Harness is responsible both for creating the user interface and for communications between the various subsystems. This communication takes place using a clearly defined abstract message structure. In the Harness, a set of core Abstract Interaction Objects (AIOs) was provided for knowledge-based modules to interact with the user. It is not realistic to expect the Harness to be a closed and fixed system. For each particular application, it will be necessary to decide, at least, on the presentation style. It may also be necessary to introduce new Abstract Interaction Objects, such as specialis.ed forms for the knowledge-based modules designer to construct a specific user interface and, naturally, it is most convenient to build such information into the Harness. This chapter discusses
the introduction of new AIOs to the Harness. A graphical tool called abstract interaction object editor, which is used for this purpose, is described in detail.

4.2 Programming the Harness

A number of approaches are possible here to introduce new objects to the Harness. Firstly, the conventional approach is that the designer can be given access to the source code of appropriate parts of the Harness and may, therefore, modify it or add new interaction object structures to meet his or her requirements. In this case, the designer, at least, has to know the Harness structure as well as its programming environment. Obviously, it is a boring and time-consuming task to read and to analyse a big complex software system.

Secondly, the knowledge-based modules could introduce a new interaction object to the Harness by sending it messages, which contain the source Prolog code to define the specific interaction object. The Harness stores the code into the internal knowledge base, and afterwards a knowledge-based module can access the interaction object by sending a message with the name and parameters of the relevant predicate to the Harness. For example, a knowledge-based module sends the following message to the Harness to define an interaction object called "inform_user":

```
message(mess_id1,
    [define_object,
     
     /** the presentation definition **/ 
     
     inform_user(information:Information):-
       feh_root(Root), 
       makepopup(top, toplevelshell, Root, 
         [title:'Information to User'], Top), 
       makewidget(form, formwidget, Top, [width:300, height:200], Form), 
       makewidget(label, labelwidget, Form, [labelString:Information], 
         Label), 
       makewidget(sep, separatorwidget, Form, [leftAttachment:form, 
```
 Afterwards a knowledge-based module can send the following message to the Harness to access the interaction object shown in Figure 4.1:

```
message(mess_id2, [call_object,
    inform_user(information:'The system is running ...')], user).
```
CHAPTER 4 THE ABSTRACT INTERACTION OBJECT EDITOR

Compared with the first method, this approach is much better for the designer. The designer does not have to know the implementation details of the Harness, but the knowledge of the relevant parts of the Harness programming environment, e.g. the Prolog predicates linked with X-window system and Motif widget set, is still needed.

Finally, a specific graphical programming tool called an abstract interaction object editor (AIO Editor) could be provided for the designer to build new interaction objects and to introduce them to the Harness by direct manipulation. Using the AIO Editor, the designer can dynamically create, modify, and maintain interactive, graphical, direct manipulation user interaction objects without doing any programming and without involving in the implementation details and the programming environment of the Harness in the conventional method. In doing so, we extend a module, which is called AIO Knowledge Base, into the FOCUS KBFE architecture; and, of course, the relevant program code has to be modified to accommodate this extension. This modification will be discussed in the next chapter. The whole architecture of this approach is shown in Figure 4.2.

![Diagram](image)

Figure 4.2 Using AIO Editor to Introduce New Objects to the Harness

The abstract interaction object knowledge base contains the presentation definitions as well as the behaviour definitions of interaction objects that are created by the designer.
through the interaction object editor. Once an interaction object is created and extended into the object knowledge base, any knowledge-based module can quote it as one of the core abstract interaction objects by sending a relevant message to the Harness and in turn the Harness accesses the object knowledge base to create and to display the interaction object on the screen. A new AIO can also be used as a component in a complex object.

In the rest of this chapter, we discuss the interaction object editor in detail. The architecture of the AIO Editor is described in section 4.3. Section 4.4 and section 4.5 discusses how to use the editor to create the definitions of the presentation and the behaviour of an interaction object respectively. A simulator used to test the behaviour of a new AIO under the Harness programming environment is discussed in section 4.6. Section 4.7 focuses on the structure of the interaction object knowledge base. Section 4.8 provides the comparison of the AIO editor with other similar tools. The summary and conclusions are given in the last section.

4.3 The Architecture of the AIO Editor

As shown in Figure 4.3, the interaction object editor is composed of six components: editor manager, presentation editor, behaviour editor, simulator, knowledge base manager and physical presentation mapper.

According to the designer's commands, the editor manager dispatches relevant components to work in co-ordination. The presentation editor is used to create the appearance or presentation definition of an interaction object, whereas the behaviour editor is used to specify the behaviour definition of the interaction object. The simulator tests the behaviour of the current edited interaction object under the Harness programming environment. The knowledge base manager is used to manage the object knowledge base such as to load, to save, to delete, to re-name an interaction object etc. The physical presentation mapper maps the logical presentation of a basic object
provided in the editor to the physical presentation of a real object i.e. a widget in the X-window system and OSF/Motif widget set.

![Figure 4.3 Architecture of the AIO Editor](image)

There is also an interaction object knowledge base in the architecture besides the above six components. The object knowledge base contains two parts: the presentation specification knowledge and the behaviour specification knowledge for those interaction objects created by the designer. The structure of the AIO knowledge base is discussed in section 4.7.

### 4.4 Editing the Presentation of Interaction Objects

The presentation editor provides interactive graphical facilities for the designer to create, to edit and to modify the presentation specification of interaction objects. The main purpose is to enable the designer to view the appearance exactly seen by the end user and to have a directly look at the effects, without delay, whilst an interaction object is being edited. That is in a so called WYSIWYG (What You See Is What You Get) strategy. This editor provides a number of basic objects, which correspond to all of the widgets and menu types supported by the OSF/Motif toolkit, for constructing a variety
CHAPTER 4 THE ABSTRACT INTERACTION OBJECT EDITOR

of user interaction objects and menu systems. This editor also supports the designer in the use of the existing interaction objects in the knowledge base as 'basic' objects to rapidly create more complex interaction objects.

Using the facilities provided by the editor, the designer can instantiate and customise the basic objects by a combination of direct manipulation and form-filling techniques. Attributes such as size, position, colour and font for basic objects can be customised by using the direct manipulation techniques. For example, to re-size an object, the designer points at the object, presses the third button and drags the mouse until the object's size becomes the desirable size. Whereas other attributes such as label string, border width and attachment are specified by using form-filling techniques in attribute editing sheets.

![Interaction Object Editor](image)

**Figure 4.4 The Interaction Object Editor**

When using the interaction object editor, the designer sees four windows as shown in Figure 4.4. The editor main window, which is on the top left, is used to give commands to the editor and to display icons of edited interaction objects. The window at the bottom left is the window of the current edited interaction object that shows what the user will see as an end user interface. The tree window on the top right graphically displays the tree structure of basic objects in the current edited interaction object data structure. The tree window will be updated automatically whenever a basic object is
created on or removed from the current edited interaction object window. The window at the bottom right is the resource edit window for a basic object, in this working sheet the designer can select or change resource values.

4.4.1 Creating General Interaction objects

On the menu bar in the main window of the interaction object editor (see Figure 4.4), there is a two-level pulldown menu called 'Objects' which includes all basic objects for the designer to select from. In this sub-section we discuss how to use these basic objects provided by the presentation editor to create a general interaction object. Let us suppose, for example, that we want to generate a user interaction object called 'colour_select' the same as that shown in Figure 4.4.

To create a new interaction object, first the designer selects the 'New' item in the pulldown menu 'Commands' on the menu bar in the main window. Then a top level window, its resource edit window and its object tree-structure window will present on the screen (see Figure 4.5a). The geometry attributes, such as position and size, can be customised by dragging, moving, and shrinking using the mouse. Other resources can be customised by changing the default values in the resource edit window. Having finished the resource customisation, the designer presses the 'Ok' button on the edit window and the top level shell window is adjusted to satisfy the designer's requirements.

Next the designer selects the 'Form' widget from the 'Container' sub-menu of the 'Objects' pulldown menu on the menu bar as a composite object and then points the mouse cursor on the newly created shell window and presses the first button, an instance of form widgets is created - in the meantime its resource edit window appears on the screen and the object tree structure shown in the object tree window is updated (see Figure 4.5b). Similar to the above step, the designer can change its resources.
To create a scrolled list, the designer first selects the 'ScrolledWindow' widget in the 'Container' sub-menu, points the cursor on the form widget and presses the first button on the mouse. A scrolled-window widget is created on the position where the cursor is. The designer can adjust its position by pressing the second button and then moving the mouse around, and can change its size by pressing the third button and then dragging or shrinking. Like the above steps, the other resources can be customised by changing the default values in its resource edit window. Then the designer selects the 'List' widget in the 'Display' sub-menu, points the cursor and presses the first button in the
scrolled-window, an instance of list widgets being created. The designer can specify and modify the colour name list items on the list item setting window (see Figure 4.5c).

Now, the designer needs to create a radio box for three toggle buttons: 'Background', 'Foreground' and 'BorderColour'. Similar to above steps, the designer first creates a 'RowColumn' object with the 'radioBehaviour' value being 'On' on the form widget, then creates three toggle buttons respectively with suitable resource values in the 'RowColumn' widget (see Figure 4.5d).

![Figure 4.5c Creating a Scrolled List Widget Instance](image1)

![Figure 4.5d Creating a Radio Box with Three Toggle Buttons](image2)
To create the 'ColourBar', the designer selects the 'Label' widget in the 'Display' sub-menu, and creates its an instance with 'labelString' value being 'ColourBar' at a proper position in the form widget. Similarly, using above techniques, the designer can adjust its position, change its size and customise its resources. The font and colour can be changed respectively by selecting a font and a colour in the 'Font Setting' and 'Colour Setting' windows (see Figure 4.5e).

Similarly, the designer creates a 'Separator' object on the form widget. The geometry relationship of the separator to other objects or its parent object can be set by changing attributes in its 'Layout Edit Window'. Figure 4.5f shows a separator instance after its geometry relationship and other resources are customised.

The final step is to create two 'PushButtons' - 'Ok' and 'Cancel'. Similar to the above steps, the designer creates the two push buttons (see Figure 4.5g). The remaining things are to specify the push buttons' behaviours, i.e. when these push buttons are pressed what actions happen? The designer can get a behaviour setting window by pressing the 'SetCallbacks' button in the resource edit window to create the behaviour specification. We leave this to be discussed in section 4.5.

Figure 4.5e Setting Fonts
So far, the designer has created the presentation of a user interaction object. Its presentation structure code can be automatically generated and be introduced to the interaction object knowledge base by the designer selecting the 'Save' or 'Save as' item in the 'Commands' pulldown menu (see Figure 4.5a). For example, we save this example as 'colour_select'. All of the basic objects in the user interaction object can be edited again at any time by using the techniques described in the above steps. A basic object's resource edit window can be obtained by the designer holding down the first
button and pressing the second button on the mouse. Also, a basic object can be removed from the current edited interaction object structure by holding down the first button and pressing the third button, and if the basic object is a composite object, its child objects are removed automatically from the structure. More conveniently, the designer can obtain a basic object edit window or remove a basic object by applying the above techniques on a corresponding tree node in the current object tree window.

4.4.2 Creating Variety Menus

Using the menu objects provided by the editor, the designer could create a variety of complex multi-level menu systems in user interaction objects. Currently, the editor provides these basic menu objects as follows:

'MenuBar', 'PulldownMenu', 'PopupMenu', 'OptionMenu' and 'SubMenu' (see Figure 4.6a). In this sub-section, we work through a simple example to show how to create the presentation of a menu system by using this editor.

![Figure 4.6a Creating a Menu Bar Instance](image)

Assuming the designer is creating an interaction object called 'menu_example', which contains a shell widget object and a form widget object on the shell, the next steps are to create a menu bar on the form widget and to create a two-level pulldown menu on the
CHAPTER 4 THE ABSTRACT INTERACTION OBJECT EDITOR

menu bar. To achieve this goal, firstly, the designer selects the 'MenuBar' object in the 'Menu' sub-menu. The designer next points the cursor and presses the first button on the form widget, with the result that a menu bar instance is created with its default resource values. Figure 4.6a shows the menu bar instance after its resources are customised by using the manipulation techniques described in section 4.4.1.

Similarly, the designer selects the 'PulldownMenu' item and creates a pulldown menu pan on the menu bar (see Figure 4.6b).

Figure 4.6b Creating a Pulldown Menu Pan Instance

Figure 4.6c Creating Pulldown Menu Items
The designer could select and create some basic objects as the pulldown menu items such as 'PushButton', 'ToggleButton', 'Label' etc. Figure 4.6c shows the pulldown menu with two menu items, both of which are push buttons.

To create a sub-menu (i.e. a second-level menu) in the pulldown menu, the designer selects the 'SubMenu' item, points the cursor and presses the first button of the mouse on the pulldown menu pan. A sub-menu is created (see Figure 4.6d).

---

Figure 4.6d Creating a Submenu Pan

---

Figure 4.6e Creating SubMenu Items
Then the designer could create some basic objects as the sub-menu items. Figure 4.6e shows the sub-menu with three menu items, which are toggle buttons with radio behaviours.

Besides the above, the designer could also use the techniques and facilities provided by the editor to create other types of menus e.g. option menus and popup menus.

4.4.3 Creating Complex Interaction Objects

Using the existing interaction objects in the knowledge base as new basic objects, the designer could create more complex user interaction objects rapidly. For example, to combine the above two example interaction objects into a new interaction object, the designer first creates a new top shell widget object. Then the designer selects the 'menu_example' item in the 'ObjectBase' sub-menu, creates an instance of the 'menu_example' on the new shell by pointing the cursor and pressing the first button (see Figure 4.7a). Similarly, the designer creates an instance of 'colour_select' on the instance of 'menu_example' (see Figure 4.7b). The designer could edit the basic objects' presentations and behaviours on the newly created interaction object window or add other basic objects to satisfy his/her requirements.

![Image of interaction objects](image-url)

*Figure 4.7a Creating an Instance of 'menu_example'*
4.5 Editing the Behaviours of Interaction Objects

In this section, we discuss how to specify the behaviours of a user interaction object. A user interaction object is the combination of some basic widget objects, so its behaviours are composed of these basic objects' behaviours (here the term 'behaviour', in X window system, refers to a 'callback'). A basic object's behaviour is a combination of a series of basic actions. In this editor, we only need to handle the higher level actions such as getting the current state of an object, sending the state to a KBM or an application and etc., because the lower level actions, such as highlighting a menu item, leaving and entering an object and etc., have been handled in the OSF/Motif widget set. The interaction object editor provides a behaviour setting window for every basic object. The designer could specify the behaviours in a direct manipulation manner without doing any programming. The designer just simply selects a series of actions from the basic action list provided by the editor and gives their parameters according to the system's prompt. The editor will automatically produce an action record for the basic object after the designer finishes the behaviour specification.
In the rest of this section, we give the behaviour specifications of the example described in section 4.4.1. In the 'coloucselect' interaction object, we first specify the behaviour of the colour list's 'defaultActionCallback': when the user selects a colour from the list, the background of the 'ColourBar' will be set to the corresponding colour. Firstly, the designer gets the behaviour setting window for the list object by pressing the 'SetCallbacks' button in its resource editing window. Then, the designer sets the callback type as 'defaultActionCallback' and the edit option as 'Append'. After the designer selects the action 'getListSelection' and presses the 'Ok' button in the behaviour setting window, the parameter window of this action appears on the screen. The designer can select or input a value or a variable for every parameter. The editor will append the action with its parameters into the action record of the list object's behaviour. Similarly, the designer can specify the parameters for the action 'setObjectValues' to set the background of the 'ColourBar'. The behaviours of the colour list object, for example, can be specified as follows:

\[
\text{getListSelection}(\text{colourlist}, \text{getColour})
\]
\[
\text{setObjectValues}(\text{colourbar}, \text{background}, \text{getColour}).
\]

Next, we specify the behaviour of the 'Ok' button's 'activateCallback', e.g. when the user presses the 'Ok' button, a message, which includes the selected colour name in the colour list and the selected toggle item in the radio box, will be sent back to a KBM or an application. Similar to the above, the designer could specify the behaviours of the 'Ok' button as follows:

\[
\text{getListSelection}(\text{colourlist}, \text{getColour})
\]
\[
\text{getRadioboxSelection}(\text{radiobox}, \text{getSelection})
\]
\[
\text{returnMessage}(\text{colour} = \text{getColour}, \text{selection} = \text{getSelection}).
\]

Finally, the designer could specify the behaviour of the 'Cancel' button as follows:

\[
\text{returnMessage('cancel')}
\]
\[
\text{removeDialog}
\]
4.6 Simulation Environment

The simulator of the AIO editor provides a Harness environment for the designer to interact with the defined user interaction objects just as the end user would. The main purpose is to enable the designer to have a direct 'feel' for the user interaction objects' behaviours without waiting for them to be embedded in a KBM or application. Under this simulation environment, all of the basic behaviour actions used in the behaviour action records, except the action 'returnMessage', are exactly executed as embedded in a KBM or an application. The action 'returnMessage' is executed to display the relevant message on the screen instead of sending a message back to a real KBM or an application. The advantage of the simulator is that there is no delay in modifying the design and testing the new interaction objects.

To simulate the current edited interaction object, the designer selects the command 'Simple Simulation' or 'Change Resources' in the 'Simulation' pulldown menu (see Figure 4.4) in the main window of the interaction object editor. The command 'Simple Simulation' displays the current interaction object with its resources exactly as that defined in the presentation editor and enables the designer to interact with it. The command 'Change Resources' substitutes its new resources for its defaults when the interaction object is being displayed on the screen and also enables the designer to interact with it to test its behaviours. To remove the simulated interaction object and to clear out the relevant information, the designer can select the 'Close' menu item in the system menu on the title bar of the simulated interaction object window.

4.7 The AIO Knowledge Base

The AIO knowledge base contains two parts: the presentation specification knowledge and the behaviour specification knowledge. The first represents objects, default attributes of each object, the object hierarchical structure and their relationships in an abstract interaction object. The second describes the actions and relevant parameters performed on the relevant objects in an abstract interaction object.
Figure 4.9 The Interaction Object Presentation Schema

The presentation knowledge of an AIO is composed of a number of the presentation descriptions of the basic objects in the AIO. Each object is described in the general structure: the object presentation schema that is shown in Figure 4.9. The \textit{object\_name}, which identifies an interaction object in an AIO, is specified when the object is being created by the designer using the AIO editor. The \textit{object\_type} defines the type of an interaction object, for example, scroll\_bar, push\_button, or other kind of objects defined in the basic interaction object library. The \textit{parent\_name} defines the parent object where the described object is placed. Therefore, the 'Parent' field identifies the position of the described object in the object hierarchical structure of an AIO and the relationship with other objects. If objects have the same parent, they are at the same hierarchical level. If the \textit{parent\_name} is null, the described object is the top level object.

The 'Attributes' field defines the appearance of the described interaction object, for example, width, height, title, font, etc are specified in this field. In fact, the presentation knowledge of an AIO is represented as a series of Prolog facts in the AIO knowledge base. Figure 4.10 shows the part of the presentation knowledge of the 'colour\_select' AIO described in section 4.4.
to send a message 'cancel' to a KBM or an application and then to remove the interaction object from the screen.

Figure 4.8 shows the behaviours of the 'Ok' button being specified. After specifying the object's behaviours, the designer can send a simulation command to the editor to display and to interact with the user interface object under the Harness UIMS environment.

Figure 4.8 Editing the Behaviours of the 'Ok' Button

The method described here avoids the designer having to remember a lot of relevant details in specifying the action parameters, because the editor provides relevant possible parameter values or variables. More conveniently, the designer can get the relevant information for any basic object in the current edited interaction object by pressing the first button on the corresponding tree node in the object tree window. The editor also allows the designer to modify, to remove or to re-specify an existing action record by the designer using above interaction techniques. The action record code will be extended to the behaviour knowledge base in the AIO knowledge base, while the presentation structure code of an interaction object is extended to the presentation knowledge base.
Similarly, the behaviour knowledge of an AIO is composed of a number of behaviour descriptions of the basic objects in the AIO. An interaction object’s behaviour is described in a general structure: the object behaviour schema that is shown in Figure 4.11. In the schema, the object_name in the 'Object' field identifies the object, which
the behaviours will be performed on. The 'Behaviours' field defines the object's callbacks: call-back types and their actions. Here, a call-back is a series of actions specified by the designer that are executed when an interaction object is operated by the end user. The callback_type in the 'Type' field specifies a call-back type which defines what kind of operations is performed on the object and when the call-back will be executed. The 'Actions' field specifies a series of actions that are performed when the call-back is executed. One object can take more than one call-back. For example, the 'colourlist' object (see Figure 4.5c), which is a list object instance, has two call-backs defined in its behaviour knowledge base: one is the 'defaultActionCallback' which is performed when the user double clicks the mouse button on the list object, and another is the 'browseSelectionCallback' which is performed when the user drags the mouse button on the list object. In fact, the behaviour knowledge of an AIO is stored as a series of Prolog facts in the design knowledge base. Figure 4.12 shows the part of the behaviour knowledge of the 'colour_select' AIO described in section 4.4.

<table>
<thead>
<tr>
<th>Object:</th>
<th>object_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behaviours:</td>
<td></td>
</tr>
<tr>
<td>Type:</td>
<td>callback_type</td>
</tr>
<tr>
<td>Actions:</td>
<td></td>
</tr>
<tr>
<td>Action_name: action_parameters</td>
<td></td>
</tr>
<tr>
<td>....</td>
<td></td>
</tr>
<tr>
<td>Action_name: action_parameters</td>
<td></td>
</tr>
<tr>
<td>....</td>
<td></td>
</tr>
</tbody>
</table>

| Type:     | callback_type |
| Actions:  |             |
| Action_name: action_parameters  |
| ....      |             |

*Figure 4.11 The Interaction Object Behaviour Schema*
| behaviour(object: colourlist,  | behaviour(object: cancelbutton,         |
|       behaviours: [ |       behaviours: [ |
|       type: defaultActionCallback, |       type: activateCallback,         |
|       actions: [ |       actions: [ |
|         getListSelection: [colourlist, getColour], |         removeDialog: [] ]       |
|         printMessage: [getColour],          |       ] |
|         setObjectValues: [colourbar,  |  ]
|         background, getColour]]         |       type: activateCallback,         |
|   ],                                      |       actions: [ |
|   type: browseSelectionCallback,         |         getListSelection: [colourlist, getColour],  |
|   actions: [                             |         printMessage: [getColour],          |
|         getListSelection: [colourlist, getColour], |         getRadioboxSelection: [radiobox, getSelect], |
|         printMessage: [getColour],         |         returnMessage: |
|         setObjectValues: [colourbar,      |         [colour=getValue, select=getValue]
|         background, getColour]]         |   ] |
| ]                                        | ] |

Figure 4.12 The Behaviour Descriptions of the 'colour_select' AIO

The above design knowledge of an AIO is automatically produced when the designer finishes the creation of an AIO using the interaction object editor. The AIO design knowledge base could be accessed by the editor to modify the existing AIO design knowledge, or by the Harness UIMS to implement the AIO in a user interface on an application's requirement or on a simulation requirement from the editor. The Harness UIMS has been extended to accept the new AIO design knowledge as an input, to implement the new AIO in a user interface it describes, and to manage the communication between new AIOs and applications. The implementation details of new AIOs in the Harness UIMS will be discussed in chapter 5.

4.8 Comparison with Other Similar Tools

In recent years, a great deal of effort has been spent on building user interfaces by direct manipulation. A number of the existing UIMSs provide a similar tool - the user interface editor for the designer to program user interfaces by direct manipulation in a visual environment. DialogEditor [Cardelli, 1988], Vu [Singh & Green, 1988] etc.
provide graphical facilities for creating the presentation of graphical interfaces. Using these tools, the designer can easily and rapidly create the appearances of the user interfaces. ALGAE [Flecchia & Bergeron, 1987], Carlsen et al's UIMS [Carlsen et al, 1989] and UAN [Siochi & Hartson, 1989] support a conventional programming language or a cryptic textual notation to specify the behaviours of user interfaces, and STN UIMSs of Jacob [Jacob, 1983 & 1986] and Statemaster [Wellner, 1989] provide a non-intuitive graphical notation. In order to avoid the designer having to remember a large amount of relevant details in specifying the behaviours of user interfaces, Peridot [Myers, 1988], OSU [Lewis et al, 1989] and Druid [Singh et al, 1990] enable interface designers to create the specification of the interface behaviour by demonstration.

Comparing the interaction object editor described in this chapter with those existing similar systems, the major difference is that AIO editor is used not only to build complete user interfaces [Huang & Edmonds, 1993] for applications, but also to extend the UIMS's capability by introducing new interaction objects to it. In addition, other applications or knowledge-based modules can share the new interaction objects to build up their own user interfaces.

Unlike Peridot, the interaction object editor provides higher level control with the designers. Peridot enables the designer to create these basic interface objects such as menus, scrollbars etc. from lower level objects (e.g. rectangle, text string, circle etc.). However, the interaction object editor described in this chapter assumes these basic objects and allows the designer to create new basic objects and complete interfaces from these objects. In addition, the designer can use the new basic objects to create more complex objects or interfaces. In the behaviour specification, Peridot allows the designer to create behaviours for lower level actions such as highlighting a menu item and leaving and entering an object by demonstration. In the AIO editor, on the other hand, these lower level actions are already handled and the designer only needs to give the behaviour specifications for higher events such as pressing a button, selecting a list item etc. Therefore, Peridot focuses on providing lower level control of graphics and
devices to the designer, and the interaction object editor concentrates on providing higher level control for the designer to create interaction objects' presentations and behaviours.

In its functionality, the interaction object editor is more flexible and capable than Druid. Both systems use graphical facilities for creating the presentation of user interaction objects by direct manipulation, support all of the widgets provided by the OSF/Motif toolkit to handle selecting of graphical objects, and allow the simulation of the behaviour specification (in Druid, this function is called 'rehearsal'). However, there are some differences. Firstly, the interaction object editor allows the designer to use these existing interaction objects, which are introduced into the Harness UIMS, to create complex user interaction objects more rapidly. In other words, the set of basic objects is self-extensible when a new interaction object is created. Druid and other existing systems do not support this capability. Secondly, the interaction editor allows the designer to create, to modify and to simulate an object's presentation and behaviour at any time in parallel. It does not need to change one editing mode to another mode. This capability makes the designer use the editor more flexibly. In Druid, for example, the designer has to stop the presentation editor and switch to the behaviour editor when he/she wants to edit the behaviour specification. Thirdly, the presentation output in the interaction object editor is the object tree data structure, which is independent of any systems and languages. In Druid, on the other hand, the presentation output is OSF/Motif's User Interface Language (UIL) description of the presentation component. Finally, the interaction object editor does not support the behaviour specification by demonstration. It supports the specification by selecting a series of actions with proper parameters from the provided basic actions in the direct manipulation manner. Whilst the demonstration is a valuable facility, it is not appropriate in our case because of the fact that, in principle, it is usable by any subsystem and must retain a more general behaviour. To date, we have found it quite convenient to express it as described above. However, there is probably a place for a certain amount of specification by demonstration and we will consider that issue in the future.
4.9 Conclusions

This chapter has presented an approach to extend the Harness UIMS's capability, i.e. by the designer using an interaction object editor (a) to create new specific abstract interaction objects; and (b) to introduce the created objects to the Harness. In doing so we have extended a module, the AIO Knowledge Base, into the FOCUS KBFE architecture. The AIO Editor has been implemented and presented in detail in this chapter. By using the approach and the editor, the designer could dynamically create, modify, and maintain interactive, graphical interaction objects and extend the interaction object knowledge base by direct manipulation without doing any programming and without becoming involved in the implementation details and the source code of the Harness UIMS. The conventional method for maintaining and extending an existing software system is boring and time-consuming, this approach solves the problem. The editor described in this chapter provides a more flexible, rapid and effective means for the designer to specify and test the presentations and the behaviours of user interaction objects. Also, it could be easy for the novice designer even though the end user, by using the editor, to design his/her own interaction objects.
CHAPTER 5

IMPLEMENTATION OF THE NEW AIO MANAGEMENT

5.1 Introduction

As has been mentioned in the previous chapter, the designer can create new AIOs by using the AIO Editor, and then extend them into the AIO Knowledge Bases. The Harness, however, must have the capabilities and functionalities to manage the extended new AIOs. For the core AIOs, the Harness manages them individually in different syntax and data structures. For the extended new AIOs, obviously, it is impossible to manage them individually, because the appearances of the new AIOs are in a large variety. It is necessary to provide the Harness with a general method to manage all extended new AIOs. Based on the analysis of the implementation of the Presentation Layer, this chapter intends to define a general purpose syntax using BNF (Backus-Naur Form) grammar for the description of all new AIOs. Then the implementation of the new AIO management in the Presentation Layer is detailed.

5.2 Analysis of the Implementation of the Presentation Layer

As we have established, the Presentation function is divided into two components: the Dynamic Presentation Manager (DPM) and the Physical Presentation Layer (PPL). The DPM is responsible for realising AIOs. It checks the AIO syntax, which is defined in a
BNF grammar. It is responsible for the logical creation, modification, destruction and management of AIO data structures. It passes the AIO data structures to the PPL for the physical visual realisation and updating. The PPL is responsible for the visual realisation of AIOs on screen and deals directly with the window system. It receives AIOs for realisation in the form of C data structures. When the interaction between the end-user and the AIOs is complete, the PPL revises the data structures to reflect the interaction and return them to the DPM.

5.2.1 BNF Specifications of Core AIOs

A core AIO is referenced in a message using the following syntax:

\[
\langle\text{AioClass}\rangle (\langle\text{AioId}\rangle, \langle\text{AioContents}\rangle, \langle\text{AioAttributes}\rangle)
\]

Classes of AIO are defined in the static Presentation Knowledge Base. Each AIO class definition specifies the permissible contents and attributes of the AIO, together with information about the mapping of contents and attributes to interaction primitives.

The AIOs are classified into two types: standard AIOs and container AIOs. Standard AIOs, which include choice, inform, input, hypertext, question, selection, panel and text, are designed to be components of interaction and must be contained within container AIOs. Container AIOs, which include group and query, support the logical grouping of standard AIOs and are distinguished by their attributes.

The syntax for all messages and AIOs can be specified in a Backus-Naur Form grammar. The standard AIO, for example, is specified as follows:

\[
\langle\text{Aios}\rangle := \[
\langle\text{AioElement}\rangle \\
[ \langle\text{AioList}\rangle ] \\
\langle\text{Container}\rangle
\]
\]

\[
\langle\text{AioList}\rangle := \langle\text{AioElement}\rangle \\
\langle\text{AioElement>, \langle\text{AioList}\rangle}
\]
And the `selection` AIO syntax is described as follows:

```xml
<SelectionAttributes>::=  
  <SelectionAttribute>  
  <SelectionAttribute>, <SelectionAttributes>  

<SelectionAttribute>::=  
  structure:<Structure>  
  label:<Label>  
  limits:(<Lower>, <Upper>)  

<Structure>::=  
  nestedList  
  directedGraph  
  matrix  

<Label>::=  
  <term>  

<Lower>::=  
  <integer>  

<Upper>::=  
  <integer>  

<Choices>::=  
  refresh ([<SelectionList>])  
  update ([<SelectionList>])  

[ <SelectionList> ]  

<SelectionList>::=  
  <SelectionItem>  
  <SelectionItem>,<SelectionList>  

<SelectionItem>::=  
  [<SelectionList>]  
```
CHAPTER 5 IMPLEMENTATION OF THE NEW AIO MANAGEMENT

For the syntax of other standard AIOs, please refer to the Appendix A.1.

5.2.2 Data Structures of Core AIOs

Different core AIOs have different definitions of internal data structures. The Harness manages and maintains them individually. For example, the data structures of the selection AIO are defined as follows:

```c
typedef struct _edges {
    int direction; /* direction of the edge */
    int cycle; /* 0=no cycle; 1=cycle; 2=dummy node */
    struct _dgLayout *thisNode; /* the node to draw the edge to */
    struct _edges *next; /* next node */
} edges_t;

typedef struct _dgLayout {
    char *id; /* id of the node */
    Widgetid_t Handle; /* id of the widget/window */
    int width; /* width of the node */
    int height; /* height of the node */
    int x; /* x coordinate */
    int y; /* y coordinate */
    int hLevel; /* horizontal position */
    int vLevel; /* vertical position */
    int cycle; /* 0=no cycle; 1=cycle; 2=dummy node */
    BOOLEAN drawn; /* TRUE if it has been drawn */
    edges_t *edges; /* list of the edges */
} dgLayout_t;
```
typedef struct _selectionitem
{
    term_t id;    /* id of the selection item */
    Widgetid_t Handle;  /* widget id of the selection item*/
    bstatus_t bstatus; /* untouched or update */
    selection_type_t selection_type; /* input or a selection item */
    union{
        input_t *input_struct;  /* pointer to input structure */
        selection_arg_t *args;    /* pointer to a selection item*/
    } type_of;
    struct _selectionitem *nextacross; /* pointer to its siblings */
    struct _selectionitem *nextdown; /* pointer to its children */
    struct _selectionitem *parent;  /* pointer to its parent */
    dgLayout_t *layoutTree; /* pointer to its layout information */
} selection_item_t;

typedef struct _ParentSize_t
{
    selection_item_t *node;
    struct _ParentSize_t *next;
} ParentSize_t;

typedef struct _selection
{
    term_t id;    /* id of the selection aio */
    Widgetid_t Handle;  /* widget id of the selection aio */
    bstatus_t bstatus; /* untouched or update */
    BOOLEAN replace; /* replace choices */
    term_t label;    /* text label */
    structure_t structure; /* matrix,directedGraph.. */
    limit_t size;    /* permitted number of choices */
    selection_item_t *p_selections; /* pointer to the first item */
    attachment_t *attach; /* how it is attached to other aios */
} selection_t;

For the internal data structure definitions of other standard AIOs, please refer to the Appendix A.4.

5.2.3 Syntactic and Lexical Analysis

Yacc and Lex have been used to generate the parser and the lexical analyser for the input messages in the implementation of the Presentation Layer.

Yacc [Johnson, 1975] is a general tool for generating a parser from a specification that describes the syntactic structures of the user input, together with some program routines which are invoked when non-terminals are recognised. Lex [Lesk, 1975] can
be used to create the lexical analyser required to process the raw user input into lexical
tokens required by the parser.

```
mparse.yac  yacc  y.tab.c
            d_parse.c  cc  d_parse.o
mparse.lex  lex  lex.yy.c
```

*Figure 5.1 Message Parser and Lexical Analyser Generation*

The generation of the FOCUS message parser and lexical analyser in the Presentation
Layer, using Yacc and Lex, is shown in Figure 5.1. The grammar of FOCUS
messages is specified within the file 'mparse.yac', which is processed by the tool
'yacc'. The specification for the tokens used in FOCUS messages is in the file
'mparse.lex', which is processed by the tool 'lex'. Each of these tools produces output
files in C format, and these output files are merged into the file 'd_parse.c'. The file
'd_parse.c' is compiled to produce the object file 'd_parse.o'. Then the object file
'd_parse.o' will be linked with other DPM and PPL routines to form the Presentation
Layer. Refer to the Appendix A.2 and Appendix A.3 for the source code of the files
'mparse.yac' and 'mparse.lex' respectively.

5.2.4 Core AIO Management in the Presentation Layer

There are four kinds of AIO messages to the end user: creation or updating,
destruction, sampling and state request. Once DPM receives a message from the
communication manager via DDM, the message will be checked by the syntactic parser
and the lexical analyser. If there is any syntactic or lexical error in the message, an error
message will be reported to the sender via the communication manager. If there is no
error, the DPM will take further process.

For a creation message, the DPM consults the relevant AIO prototype definitions in the
internal static AIO library, by calling relevant routines, to construct the internal data
structures and to set the default values for all core AIOs in the message. Then the DPM instantiates the AIO prototype definitions as necessary from the contents of the message, and overwrites the default values. The resulting AIO data structures are passed to the PPL. According to the data structures and the values, the PPL, by calling the relevant routines provided by the window system, maps the AIOs into the corresponding physical interaction objects and renders them on the user’s screen. When the user interacts with window objects, the PPL, by the window’s triggering off the calling of the callback routines, revises the data structures to reflect the interactions and passes them to the DPM. The DPM constructs a return message and then sends the message to the sender via the communication manager.

For a destruction message, the DPM finds the relevant AIO data structures, instructs the PPL to destroy the relevant objects on the screen by the PPL calling relevant window routines, and then releases the memories occupied by the data structures. We use the following diagram (Figure 5.2) briefly to illustrate the core AIO management. For the management of other kinds of messages (sampling and state request), we can get the ideas easily from the diagram.

5.3 Implementation of New AIOs

To extend the new AIO management into the Presentation Layer, first we have to specify the general syntax in BNF grammar to describe new AIOs. The message parser (file `mparse.yac`) and the lexical analyser (file `mparse.lex`) need to be modified and to be extended to deal with the new AIOs. Then, the general internal data structures have to be defined to store the new AIO information for maintenance purposes. And lastly, the relevant routines, which deal with the new AIOs, have to be coded and to be extended into the Presentation Layer. The extended routines co-operate with other existing routines to maintain the extended and the existing AIO data structures.
5.3.1 BNF Specification of New AIOs

As the extension of the standard AIOs, new AIOs are referenced in messages using the following general syntax:


A new AIO named NewAioName has been extended in the Presentation Knowledge Base by using the AIO Editor. Its default attributes and behaviours have been defined in the Presentation and the Behaviour Knowledge Bases respectively. The
**AioAttachment** specifies the relationship with another AIO within the same group. Similar to the standard AIOs, new AIOs are the components of interaction and must be contained within container AIOs.

Unlike the standard AIOs, the syntax for new AIOs cannot be given individually. A general-purpose syntax for all new AIOs must be defined. This kind of syntax for new AIOs using the BNF grammar is given as follows:

```
<NewAio>::= new_aio(<AioId>, <NewAioName>,
                  <NewAioAttributeList>,
                  <NewAioAttachment>)

<AioId>::= <term>

<NewAioName>::= <atom>

<NewAioAttributeList>::= [ ]
                          [ [<NewAioAttributes>]]

<NewAioAttributes>::= <NewAioAttribute>
                      [ <NewAioAttribute>, <NewAioAttributes>]

<NewAioAttribute>::= <ObjectName>:<ObjAttributeList>

<ObjectName>::= <atom>

<ObjAttributeList>::= [ ]
                      [ [<ObjAttributes>]]

<ObjAttributes>::= <ObjAttribute>
                   [ <ObjAttribute>, <ObjAttributes>]

<ObjAttribute>::= <ResourceName>:<ResourceValue>

<ResourceName>::= <atom>

<ResourceValue>::= <atom>
                   [ <number> ]
                   [ <string> ]

<NewAioAttachment>::= [ ]
                      [ attachment: <AioAttachment> ]

<AioAttachment>::= independent
                   above(<term>)
                   below(<term>)
                   next_to(<term>)
```

For the complete syntax definitions of all messages and AIOs together with new AIOs, refer to the appendix A.1 of this thesis.
5.3.2 Data Structures of New AIOs

The general data structures have to be defined to meet all of new AIO's requirements.

We list them below:

```c
/* new_aio data structures */
typedef struct _variable
{
    term_t v_name;  /* variable name */
    term_t v_value; /* variable value */
    struct _variable *v_next; /* next variable */
} variable_t;

typedef struct _action
{
    term_t a_name;    /* action name */
    struct _action *a_next; /* next action */
} action_t;

typedef struct _callback
{
    term_t c_type;    /* callback type */
    variable_t *c_variable; /* variable list */
    action_t *c_action; /* action list */
    struct _callback *c_next; /* next callback */
} callback_t;

typedef struct _presentation
{
    term_t p_name;  /* name of resource */
    term_t p_value; /* value of resource */
    struct _presentation *p_next; /* next resource */
} presentation_t;

typedef struct _object
{
    term_t o_name;   /* name of object */
    Widgetid_t o_wid; /* window id */
    int o_type;      /* window type */
    Boolean o_update; /* update flag */
    struct _object *o_parent; /* parent of object */
    struct _object *o_child; /* first child object */
    struct _object *o_brother; /* next brother object */
    presentation_t *o_presentation; /* appearance */
    callback_t *o_callback; /* behaviour */
} object_t;

typedef struct _new_aio
{
    term_t name;    /* name of new aio */
    term_t id;      /* id of new aio */
    bstatus_t bstatus; /* update flag */
    Widgetid_t handle; /* window handle id */
    object_t *p_objects; /* top object */
    attachment_t *attach; /* attachment */
    callback_t *return_actions; /* return date actions */
    term_t return_data; /* return message */
} new_aio_t;
```
5.3.3 New AIO Management in the Presentation Layer

Like a standard AIO, a new AIO can be embedded in a container AIO. The message parser and the lexical analyser must have the capability to deal with the extended new AIOs in the received messages. This kind of capability has been extended into the parser and the lexical analyser. Refer to the Appendix A.2 and A.3 for the complete code of the syntactic parser and lexical analyser, which deal with the new AIOs as well as the core AIOs. The relevant routines, which deal with the new AIOs, have been developed and extended into the DPM and PPL.

![Diagram of New AIO Creation](image)
The management of new AIOs is similar to the core AIOs’. However, there are two major differences. First, when creating a new AIO, the DPM calls the extended routines to load the presentations and the behaviours from the external dynamic new AIO Knowledge Base and to set the default values for the new AIO data structures. For the core AIO creation, the DPM uses the internal static AIO definition library to set the core AIO’s defaults. Secondly, when the user interacts with the window objects, the PPL uses the build-in callbacks to set the core AIO current states with the fixed return message format for each AIO. But, for the new AIOs, the PPL has to call the behaviour actions, which are defined by the designer and are loaded from the AIO Behaviour Knowledge Base, to set the new AIO states and the return message. We use a diagram (Figure 5.3) to illustrate briefly the new AIO creation procedure.

We list some routines, which have been extended into the DPM, as follows to demonstrate how to load the new AIO definitions from the external AIO Knowledge Base.

```c
/* main routine : dpm load new aio from aio kbs */
object_t *dpm_load_new_aio(new_aio_t *newAioPtr)
{
    FILE *fp;
    object_t *top_object = NULL,*return_object;
    char file_path[NAMESIZE];
    char aios_kbs_dir[NAMESIZE];
    char *object_name = newAioPtr->name;
    strcpy(aios_kbs_dir, getenv("AIO_KBS_DIR"));
    sprintf(file_path, .. %s/presentation/%s.obj", aios_kbs_dir, object_name);
    if (fp = fopen(file_path, "r"))
    {
        /* load presentations */
        top_object = dpm_loadTopObject(fp);
        dpm_loadSubObject(fp, top_object);
        fclose(fp);
        /* load behaviours */
        sprintf(file_path, "%s/behaviour/%s.act", aios_kbs_dir, object_name);
        dpm_loadBehaviour(file_path, top_object);
        /* load return actions */
        sprintf(file_path, "%s/return/%s.rtn", aios_kbs_dir, object_name);
        dpm_loadReturnActions(file_path, newAioPtr);
    }
    else
    {
        fprintf(stderr, "new aio:%s doesn't exist!!\n", object_name);
    }
}
```

111
CHAPTER 5 IMPLEMENTATION OF THE NEW AIO MANAGEMENT

dpm_error("dpm_load_new_aio");
}
return_object = top_object->o_child;
top_object->o_child = NULL;
return_object->o_parent = NULL;
dpm_freeObject(top_object);
return(return_object);

/* routines for loading new aio's presentations */
object_t *dpm_loadTopObject(FILE *fp)
{
    Widget_t top_wid = NULL;
    char cchar[2], parameter_type[NAMESIZE];
    char object_name[NAMESIZE];
    int type;
    object_t *top_object;

    /* read the top object parameters */
    fscanf(fp, "%s\n", object_name);
    fscanf(fp, "type: %d\n", &type);
    top_object = dpm_makeNewObject(object_name, top_wid, type);
    fscanf(fp, "parameters:\n");
    /* load parameters for the top object */
    top_object->o_presentation = dprn_loadResourceValues(fp);
    return(top_object);
}

dpm_loadSubObject(FILE *fp, object_t *parent_object)
{
    Widget_t sub_wid = NULL;
    char cchar[2], object_name[NAMESIZE], parameter_type[NAMESIZE];
    int type;
    object_t *sub_object;

    if (!feof(fp))
    {
        fscanf(fp, "%s\n", cchar);
        if (strcmp(cchar, ")") == 0)
        {
            fscanf(fp, "name: %s\n", object_name);
            fscanf(fp, "type: %d\n", &type);
            sub_object = dpm_makeNewObject(object_name, sub_wid, type);
            fscanf(fp, "parameters:\n");
            /* load parameters for the object */
            sub_object->o_presentation = dpm_loadResourceValues(fp);
            dpm_insertObject(parent_object, sub_object);
            dpm_loadSubObject(fp, sub_object);
        }
    }
    if ((strcmp(cchar, ")") == 0) && (parent_object->o_parent != NULL))
        dpm_loadSubObject(fp, parent_object->o_parent);
}

/* routines for loading new aio's behaviours */
action_t *dpm_loadOneAction(FILE *fp)
{
    action_t *current, *last, *return_action;
    char action_string[STRINGSIZE];

    return_action = (action_t *) calloc(sizeof(action_t), 1);
    return_action->a_name = NULL;
    return_action->a_next = NULL;

    return return_action;
}
current = return_action;
last = NULL;
fscanf(fp, "\n");
gets(action_string, STRINGSIZE, fp);
action_string[strlen(action_string) - 1] = '0';
while ((action_string[0] != '0') && (!feof(fp)))
{
    if (current == NULL)
        current = (action_t *) calloc(sizeof(action_t), 1);
    current->a_name = (char *) malloc(strlen(action_string) + 1);
    strcpy(current->a_name, action_string);
    current->a_next = NULL;
    if (last != NULL)
        last->a_next = current;
    last = current;
    current = NULL;
    fgets(action_string, STRINGSIZE, fp);
    action_string[strlen(action_string) - 1] = '0';
}
return (return_action);

dpm_loadBehaviour(char *file_path, object_t *top_object)
{
    FILE *fp;
    action_t *behaviour_action;
    char object_name[NAMESIZE], behaviour_type[NAMESIZE];
    object_t *tmp_object;
    callback_t *tmp_behaviour;
    if (fp = fopen(file_path, "r") )
    {
        while ( !feof(fp) )
        {
            fscanf(fp, "object: %s\n", object_name);
            tmp_object = dpm_name20bject(top_object, object_name);
            fscanf(fp, "behaviour_type: %s\n", behaviour_type);
            behaviour_action = dpm_loadOneAction(fp);
            if ((behaviour_action->a_name != NULL) && (tmp_object != NULL))
                if (tmp_object->o_callback == NULL)
                    {
                        tmp_object->o_callback = (callback_t *) calloc(sizeof(callback_t), 1);
                        tmp_object->o_callback->c_type = (char *) malloc(strlen(behaviour_type) + 1);
                        strcpy(tmp_object->o_callback->c_type, behaviour_type);
                        tmp_object->o_callback->c_action = behaviour_action;
                        tmp_object->o_callback->c_variable = NULL;
                        tmp_object->o_callback->c_next = NULL;
                        tmp_behaviour = tmp_object->o_callback;
                    }
                else
                    {
                        tmp_behaviour = tmp_object->o_callback;
                        while (tmp_behaviour->c_next != NULL)
                            tmp_behaviour = tmp_behaviour->c_next;
                        tmp_behaviour->c_next = (callback_t *) calloc(sizeof(callback_t), 1);
                        tmp_behaviour->c_next->c_type = (char *) malloc(strlen(behaviour_type) + 1);
                        strcpy(tmp_behaviour->c_next->c_type, behaviour_type);
                        tmp_behaviour->c_next->c_action = behaviour_action;
                    }
            }
        }
    }

5.4 Summary

The implementation of core AIOs in the Presentation Layer has been discussed in the first part of this chapter. Based on the above analysis of the core AIO management, a general syntax in BNF grammar has been given to describe the new AIOs in the FOCUS messages. The general internal C data structures to be used in the internal management of the new AIOs have been defined. The differences between the management of new AIOs and the core AIOs have been discussed. The relevant code and routines have been created and extended into the DPM and PPL, co-operating with the existing routines, to manage both of the core AIOs and the extended new AIOs.
CHAPTER 5 IMPLEMENTATION OF THE NEW AIO MANAGEMENT

To extend the new AIOs into the Harness UI/MS, first the designer can create or design their own specific AIOs, by using the AIO Editor, and append the new AIOs into the external AIO Knowledge Bases. Then, according to the AIO message syntax, the designer can combine the extended new AIOs or the core AIOs together to construct the complete user interfaces in a KBM. In this way the designer extends the capabilities of the new AIO management into the Harness without becoming involved in the implementation details and the source code of the Harness.
6.1 Introduction

In this chapter, the development of help systems is reviewed briefly and a summary of design and implementation aspects of help systems is provided. However, it would be relatively difficult to develop domain/application independent help systems. This chapter describes a compromise approach to separate the domain/application-independent help part from applications, and to develop a general software component called a Help System Manager used in the design and the management of help systems under a multi-application environment. This separation provides benefits for user interface designers: they do not need to spend any time on constructing help information windows and on implementing the information access. This approach makes context-sensitive help possible without having help resided in an application program for such contextual information. The Help System Manager and an application interface communicate about context through a help index stack by sending and receiving messages. Several relevant message structures, which are currently provided by the Help System Manager, are described in detail. Using these message structures, the interface designers could easily build context-sensitive help systems.
CHAPTER 6 THE GENERAL HELP SYSTEM MANAGER

6.2 Background: Development, Design and Implementation

Explicit help systems are relatively recent developments in the design of computer software. Prior to about 1975, help capabilities were rare and to be found only in a small number of systems [Kearsley, 1988]. Today, help capabilities are usually considered essential components of well-designed software and on-line help systems have come to be a standard feature of most new software systems. A software system that lacks help features is considered to be incomplete or inadequate.

There are many help systems that have been developed and discussed by Borenstein [Borenstein, 1985], Houghton [Houghton, 1984] and Kearsley [Kearsley, 1988]. Elkerton [Elkerton, 1988] gives a brief list of help systems with appropriate references, and provides a summary of help system research development. He concludes that there is no technology barrier relating to presenting help information. Unfortunately, few behavioural investigations have been conducted on these help systems, and, more importantly, few studies have considered seriously how these help interfaces fit into the user's computer-based task.

Help that is context sensitive is one of the popular research issues. Contextual help is fairly common in today's systems. For example, Lotus 1-2-3 (from Lotus Development Corporation), one of the most popular spreadsheet systems available for PCs, features a context sensitive help that provides help information relevant to the user's present location in the system (as determined by current menu selection).

Fenchel and Estrin [Fenchel and Estrin, 1982] have developed a technique which integrate help information into the design and implementation of a software system. The system is processed by a combined parser generator and an on-line help generator that is part of their SARA (System ARCHitects Apprentice) support environment. As a result, the help generator has a concise representation of the user interface available, making context sensitive help easy to generate.
Involving artificial intelligence techniques and methodology in the design of help systems is a popular research area. From an artificial intelligence perspective, the central problem in designing a good help system is mapping what the user wants to do onto what the system can do [Mark, 1982]. In other words, the system must be able to interpret the intention of the user in performing a certain action so that a meaningful explanation of why it worked, or did not work, can be provided.

An intelligent help system should be able to provide a high level of context sensitivity because it has an understanding of what the user is trying to accomplish [Kearsley, 1988]. Finin [Finin, 1983] showed how a help system could recognise when the user needs help on the basis of inefficient or ineffective actions. Rissland, Valcarce and Ashley [Rissland, et. al., 1984] discuss how an intelligent help system should be able to generate examples relevant to the current context. Fisher, et al. [Fisher, et. al., 1985] describe knowledge-based approaches to provide more intelligent on-line help.

The development of intelligent help systems requires the implementation of rules to determine what advice to give the user. Aaronson and Carroll [Aaronson & Carroll, 1987] have observed and analysed the dialogues between a user and human consultant of an electronic mail system to determine what enquiries and responses will be required with more intelligent on-line help. They identified several strategies that may be useful in future help systems. These strategies include: 1) making explicit assumptions about user goals, 2) proving alternative solutions, 3) assuming an interface configuration, 4) avoiding the problem and 5) pointing to referenced sources.

Another major requirement of an intelligent help system is a comprehensive knowledge base of the application the help is designed for. Rich and Morgan [Rich & Morgan, 1982] suggest that the actual program itself should be intelligent and be able to deal with questions about its operation.

Many researchers feel that natural language understanding is a critical aspect of building intelligent help systems [Kearsley, 1988]. Wilensky, et. al. [Wilensky, et. al., 1984]
describe a help system with a natural language front end called UC (the Unix Consultant), which was designed to provide assistance with Unix Operating System.

Kearsley [Kearsley, 1988], and Elkerton [Elkerton, 1988] discuss relevant research based on the task-analytic model, GOMS, of Card, Moran and Newell [Card et al., 1983], for describing the user's Goals (what the user must accomplish), Operators (the individual actions, such as pressing a key or moving a mouse), Methods (step-by-step procedures for accomplishing goals) and Selection rules (heuristics for specifying which method to use in specific circumstances).

In any case, however, designing an effective help system is a complex process. Some of the design and implementation aspects, which should be considered, are as follows.

*Static vs Dynamic Help*

Static help is independent of where the user is in the system and of any previous user actions. Dynamic help will be dependent upon where the user is or on previous actions. Dynamic help can be either context sensitive or dialogue-based. Context sensitive help information is determined by the current location or function being used. Dialogue help information takes the form of responses to user questions or answers.

*Multiple Levels of Help*

Levels of help, or query-in-depth, is a popular issue in the implementation of help systems. Its great advantage lies in allowing the user to access additional on-line assistance beyond that provided in the first help inquiry [Houghton, 1984]. Houghton suggests that, used together, query-in-depth and context-sensitive help provide a powerful help system.

*Methods of Accessing Help*

Clearly, the method for interacting and accessing help information is a significant problem with user assistance and on-line help [Sondheimerand & Relles, 1982].
Borenstein [Borenstein, 1985] found that providing several access methods (i.e. window-based help, keyword help, computer-initiated help, menu-based help, and context-sensitive help) in a single interface improved the performance of experts and novice users. Hypertext [Conklin, 1987] is another access method. It is likely to become increasingly popular in future software and is ideally suited to help systems [Kearsley, 1988].

**Screen Formatting**

Display of help information is another important design consideration. There are two primary considerations in formatting help displays: 1) location and size of information area, and 2) scrolling or paging within help information area.

Apart from the above, other aspects should also be considered such as extensibility, user vs system initiation, file structure, etc. [Kearsley, 1988]. Carroll, et al. [Carroll, et. al., 1986] emphasised the need for well-designed help texts.

However, it is not an easy task to design an effective on-line help system. Why might a successful help interface be difficult to implement? Elkerton [Elkerton, 1988] states three reasons. The most important reason is that on-line help is not focused on the user's tasks and goals. The second reason may stem from current practices in software interface design. Current design techniques are time consuming and costly since empirical data must be collected from interface users. The third, basic, reason is that on-line help is often not viewed as an integral part of interface design. Campbell [Campbell, et. al. 1988] also emphasised that tools for implementing help should be part of an integrated user interface toolkit. Furthermore, Walker points out that the biggest problem in implementing help systems is the lack of a standard software substrate for storing, managing and retrieving the help information. As a result, each application program is forced to 're-invent the wheel' to provide its own interface to the help information and its own commands for accessing it [Campbell, et. al. 1988].
Kearsley [Kearsley, 1988] addressed several research issues on help. One of them is to develop domain/application independent help systems. However, it would be relatively difficult to develop these systems. One of the compromise approaches is to separate the domain/application independent part from applications and to develop general tools, which can be used in the design of help systems [Huang & Edmonds, 1990, 1991].

In the FOCUS project (ESPRIT 2 Project: 2620 'Front-ends for Open and Closed User Systems'), it was felt necessary to develop general tools to aid the construction of help systems. This was because the writing of various help windows proves very time-consuming. There is a need for software assistance in the initial construction of help windows and in the tasks of modifying windows and transferring windows to new environments. It has been suggested that some textual display facilities should be provided as part of the Harness UIMS that the FOCUS project has developed. Therefore, it is an unnecessary repetition for each module to provide its own interface to handle help information.

Based on the above points, we separate the domain/application-independent part of help systems from applications and develop a general tool called Help System Manager which is used in the design and the management of help systems.

6.3 The Help System Manager

A Help System Manager is a software component designed to handle the display, storage and access of help information. In FOCUS, a proposed architecture for a KBFE (Knowledge Based Front End), which consists of a Front End Harness, Knowledge Based Modules and a Back End Manager, was defined [Edmonds & McDaid, 1990], and a help system and a help system manager are included in the interface component of a knowledge based application module (see Figure 6.1).

In Figure 6.1, the help provided by the help system, can be tailored to the background of the user through the user model. The technical language system can incorporate problem-specific or user-specific terminology in conjunction with the user model.
However, the help system manager can be user/problem-independent. It is necessary to separate the help system manager from the application component and to develop a general help system manager for constructing and managing help systems in the knowledge based application modules.

![Interface Component of a Knowledge Based Application Module](image)

*Figure 6.1 Interface Component of a Knowledge Based Application Module*

This separation brings benefits to the application module designers. The designers do not need to spend any time on constructing help information windows and on implementing the information access. They only need to decide where users need help and what help information is to be provided. This is, of course, unavoidable because these are dependent on application components.

The General Help System Manager, which has been developed at the LUTCHI Research Centre in the FOCUS project, is an additional module to the Harness UIMS. It is responsible for the management of (1) the initialisation, creation and destruction of help index stacks, (2) the creation and destruction of relevant help windows and (3) the access and display of relevant help information from the help information database. The access of the help information database is implemented through the index stacks, which are organised in 2-tuple form: (Context, Index). Here, the 'Index' is a help index related to the 'Context'. Each application module executed under the Harness is provided with a corresponding help index stack, which is maintained by the Help Manager. This approach is shown in Figure 6.2.
The communications between the Help System Manager and application modules as well as the interactions between users and help systems are under the control of the Harness, via standard message structures. These standard messages are passed through the channel, which is created by the Harness when an application module is installed. The basic form of a message is: message(identifier, content, route).

The procedures to execute the General Help System (GHSM) Manager are described as follows:

- Initialisation: when the GHSM receives an "initialise" message from an application KBM, a help index stack corresponding to this application will be created and maintained by the GHSM. A communication channel between the GHSM and the application will be established.

- Maintenance: when the GHSM receives a "push" message with a help index, the help index with the application's current context will be stored on the top of the application's help stack. Once GHSM receives a "pop" message, the help index on the top of the stack will be deleted. It is the application KBM's responsibility to
make the decisions: when, what kind of messages and which index will be sent to the GHSM.

• Help information: when the user needs help (by pressing a help button), the GHSM gets the help index from the top of the stack; searches the Help Information Database using the index to get the help information; and puts the help information in a help window. When two applications require help at the same time, two help information windows will be put on the screen respectively.

• Termination: when the GHSM receives a "terminate" message from an application, the index stack for this application will be removed, and the communication channel between the GHSM and the application will be terminated.

The General Help System Manager provides four types of message structures with the designer to create help systems. The General Help System manager, with the support from the application KBMs, makes the help context-sensitive possible. These are described in the next section.

6.4 Building Context-Sensitive Help

Contextual help is fairly common in today's systems, for example, Lotus 1-2-3 (from Lotus Development Corporation), one of the most popular spreadsheet systems available for PCs, features a context-sensitive help that provides help information relevant to the user's present location in the system (as determined by current menu selection). However, this is not necessarily an easy facility to implement in a complex system.

When the user asks for help within an application module, the specific information related to the current user's problem must be provided. This means that the context of the help request must be known in order to present the adequate information. The Help System Manager eases this task because it is only necessary to send relevant messages
to it when a new task begins and ends. Message structures used to build context-sensitive help are described as follows:

- \texttt{message(Id, \text{help initialise}(String), hsm)}

This message is used to create and initialise a help index stack for the module which sends this message, and on receiving this message the Help System Manager ('hsm') displays an active 'help' button in the window identified by the 'Id'; the 'String' is also displayed in the window (see Figure 6.3). At any time, a user can request help on the current task by pressing the 'help' button.

- \texttt{message(Id, \text{help push}(Index), hsm)}

At the beginning of a task, the module sends this message to the Help System Manager to push the corresponding help information index about the current task into its help index stack. When the user presses the 'help' button, the help information, which is identified by the index 'Index', will be displayed in a help information window (see Figure 6.3).

- \texttt{message(Id, \text{help pop}(), hsm)}

At the end of a task, the module sends this message to the Help System Manager to remove the current help index from its help index stack and to restore the previous help index.

- \texttt{message(Id, \text{help terminate}(), hsm)}

This message structure is used to clean up the link between a module and its help index stack. On receiving this message, the Help System Manager releases the stack and removes the 'help' button window created by the 'help initialise' message.

Using the facilities provided by the Help System Manager, we have developed a context-sensitive help system in a knowledge-based front-end system to the NAG
(Numerical Algorithms Group) Fortran library (see Figure 6.3 & 6.4). This has demonstrated the feasibility of the approach.

![system main menu](image)

![Help Message](image)

**Figure 6.3 A Prototype System for NAG Fortran Routine Selection**

### 6.4 Using Hypertext

Hypertext is popular and suited to help systems [Conklin, 1987 & Kearsley, 1988]. The Help System Manager incorporates the Hypertext tool to access help information. Replacing 'Index' with '[Hypertext, Index]' in the above message structures, the Help System Manager will access a hypertext file identified by the 'Index'. The hypertext tool in the Help System Manager supports the output of data in text, tree and graph
forms. Figure 6.4 gives an example of a hypertext display, in which the emboldened items are hypertext references.

A matrix with $n$ rows and $m$ columns is rectangular if the number of rows is not equal to the number of columns, and is said to be an $n$-by-$m$ or $m$-by-$n$ matrix. The mathematical definition of a rectangular matrix includes the case $m=n$ as a special case. An example of a 3-by-4 matrix with real elements is:

\[
\begin{bmatrix}
2.3 & 0.5 & 2.6 & 4.3 \\
1.9 & 8.7 & 9.2 & 12.3 \\
9.9 & 0.12 & 5.4 & 9.6
\end{bmatrix}
\]

Figure 6.4 An Example of Using Hypertext

6.5 Conclusions

This chapter has presented an approach to separate the domain/application-independent help part of a system from applications as well as a general tool called Help System Manager used in the design and management of help systems under a multi-application environment. Relevant message structures used to handle help systems were described in detail. Using these message structures, the interface designers can easily build context-sensitive help systems. This approach makes context-sensitive help possible
without having help residing in an application program or module. It is unnecessary for each application to provide its own interface part to handle help information, but the designers do not need to spend any time on constructing help information windows and on implementing the information access, because this General Help System Manager is domain/application-independent and it deals with all of these tasks.

These messages and relevant facilities have been demonstrated a system that front-ends a software library. This has indicated that the approach is feasible and the results encouraging.
CHAPTER 7

GRASS KBFE: A PROTOTYPE SYSTEM DEVELOPED UNDER THE ENHANCED UIMS

7.1 Introduction

This chapter describes the development of the GRASS KBFE (Knowledge-Based Front-End) using the FOCUS architecture and the enhanced toolkits, the AIO Editor and the General Help System Manager, which have been mentioned in previous chapters, to demonstrate the advantages of the enhanced UIMS.

7.2 What Is GRASS

GRASS, formally known as the Geographical Resource Analysis Support System, is a public-domain Geographical Information System (GIS). It was developed for GIS specialists to perform serious geographical data management, raster analysis and display [Westervelt, 1991]. GRASS has a command line interface that requires the user to specify a series of complex low-level commands with various parameters and options, although a limited functionality, graphical interaction version was recently released. For example, to display a raster map called 'roads' (Figure 7.1) which is currently in the map database, we should key in the following GRASS commands:

```
d.mon start=x0  
```

open a graphics monitor
d.mon select=x0  

g.list rast  

d.rast roads  

```
select a graphics monitor
list available raster maps in the current location
display 'roads' on the graphics window
```

![Figure 7.1 Display a Raster Map 'roads'](image)

This factor creates difficulties for casual or occasional users of GRASS. In addition, little explicit task support is provided, as is common with many applications. To extend its availability to a broader range of users, GRASS requires a highly usable interface and support for user tasks. Ideally, this should involve minimal development and not require re-implementation of GRASS's capabilities. The enhanced UIMS, which consists of the FOCUS architecture and the enhanced toolkits described in previous chapters, provides the solution.

### 7.3 GRASS KBFE System Configuration

The FOCUS solution presented in this chapter and used in developing the GRASS KBFE adopts a software re-use approach which enables GRASS to be easily integrated into a user interface software architecture without changing the GRASS source code [Edmonds, 1991 & 1992]. The architecture provides a highly usable interface and supports knowledge-based components required by the user for support in non-trivial
problem domain, such as geographical planning. The architecture is modular and operates in a distributed heterogeneous system and machine environment. Currently, the GRASS KBFE prototype is running in the Motif/X11-windows environment on two machines developed by different manufacturers - a SUN Sparc workstation in the LUTCHI Research Centre and a HP workstation in the Computer Centre. Both of the machines are networked via the local network. The modules running on the SUN Sparc machine are Harness, GRASS KBMs, Help Manager, BEM and GRASS KBs (Knowledge Bases) for the BEM. As a back-end application, GRASS4.0 is running on the HP (Hewlett-Packard) machine (Figure 7.2).

![GRASS KBFE Configuration](image)

*Figure 7.2 GRASS KBFE Configuration*

All the information necessary for the configuration of the GRASS KBFE is stored declaratively in the Configuration Knowledge Base. The knowledge is a collection of Prolog facts which include all the host machines which may be used in the KBFE, the physical UNIX process which may run, the KBMs which may be grouped in each physical process, and the resources and tools which KBMs or the BEM may use. The details about the KBFE configuration can be seen in the FOCUS report "Communication Manager User Guide". For this GRASS KBFE prototype, the configuration is declared as follows:

```prolog
/***************************************************/
/* Grass KBFE Sample Configuration */
```
**CHAPTER 7 GRASS KBFE: A PROTOTYPE SYSTEM UNDER THE UIMS**

```plaintext
/* **********************************************/
host(woodstock, sun4, berkeley).
%
process(harnessProcess, _Local, '$HARNESS/presentation', '70x25-1-1', _Font, [pl=_L, debug, show_message]).
process(grassKBMPProcess, _Local, '$HOME/mums_kbfe/grass/KBMS', '70x25-1+1', _Font, [debug, show_message]).
process(bemProcess, _LocalHost, '$FOCUS_ROOT/focus_tools/bem', '70x25-1-l', _Font, [debug, show_message]).
process(hsmProcess, _LocalHost, '$HOME/extension/help_manager/KBMS', '70x25-1-1', _Font, [debug, show_message]).
%
application_resources(harness-runtime, '$HARNESS/bin', runtime(harnessProcess)).
application_resources(bemRes, '$HOME/mums_kbfe/grass/bin', runtime(bemProcess)).
application_resources(grassKB, '$HOME/mums_kbfe/grass/KBMS', [grass_user_env, grass_interface, grass_engine, grass_main]).
application_resources(hsmKB, '$HOME/extension/help_manager/KBMS', [help_manager, help_knowledge]).
%
application_instance(user, harness-runtime, harnessProcess).
application_instance(grass_kbm, grassKB, grassKBMPProcess).
application_instance(bem, bemRes, bemProcess).
application_instance(hsm, hsmKB, hsmProcess).
%
instigate(user).
instigate(bem).
instigate(grass_kbm).
instigate(hsm).
```

### 7.4 Integrating GRASS to Be a FOCUS Back-End

The BEM (Back-End Manager) and its components have been described in great detail in chapter 2. We summarise the procedures to execute the back-end application as follows:

- the BEM receives a message (from a KBM) containing a description of a back-end 'task' to be performed.
- the Task Manager decides on the appropriate application with which to implement the task.
- the Action Manager decomposes the 'task' into one or more back-end 'actions'.
- the Template Engine maps the actions into back-ends commands and executes the back-end application.
- the Extraction Engine extracts the 'relevant information' from the output of the application.
• the Task Manager sends a message back to the calling KBM with the 'relevant information' which then may be conveyed to the user or used as a basis for further reasoning by the KBMs.

To integrate the GRASS4.0 to be a FOCUS back-end via the BEM, the following actions should be investigated:

• Build relevant GRASS KBMs for the Harness.
• Build the environment knowledge base.
• Build the relevant task knowledge base.
• Build the relevant action knowledge base.
• Build the template files for the GRASS commands.
• Build the extraction files for extracting the GRASS output.

7.4.1 GRASS KBMs

Relevant KBMs have to be built to interact with the user as well as with the GRASS back-end through the FOCUS Harness and the BEM. The responsibilities of those KBMs are 1) to send relevant GRASS back end tasks to the BEM, 2) to receive reply message from the BEM with the output parameters instantiated, and 3) to make decisions for the next stage based upon the user's selection and the GRASS' reply. On receiving a task message from the KBMs, the BEM will combine the GRASS package knowledge of tasks, actions, and environment together to invoke the GRASS back-end, and get some output back if necessary. Here, it should be emphasised that when a task is sent to the BEM, it should be fully specified, so that the knowledge available must be sufficient to execute that task. For example, if the KBM sends a task to display a map, the data file name and the data type of the map must be included in that task message. Furthermore, there should be enough context to execute the GRASS command to display the map, e.g. a graphics monitor must be running and be selected.

The communication between the KBMs and the BEM takes place by sending or receiving 'task' messages. The basic format of a task message is:
message(identifier, goal(task-id, task-description), sender:bem).

Here the task 'task-description' is defined in the task knowledge base. To get the raster map list under the current database, for example, a KBM may send the following message to the BEM:

message(list, goal(task1, get_map_list(rast, MapList)), bem).

When the message is sent back from the BEM, the parameter 'MapList' should be instantiated with a list of the available raster maps in the current database.

### 7.4.2 Environment Knowledge Base

The environment knowledge base is divided into two parts: the knowledge required by the BEM and the knowledge required by the GRASS4.0. Following code is a typical example of the environment knowledge base for the GRASS4.0 KBFE prototype system.

```prolog
/* Environment Knowledge Required by BEM */
harness(3.0).
bem_bim_version(3.1).
bem_directory(template_definitions, 
  '/home/mums/mums/mums_kbfe/grass/TEMPLATES').
bem_directory(extraction_definitions, 
  '/home/mums/mums/mums_kbfe/grass/EXTRACTIONS').
bem_directory(bem_temp_files, '/home/mums/mums/tmp').
package(os(unix), _, 'grass4.0', _).
/* GRASS Running Environment Knowledge */
grass_environment(hpf, cohh, LocalHost):- 
  f_shell('hostname', [LocalHost]).
grass_output_file('/home/mums/mums/tmp/grass.out').
grass_command_directory('/sw/grass4.0/bin').
```

The directories of template, extraction and temporary files are included in the knowledge base for the BEM. The GRASS environment knowledge specifies the host, which the GRASS4.0 is running on, the temporary output file and the GRASS command directory on the host. The host can be a local host or a remote host. The back-end package, GRASS4.0, is an OS type package with commands that can be used under the UNIX OS prompt. This is also included in the knowledge base.
7.4.3 Task Knowledge Base

As has been described in section 7.4.1, the message unit transferred between a KBM and the BEM is a back-end task. All of back-end tasks are described in the Task Knowledge Base. Usually, an ordered sequence of one or more back-end specific actions is used to complete a task. Such a map between a task and an ordered sequence of actions is defined in this knowledge base. For example, the task 'gecmap_list' can be specified as follows:

\[
\text{task( get\_map\_list(MapType, MapList) ):-}
\]
\[
\text{grass\_environment(RemoteHost, RemoteUser, LocalHost),}
\text{bem\_directory(bem\_temp\_files, TmpDirectory),}
\text{action\_engine(get\_map\_list,}
\text{os(unix),}
\text{[RemoteHost, RemoteUser, MapType, TmpDirectory],}
\text{[[]],}
\text{task\_extract\_output(MapType, MapList).}
\]

7.4.4 Action Knowledge Base

A back-end action is an object which creates a particular sequence of commands to a back-end depending on the action input parameters, processes the resulting output from the back-end and extracts the required output parameters. The action knowledge base defines the mappings between actions and back-end commands. The back-end commands are defined in the template files. For example, the action 'get_map_list' can be described as follows:

\[
\text{def\_action(get\_map\_list,}
\text{tpl('grass\_tem G\_LIST',}
\text{[remoteHost, remoteUser, dataType, tmpDirectory]),}
\text{[]},
\text{os(unix)).}
\]

7.4.5 Templates

At run-time, the Template Engine mixes the action input parameters with the template to create the correct sequence of back end commands. The templates are built by a back-end expert using the TPL (TemPlate Language), which is one of the BEM tools. For
example, the template 'G_LIST' in the template file 'general.tem' can be defined as follows:

```
G_LIST
  rsh {remoteHost} -l {remoteUser} g.list
type={dataType} > {grassOutput}
ENDTEM
```

7.4.6 Extraction of the Output

If the GRASS output is in graphical format, the information is placed directly onto the user’s display. If the output is in textual format, the BEM extracts the "useful" information and sends it to the relevant KBM for further inference. At run-time the Extraction Engine uses the extraction definition to extract the appropriate action output parameters from the output of the back-end which has resulted from the input generated by the Template Engine. The extraction programs (definitions) are built by a back-end expert using the XTL (eXTraction Language) for extracting various data objects such as vectors, matrices, text, etc. from the output ASCII files.

Because currently the BEM version 4.0 does not support the output extraction for any OS type package back-ends, we have to build our own extraction engine to get some necessary text output back. For example, we want to get a list of raster data files in the current map database. First, we use the GRASS command 'g.list type=rast' to get the following output:

```
raster files available in mapset PERMANENT:
  aspect=aspect  landcov  rail  topo
  contours  plant  roads  urban
  image  popln  source  water
```

Then, we use ‘awk’ or other UNIX facilities to define an extraction engine to extract the raster or vector data files back to be a Prolog list:

```
/* extract grass output predicates */
task_extract_output(rast, MapList) :-
  grass_output_file(GrassOutputFile),
  f_concat_list(['awk "$1 !+/---!rast/", GrassOutputFile}, System),
```

---

136
CHAPTER 7 GRASS KBFE: A PROTOTYPE SYSTEM UNDER THE U1MS

f_shell(System, MapList1),
  (atom(MapList1), MapList = [MapList1])
  ;
  MapList = MapList1
).  
task_extract_output(vect, MapList):-
  grass_output_file(GrassOutputFile),
  f_concat_list(['awk "$1 !~---|vect/"', GrassOutputFile], System),
  f_shell(System, MapList1),
  (atom(MapList1), MapList = [MapList1])
  ;
  MapList = MapList1
).

For every task which needs some information back to the KBM, we should define a relevant extraction engine. Refer to "Implementation issues for fixing GRASS4.0 to be a FOCUS back end" [Huang, 1992] for the details how to integrate GRASS to the BEM.

7.5 User Tasks

As mentioned at the beginning of this chapter, the GRASS lacks explicit user task support, as is common with many other applications. This makes difficulties for casual users of GRASS. The GRASS KBFE prototype provides the support for some user level tasks. These range from general user tasks such as displaying map layers and map information queries to specific user tasks such as optimum routing and site selection. Although the user tasks provided are limited and some are simple in the prototype, the approach adopted in the prototype development is useful and suitable for other similar applications. It is not difficult to extend more user tasks using the enhanced U1MS according to my experience in the development of the GRASS KEBF prototype. There are five categories of user tasks currently supported by the prototype:

- Map Layer Management
- General Map Information
- Map Information Manipulation and Extraction
- Route Optimisation
- Site Selection
Each category has several user tasks. For example, in the category of general map information, there are five user tasks available: 1) available data base file list, 2) map layer content report, 3) map layer legend display, 4) map layer histogram display and 5) location enquiry. The relevant KBM maps an user task into one or a series of back-end tasks and sends them to the BEM. For example, to achieve the user task of the population calculation in the category 3, the following series of back-end tasks will be sent to the BEM: reclass_rast \( \rightarrow \) buffer_zones \( \rightarrow \) reclass_rast \( \rightarrow \) map_calculate \( \rightarrow \) report_map_content.

For more complex user tasks, such as route optimization and site selection, the prototype provides a series of clear steps or sub-user-tasks and help information to assist the user to achieve the task goal. For example, the user has to go through the following sub-user-tasks to finish the route optimisation: 'relative costs assignment' \( \rightarrow \) 'cost-distance surface creation' \( \rightarrow \) 'least-cost pathway determination' \( \rightarrow \) 'final map improvement' \( \rightarrow \) 'final map display'. For the full description of user tasks provided in the prototype, refer to "Design notes for the prototype GRASS KBM" [Huang & Jones, 1992].

### 7.6 The User Interface

The prototype is implemented under the Motif/X11-window system on a Sparc workstation using the FOCUS architecture and the enhanced UIMS toolkit. There are a number of container AIOs, standard AIOs and new (user defined) AIOs employed - these relate to the different combinations of functionality required - and thus to the different types of messages used in the prototype system.

The main window, which is a selection menu (Figure 7.3), shows the five top-level user tasks that the system currently provides. A series windows will pop-up, when the user selects a task.
Most of the user windows can be created from the core AIOs by sending relevant messages to the Harness. Because of the limitation of the core AIOs, however, some of the specific user windows are difficult to create if only the core AIOs are employed. In this case, it is essential to introduce new AIOs to the Harness by the user employing the AIO Editor that has been described in chapter 4. With the combinations of the FOCUS core AIOs and the user defined new AIOs the more complicated user interface can be created. For example, we usually use a selection AIO to display raster maps currently available in the database. However, when the available maps are too many, it will not be convenient for the user to choose one map from the selection menu because the user has to drag the scroll bar down and up to find the desired item (Figure 7.4a). It will be more convenient for the user to display such selections when an user defined AIO, a 'list' AIO for example, is used (Figure 7.4b).

The graphical format output is directly placed on the GRASS' graphics windows. The graphics windows are not only used to display the map's information, but also can be used for the user to interact directly with the system. For example, if the user wants to enquire about the location of a particular site on a map, he/she can interact with the
system by clicking the mouse button on the site, then the site's co-ordinates (x, y) will be extracted from the back-end GRASS and be put on an information window (Figure 7.5).

While the above discussion of the user interface is necessarily brief, it can be seen that general interface windows can be created easily from the FOCUS core AIOs. In addition, by using the AIO Editor the interface designer can create more flexible and specific interaction objects to address the necessary functionality of the front-end application.
7.7 The Help System

A help facility has been implemented in the prototype to provide the user with the assistance to use the system. The help system is context-sensitive and is developed under the Help System Manager, which has been discussed in chapter 6.

To make the help context sensitive, the relevant module only needs to do two things: 1) at the beginning of a task the module sends the help manager a message to push the help index into the help stack, and 2) at the end of the task the module sends another message to remove the help index from the stack and to restore the previous one. The Help System Manager will perform the rest of the tasks, for example, display the help information, maintain the internal help index stacks, etc.

A help knowledge base for the GRASS prototype has to be provided with the Help System Manager. When a help is required, the Help System Manager first traces the knowledge base, via the current help index which is on the top of the index stack, to get the relevant information about the help topic, and then puts the help message on the
screen. In the help knowledge base, there are two kinds of Prolog facts. One is in the format:

help_directory (kbModuleName, helpDirectory).

which indicates that the help files of the KBM module 'kbModuleName' are under the directory 'helpDirectory'. Another kind of fact is in this format:

help_information (kbModuleName, index, type, markup, title, fileOrText).

Here, the 'index' is the help index. The 'type' specifies that the help information is in a file or a piece of text. The 'markup' is 'focus' or 'hypertext' format. The 'title' specifies the window title of the help information. The 'fileOrText' will be a file name if the 'type' is 'file'. Otherwise, it will be a piece of text. Figure 7.6 shows one of the help information windows.

Figure 7.6 A Help Information Window
7.8 Evaluation

One of the aims of the development of this GRASS prototype was to produce some evaluation of the FOCUS architecture, tools and the enhanced tools that include the AIO Editor and the Help System Manager. Through the development of the GRASS prototype and other KBFE systems such as SEPSOL [Murray et. al., 1992], the general FOCUS architecture and the functionality of the tools was found to correspond closely to the KBFE developers' needs. Although the BEM currently cannot handle the output extraction for any OS type package back-ends, the developers have to build their own Extraction Engine. The BEM provided a good robust tool that permitted the easy integration and re-use of the existing back-end packages.

In explaining the value of the FOCUS approach, a number of points need to be made. First, the FOCUS architecture enables several existing back-end applications/packages to be easily integrated into a KBFE system. As mentioned in chapter 2, two additional modules, the KBM and BEM, have been introduced into the FOCUS KBFE architecture. It is these additions that address the integration issue [Edmonds et. al., 1992]. The BEM provides the access to the back-end applications transparently to the user. The KBM provides support to the user in employing the particular application. With the support from the Knowledge-Based Modules, this integration provides the user with a task-oriented solution to the complex problems. For example, in SEPSOL, the integration problem involved concerned some eight existing FORTRAN application programs used to calculate the various statistical models and to perform the comparison and evaluation of those models. These FORTRAN programs have quite difficult input format requirements and are generally not easy to use. The KBSM (Knowledge-Based Support Module) in SEPSOL provides support to the user throughout the problem solving process in a user task context.

Secondly, there are many existing software systems for engineering and scientific that were developed with great care in relation to accuracy and function, but usually with
badly designed or obsolete user interfaces. The FOCUS architecture provides the solution to re-use these existing back-end packages. For example, the GRASS KBFE described in this chapter provides a modern user interface to the command-based GRASS without any change of the source code in this package. SEPSOL also makes the statistical FORTRAN application re-usable.

Finally, the FOCUS architecture is also suitable to front-end to the existing open user systems. For example, the KBFE for NAG FORTRAN routine selection, which was developed at the early stage in the FOCUS project, provided an interface to the NAG FORTRAN Library. In this KBFE, one or several NAG FORTRAN routines will be selected according to the user's problem requirements and an example how to use the selected FORTRAN routines will be generated. The example program will be compiled and linked with NAG FORTRAN Library and the results will be produced to the user.

From the above KBFE prototype systems' development, we conclude that: 1) the FOCUS architecture provides a solution to integrate existing back-end packages easily; 2) The FOCUS architecture is not only suitable to front-end to the existing closed systems, such as GRASS, but also suitable to the open systems, such as the NAG FORTRAN Library; 3) The FOCUS architecture can be used to front-end to a wider range existing applications which are using the text-based or command-based interfaces. However, the architecture is not for the applications that have the direct manipulation front-ends, unless we can "go inside the systems". Also, the designer must develop the relevant KBMs to support the user tasks.

The work reported in this thesis does not extend the basic capability of the FOCUS architecture. However, it extends its utility in a number of ways. The range of AIOs supported by the FOCUS Harness was originally quite limited. However, the more sophisticated KBFEs such as the GRASS KBFE and SEPSOL required a wider range of interaction objects to support the modern mixed initiative interface required. The enhanced Harness with the AIO editor makes the range of the interaction objects theoretically unlimited. The editor enables the developers graphically and textually to
specify the details of new AIO prototypes. While the provision of the editor and the freedom to create new interaction objects adds greatly to the flexibility and utility of the UIMS, it does introduce certain problems. Clearly the developer is at liberty to create AIOs that are not consistent with either the other AIOs in the system or indeed in line with accepted ideas on good interface design [Edmonds et al., 1992]. Good practice is, therefore, still required.

The help system manager makes the separation between the help system and the user interface code possible. The developer does not need to spend any time on constructing help information windows and on implementing the help information access, because the help system manager will deal with these tasks. Although the help in the GRASS prototype is simple, it has demonstrated that the approach is feasible for the development of the context-sensitive help.

The GRASS prototype system provided a good platform to test the enhanced UIMS including the tools. The successful development of the GRASS prototype is clear evidence that the enhanced tools co-operating with the FOCUS tools are both appropriate and sufficiently integrated to enable rapid prototyping of KBFEs. The GRASS prototype system was constructed in less than three months: the development of the whole system from the investigation of the GRASS system to the implementation of the prototype, without the tools support, would take years, rather than months, of effort.

7.9 Summary

We have presented most of the details on the development of the GRASS KBFE using the FOCUS architecture and the enhanced UIMS tools: the AIO Editor and the General Help System Manager. The development of the prototype system has demonstrated that the FOCUS Harness UIMS provided good robust tools for the easy integration of existing software, while the enhanced UIMS, with the support of the AIO Editor and
the Help System Manager, provided the developer with great flexibility and the freedom
to develop more sophisticated KBEE systems rapidly.
CHAPTER 8

DISCUSSION AND CONCLUSIONS

8.1 Introduction

The work described in this thesis is an attempt to enhance the capabilities and the flexibilities of an existing UIMS, the FOCUS Harness, by providing an AIO Editor to make the Harness programmable or extensible in a graphical, direct manipulation manner and by providing a General Help Manager to make the context-sensitive help possible without having help resided in an application. The main work carried out is as follows:

1) An Abstract Interaction Object Editor has been designed and implemented for user interface designers to create and modify AIO knowledge in a high graphical, direct manipulation manner.

2) The Dialogue control and Presentation components in FOCUS Harness has been extended to deal with the new AIOs which are created by the designer using the AIO Editor.

3) A domain/application-independent Help System Manager has been developed for the interface designers to create context-sensitive help facilities in applications.
4) A prototype application system: a knowledge-based front-end to GRASS, which is a GIS application, has been designed and implemented using the AIO Editor and the General Help System Manager to demonstrate the advantages under the enhanced UIMS.

The next section summarises the main work carried out in this thesis. Section 3 gives some discussion and proposals for further work. The last section concludes the thesis.

8.2 Summary

A User Interface Management System is a software sub-system that comprises special tools and techniques to enable designers to create complete and working user interfaces without having to program in a traditional programming language. The motivation of UIMS is to reduce the repetition of the large percentage of code, which is written to implement the user interface. A UIMS intends to be a flexible, re-usable interface module that communicates with the application on one side and with the end user on the other side. The key issue in UIMS is to separate the user interface from the applications. Ideally, the application and the user interface might be separated in such a way that changes to one component cause no changes to the other. However, it is believed that this ideal goal is probably difficult to achieve completely. Once the separation is allowed, how those two components communicate must be considered. Conceptually, this provides us with the three major logical components of an interactive system: presentation, dialogue control and application interface. That is the well-known abstract UIMS model - the Seeheim model. Chapter 2 first described the three components in the Seeheim model in general. Three examples of post-Seeheim models: Dance et al’s, Coutaz’s PAC and the Slinky model were discussed briefly. However, most of the models are variations on the Seeheim view, rather than radical departures from it. They are not module-based systems, lack knowledge-based user support components and have not addressed the general problems of front-ending existing software.
The FOCUS KBFE architecture, which was developed within the FOCUS project, is an extension of the Seeheim model. The FOCUS KFE emphasises the front-ending of existing software systems by extending the Knowledge-Based Modules, which provide the user with guidance and assistance, and the Back-End Manager which enables the interfacing with the integrated applications in a clean and clear separation between the rest of the interface and the applications. Thus, the FOCUS KBFE consists of three components - the Harness, KBMs and BEM. The core component in the architecture, the Harness, is the FOCUS KBFE's operating system and is actually a UIMS. The Harness is divided into five components - HCM, ACM, DDM, DPM and PPL. The Communications Managers - HCM and ACM - control all of the communications in the FOCUS KBFE. This communication takes place using a clearly defined abstract message structure. DDM maintains the logical link between the state of the KBFE and the state of the interface. DPM maintains the logical state of the interface, while PPL maintains the physical state of the interface. The FOCUS KBFE architecture has been discussed in great detail in chapter 2.

Chapter 3 presented an investigative study on user interface tools in two aspects: user interface toolkit approaches and UIMS approaches. The examples of the most popular user interface toolkits today are X-Window Toolkit (Xtk) for workstations, Toolbox for Apple Macintosh and Software Development Kit (SDK) for MS-Windows. One of the motivations for the toolkits is that the final interface will have a "look and feel" consistent with others created using the same toolkits. Although there are many similarities between the various toolkits, people still need to do a lot of work to convert their software from one windowing system to another. A number of virtual toolkits have been developed to try to hide the differences among the various toolkits and to overcome the problem. Two examples of virtual toolkits are XVT (Extensible Virtual Toolkit) and SUIT (Simple User Interface Toolkit). The advantage with the toolkits is that the programmer has extensive control and great flexibility in creating the user interface. The disadvantages are: they are expensive to create, difficult to use and to maintain, they are complex, there is a lack of standardisation and no well separated
code for the user interface and the application, etc. Therefore, there is a need for additional support in the design and use of toolkits to overcome their deficiencies. Beyond the toolkits, UIMS approaches provide another level of services for the user interface design.

UIMS approaches are classified into four groups: language-based specification, direct graphical interface specification, automated interface generation and model-based user interface development systems. Earlier systems are usually language-based. Systems such as Robertson et al's ZOG, Kasik's TIGER/UIMS, Newman's Reaction Handler, Jacob's State Diagram Specification Interpreter, Olsen's SYNGRAPH, Edmonds' SYNICS, Green's UofA, Flechti's ALGAE, Hayes' COUSIN, etc are grouped as the category of language-based specification. Object-oriented approaches have become the fashion in user interface design and specification. One of the well-known systems that provides an overall application framework and use the object-oriented language is MacApp. The advantages of object-oriented systems are: higher productivity because the code can be shared and re-used; decoupling of representation from implementation; reliability, consistency, and locality of definition from inheritance; and low code bulk. The disadvantages include a steep learning curve for programming, a high performance penalty for systems due to the fact that they are interpretative and use dynamic binding and message passing, etc. With the appearance of direct manipulation interfaces, direct graphical specification has emerged for constructing the user interface in direct manipulation manner. In direct graphical specification, the designer defines the user interface by placing an object on the screen interactively using a pointing device without programming. A number of research UIMS (such as DialogEditor, vu, Gilt) and commercial systems (such as UIMX, Prototyper, MS-Visual C++) provide graphical facilities called interface builders or resource editors for creating the presentation or appearance of the user interface. However, the designer has to provide the code for the behaviours of the user interface. An advanced method - demonstrational interfaces - has emerged. Demonstrational interface builders allow the creation of the complete user interface in direct manipulation and demonstrational manner without programming. The
system pioneered to support demonstrational interface creation is Myers' Peridot. Other systems such as OSU, Druid and DEMON are also demonstrational. Although all of these systems are research prototypes, some of the ideas are beginning to appear in commercial systems. Automated user interface generation approaches try to create the user interface automatically from a high level specification of the application semantic description and then allow the designer to modify the generated interface to improve it. Examples of UIMS that automate some aspects of this process are MIKE, Mickey, Jade, UofA*, DON, GENIUS and TRIDENT. One group of UIMS, which automatically generate the user interface by building a model of the presentation and behaviour (such as UIDE, HUMANOID and ITS), are characterised as model-based systems. The approach of automated user interface generation still needs to do further research, because the user interfaces that have been generated are not good enough and the languages for the model specification are difficult to learn and use.

The Harness UIMS has been developed for knowledge-based front-ends. This has been done as part of the ESPRIT project FOCUS (2620), in which an architecture and set of tools have been developed to support the front-ending of existing software systems. In the FOCUS Harness, a set of AIOs was provided for knowledge-based modules to interact with the user. However, it is not realistic to expect the Harness to be a closed and fixed system. For each particular application, it will be necessary to decide, at least, on the presentation style. It may also be necessary to introduce new Abstract Interaction Objects, such as specialised forms for the knowledge-based module designer to construct a specific user interface and, naturally, it is most convenient to build such information into the Harness. To extend such capability into the Harness, the conventional approach is that the designer can be given access to the source code of appropriate parts of the Harness and may, therefore, modify it or add new interaction object structures to meet his or her requirements. In this case, the designer, at least, has to know the Harness structure as well as its programming environment. Obviously, it is a boring and time-consuming task to read and to analyse a big complex software system. Chapter 4 presented an approach to extend such
capability into the Harness in a graphical, direct manipulation manner, i.e. by providing the designer with an interaction object editor for the creation of new specific abstract interaction objects and for the introduction of them to the Harness. In doing so, an AIO knowledge base has been extended into the FOCUS KBFE architecture and an AIO editor has been designed, implemented and presented in great detail in the fourth chapter. The AIO knowledge base contains the presentation definitions as well as the behaviour definitions of interaction objects, which are created by the designer through the AIO editor. By using the approach and the editor, the designer could dynamically create, modify, and maintain interactive graphical interaction objects and extend the interaction object knowledge base by direct manipulation without doing any programming and without becoming involved in the implementation details and the source code of the Harness UIMS. This AIO editor provides a more flexible, rapid and effective means for the designer to specify and test the presentations and the behaviours of user interaction objects. Once a new AIO is created and extended into the AIO knowledge base, the knowledge-based modules can quote it as one of the core AIOs by sending a relevant message to the Harness and, in turn, the Harness accesses the AIO knowledge base to create and to display the interaction object on screen.

To accommodate the above extension, the relevant program parts in the Harness have to be modified, so that the Harness has the capabilities to manage the extended new AIOs. For the built-in core AIOs, the Harness manages them individually in different syntax and data structures. For the extended new AIOs, obviously, it is impossible to manage them individually, because the appearances of the new AIOs are in a large variety. It is necessary to provide the Harness with a general method to manage all extended new AIOs. Based on the analysis of the implementation of the Harness, chapter 5 defined a general-purpose syntax using BNF (Backus-Naur Form) grammar for the new AIO description. The general internal C data structures, which are used in the internal management of the new AIOs, have been defined. The implementation detail of the new AIO management in the enhanced Harness has also been discussed.
Today, help capabilities are usually considered essential components of well-designed software and on-line help systems have come to be a standard feature of most new software systems. A software system lacks help features is considered to be incomplete or inadequate. Chapter 6 reviewed the development of help systems and summarised the design and implementation aspects. However, it is not an easy task to design an effective on-line help system. The biggest problem in implementing help systems is the lack of a standard software substrate for storing, managing, and retrieving the help information. As a result, each application program is forced to 're-invent the wheel' to provide its own interface to the help information and its own commands for accessing it. Obviously, this is an unnecessary repetition for each application. In FOCUS project it was also felt necessary to develop general tools to aid the construction of help systems. One of the research issues to overcome the problems is to develop domain/application independent help systems, but it would be relatively difficult to develop such systems. One of the compromise approaches is to separate the domain/application independent part from applications and to develop a general tool, which can be used in the design of help systems. Chapter 6 presented this approach and a general tool called Help System Manager for this purpose. The Help Manager, as an additional module, has been implemented and extended into the FOCUS KBFE architecture. This separation brings some benefits to the application designers. They do not need to spend any time on the construction of help information window and on the implementation of the information access. They only need to decide where users need help and what help information is to be provided. These are unavoidable because they are dependent on applications. The Help Manager is responsible for the management of 1) the creation and initialisation of individual help index stack, 2) the construction and destruction of help information windows and 3) the access and display of relevant help information from the help information database. Relevant message structures used to handle help systems were described in detail in the sixth chapter. With these messages, the designers can easily build context-sensitive on-line help systems. This approach
makes context-sensitive help possible without having help residing in an application program or module.

Three models - AIO Editor, AIO Knowledge Base and Help Manager - have been extended into the FOCUS KBFE architecture. Here, we summarise the enhanced KBFE architecture in Figure 8.1.

![Diagram](image)

**Figure 8.1 The Enhanced KBFE Architecture**

Chapter 7 presented a prototype system, a KBFE to the GRASS geographic information system, developed using the enhanced architecture and tools: the AIO Editor and the General Help System Manager, to demonstrate the advantages of the enhanced UIMS. GRASS has a poor command line interface that requires the user to key in a series of complex low-level commands with various parameters and options. It provides little explicit user task support. These factors create difficulties to casual users. The enhanced UIMS provided a solution to extend its availability to a broader range of users with the highly usable interface and the user task support without requiring re-implementation of GRASS's capabilities. The development details of the GRASS KBFE have been presented in the seventh chapter. Because of the limitation of the core AIOs, some specific user interfaces are difficult to create if only the core AIOs are employed. The AIO Editor provided the designer with the power to create more flexible and specific interaction objects to address the necessary functionality of the front-end
application. A help facility was implemented in the prototype to provide the user with system assistance. The help system was developed under the Help System Manager, which makes the separation between the help system and the user interface code possible. The developer does not need to spend any time on constructing help information windows and on implementing the help information access, because the help system manager will deal with these tasks. Although the help in the GRASS prototype is simple, it has demonstrated that the approach is feasible for the development of the context-sensitive help. The GRASS prototype system provided a good platform to test the enhanced UIMS. The successful development of the GRASS prototype is clear evidence that the enhanced tools co-operating with the FOCUS tools are both appropriate and sufficiently integrated to enable rapid prototyping of KBFEs.

8.3 Discussion and Proposals for the Further Work

As has been discussed in chapter 7, the provision of the AIO editor and the freedom to create new AIOs add greatly to the flexibility and utility of the UIMS on the one hand, but they do also introduce certain problems. Clearly, the developer is at liberty to create AIOs that are not consistent with either the other AIOs in the system or indeed in line with accepted ideas on good interface design. Currently, there is no way in which the developer can, realistically, be constrained within the editor provided. However, as the user interface consistency and quality is a central issue, this problem is of considerable importance. Its solution could be in the form of a Knowledge-Based AIO Editor that provided a CASE like environment. This is an interesting research issue in its own right.

In recent years, research in the area of demonstrational interface or programming by example has been growing fashionable. Demonstrational user interfaces provide concrete examples on which the user operates, rather than requiring the user to deal with abstractions such as variables and control structures. A "demonstrational interface" watches while the user executes conventional direct manipulation actions, but creates a
general abstraction from the specific example. It is worth exploring this technique for the AIO Editor in the future.

As we discussed in chapter 2 how to create a user interface or a dialogue window, a module sends a message to the Harness. The message consists of a meaningful combination of AIOs and their attributes in a correct syntax. To check the syntax in an interface message and to have a feel of it, we have to embed this message into a module and to execute the Harness to check its syntax first and then display the window on screen if the grammar is correct. Although the message syntax is not too complicated, the designer still needs time to learn and often some mistakes are unavoidable. Ideally, a high level user interface editor, which can be used to produce the message automatically in a graphical, direct manipulation manner and to test the interface without having the message resided in a module, will be a very helpful tool in the user interface design.

8.4 Conclusions

It is not realistic to expect an existing UIMS to be a closed and fixed system. However, the conventional method for maintaining and extending an existing system is boring and time-consuming. The work described in chapter 4 and 5 shows that it is feasible to extend the capabilities of an existing UIMS in the graphical, direct manipulation manner without becoming involved in the implementation details and the source code of the system.

Although it would be difficult to develop domain/application independent help systems, the work described in chapter 6 shows a compromise approach to separating the domain/application-independent help part from applications. This separation provides benefits for user interface designers: they do not need to spend any time on constructing help information windows and on implementing the information access. This approach makes context-sensitive help possible without having help resided in an application program for such contextual information.
Finally, the development of the GRASS prototype system demonstrates that the enhanced UIMS, with the support of the AIO Editor and the Help System Manager, provides the developer with great flexibility and the freedom to develop more sophisticated KBEF systems rapidly.
REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


APPENDICES

A.1 Message and AIO Syntax

The following syntax uses Backus-Naur Form.

A.1.1 General Message Form

\[ \text{<Message>} ::= \text{message}(\text{<MessageId>,<ContentAndDestination>}) \]
\[ \text{<ReturnMessage>} ::= \text{message}(\text{<MessageId>,<ReturnContent>,user:<Destination>}) \]
\[ \text{<MessageId>} ::= \text{<Id>} \]
\[ \text{<Id>} ::= \text{<Term>} \]
\[ \text{<ContentAndDestination>} ::= \text{<RouterContent>,<From>:feh} \]
\[ \text{<From>:user} \]
\[ \text{<ControIContent>,<From>:<Destination>},\text{<Term>,<From>:<Destination>} \]

\[ \text{<Destination>} ::= \text{<Process>} \]
\[ \text{<From>} ::= \text{<Process>} \]
\[ \text{<RouterContent>} ::= \text{instigate(<SymbolicName>)}, \text{terminate_request(<SymbolicName>)}, \text{end_of_session} \]
\[ \text{<ControlContent>} ::= \text{predicate(<Goal>), predicate_results(<YesOrNo>,<Goal>)} \]

\[ \text{<Process>} ::= \text{atom} \]
\[ \text{SymbolicName} ::= \text{atom} \]
\[ \text{<Goal>} ::= \text{<Term>} \]
\[ \text{<YesOrNo>} ::= \text{yes} | \text{no} \]
\[ \text{<Term>} ::= \text{atom} | \text{<Real>} \]
integer |<List> | atom(<Term>)

<Real>::= real | real-between-0-and-1

<list>::= <Term> |<List>,<Term>

A.1.2 Message to the End User

A.1.2.1 Container Contents

<ContainerContent>::=<Destruction> |<Sampling> |<StateRequest> | (<DdmAction>,<Notify>,<Container>)

<DdmAction>::= discard | queue

<Notify>::= notell | tell

<Container>::= group(<Id>,<Aios>,<ContainerAttributeList>) | panel(<Id>,[<Panel>],<ContainerAttributeList>) | query(<Id>,<TextPresentation>,<ContainerAttributeList>)

<Destruction>::= destroy(<Ids>)

<Sampling>::= sample(<Id>,<Frequency>)

<stateRequest>::= request_state(<Ids>)

<Ids>::= <Id> |<IdList>

<IdList>::= <Id> |<Id>,<IdList>

<Frequency>::= integer

<Panel>::= [<PanelRow>] |<Panel>,[<PanelRow>]

<PanelRow>::= <PanelAio> |<PanelAio>,<PanelRow>

A.1.2.2 Container Attributes

<ContainerAttributeList>::=[<ContainerAttributes>] |
<ContainerAttributes>::=<ContainerAttribute>  |  
<ContainerAttribute>,<ContainerAttributes>

<ContainerAttribute>::=controls:<Controls>  |  
<GeneralContainerAttribute>

<GeneralContainerAttribute>::=behaviour:<Behaviour>  |  
<ControlId>:<Term>  |  
minor_controls:<Menu>  |  
relation:<Relation>  |  
disposition:<Disposition>

<Behaviour>::= modal  |  
modeless

<Controls>::= [<ControlList>]  |  
update([<ControlList>])

<ControlList>::= <Control>  |  
<Control>,<ControlList>

<Control>::= <ControlId>  |  
<ControlId>(<ControlArgList>)

<ControlId>::= atom

<ControlArgList>::=<ControlArg>  |  
<ControlArg>,<ControlArgList>)

<ControlArg>::= disposal:<Disposal>  |  
validate:boolean  |  
<SelectionArg>

<Disposal>::= immediate  |  
ifValid  |  
postponed

<SelectionArg>::= state:<State>  |  
external:<External>

<State>::= on  |  
off  |  
disable

<External>::= <Icon>  |  
<Term>

<Icon>::= icon(<PathFile>,<format>)

<PathFile>::= host:path  |  
path

<Format>::= xwd  |  
cgm  |  
postscript  |  
bitmap  |  
macpaint
A.1.2.3 Abstract Interaction Objects

<CoreAio>::= selection(Id,Selections,SelectionAttributes)
inform(<Id>,<Presentation>,[<InformAttributes>])  
hypertext(<Id>,<Presentation>, [<HyperTextAttributes>])  
question(<Id>,<QuestionText>, [<QuestionAttributes>])  
<PanelAio>  
<html Aio>

A.1.2.4 Selection AIO

<Selections>::= [<SelectionList>]  
update([<SelectionList>])  

<SelectionList>::= <SelectionItem>  
<SelectionList>,<SelectionList>

<SelectionItem>::= [<SelectionList>]  
input(<Id>,<InitialValue>,<InputAttributes>)  
<ControlId>  
<ControlId>(<SelectionArgList>)

<InitialValue>::= append(atom)  
insert(atom)  
atom

<SelectionAttributes>::=<SelectionAttribute>  
<SelectionAttribute>,<SelectionAttributes>

<SelectionAttribute>::=structure:<Structure>  
label:<Label>  
limits:(<Lower>,<Upper>)

<Structure>::= nestedList  
directed_graph  
matrix

<Label>::= <Term>

<Lower>::= integer

<Upper>::= integer  
?

A.1.2.5 Inform AIO

<Presentation>::= atom  
[<AtomList>]  
<AtomList>  
textfile(<Pathfile>)  
graphic(<Id>,<Pathfile>,<Format>)  
plot(<Id>,[<PlotData>],<PlotAttributes>)

<AtomList>::= atom  
atom, <AtomList>

<PlotData>::= <PointList>

182
A.1.2.6 Hypertext AIO

A.1.2.7 Question AIO

<Validation>::= ( <UserInput>, <Goal>, <ErrorTemplate> )

<UserInput>::= atom

<ErrorTemplate>::= <Presentation>
A.1.2.8 Panel AIO

\[
\text{<PanelAio>::= text(<Id>,<TextPresentation>,<TextAttributes>) | input(<Id>,<InitialValue>,<InputAttributes>) | choice(<Id>,<Choices>,<ChoiceAttributes>)}
\]

\[
\text{<TextPresentation>::= atom}
\]

\[
\text{<InputAttributes>::= validate:<Validation>}
\]

\[
\text{<Choices>::= [ChoiceList] | update([<<ChoiceList>]})
\]

\[
\text{<ChoiceList>::= <ChoiceItem> | <ChoiceItem>,<<ChoiceList>}
\]

\[
\text{<ChoiceItem>::= variable | <ControlId> | <ControlId>(<ChoiceArgument>)}
\]

\[
\text{<ChoiceArgument>::= state:<State>}
\]

\[
\text{<ChoiceAttributes>::= []}
\]

A.1.2.9 New AIO

\[
\text{<NewAio>::= new_aio(<AioId>, <NewAioName>, <NewAioAttributeList>, <NewAioAttachment>)}
\]

\[
\text{<AioId>::= <Term>}
\]

\[
\text{<NewAioName>::= atom}
\]

\[
\text{<NewAioAttributeList>::= [] | [<NewAioAttributes>]}
\]

\[
\text{<NewAioAttributes>::= <NewAttribute> | <NewAttribute>,<NewAioAttributes>}
\]

\[
\text{<NewAioAttribute>::= <ObjectName>:<ObjAttributeList>}
\]

\[
\text{<ObjectName>::= atom}
\]

\[
\text{<ObjAttributeList>::= [] | [<ObjAttributes>]}
\]

\[
\text{<ObjAttributes>::= <ObjAttribute> | <ObjAttribute>,<ObjAttributes>}
\]

\[
\text{<ObjAttribute>::= <ResourceName>:<ResourceValue>}
\]

\[
\text{<ResourceName>::= atom}
\]
<ResourceValue>::=atom | number | string

>NewAioAttachment>::=[ ]

[AioAttachment]:=[ attachment : <AioAttachment>]

<AioAttachment>::=independent |
    above(<term>) |
    below(<term>) |
    next_to(<term>)

A.1.3 Return Messages

<ReturnContent>::=return(<Ids>,<ContainerCause>,<ContainerState>) |
    absent_application(<Process>) |
    aio_status(<ContainerId>,<DdmResponse>)

<ContainerId>::= <Id>

<ContainerCause>::=control(<ControlId>) |
    menu(<Ancestry>) |
    <AioCause>

<AioCause>::= click(<Symbol>) |
    sample |
    kbm_request

<ContainerState>::=<AioStatus> |
    <AioStatus>,<ContainerState> |
    [<ContainerState>]

<AioStatus>::= <Id>=<AioState>

<Ancestry>::= <List>

<AioState>::= <list> | %from selection
    atom | %from question, input, choice
    [PanelState] | %from panel

<PanelState>::= [<RowState>] |
    <PanelState>,[<RowState>]

<RowState>::= <AioState> |
    <AioState>,<RowState>

<DdmResponse>::=realised |
    discarded;<Reason> |
    queued;<Reason>

<Reason>::= user_busy |
    unknown_request
APPENDICES

A.2 Yacc Program List

/* file name : mparse.yac */
%
#include "d_header.h"
#include "d_AioL.h"

/* Variables for building linked lists of elements */
controllist_t *CurrentControlPtr = NULL, /* used to build */
    *PrevControlPtr = NULL; /* control list */
list_choices_t *CurrentChoicePtr = NULL, /* Choice list */
    *PrevChoicePtr = NULL;
/* Variables filling up common structures */
selection_arg_t *SelectionArgStruct= NULL;
/* Temp storage for altered values */
aio_structure_t *tempContainer;
question_t *tempQuestion;
/* Temp storage for markup */
markup_t markup;
/* Temp storage for attachment */
attachment_t *attachPtr = NULL;
/* Choice list */
plot_t *CurrentPlotPtr; /* For inform plot */
BOOLEAN NewControl, /* For inform plot */
    NewQuestion = TRUE, /* if TRUE allocate */
    NewStructure = TRUE,
    NewSelection = TRUE,
    NewSelectionArg = TRUE,
    NeedControls = TRUE,
    NewPlot = FALSE;
int **IntPtr; /* Array of ints */
char *InputPtr; /* Current Message */
/* Temp storage for new_aio */
char new_aio_id[100];
object_t *tempNewAio;
object_t *tempObject;
%
%start ContainerContent
%union{
    char string[BUFSIZ];
    int integer;
    float real;
    frame_t *StructureType;
    aio_structure_t *aioStructure;
    controllist_t *ControlItems;
    able_t state;
    external_t *external;
    file_format_t format;
    multiple_t *multiple;
    inform.presentation_t *presentation;
    plot_type_t plottype;
    plot.data_t *plotdata;
    values_t *values;
    term_t *labels;
    response_t response;
    disposal_t disposal;
    behaviour_t behaviour;
    disposition_t disposition;
    list_choices_t *choice_list;
    initial_value_t *initial_value;
}
APPENDICES

%token <string> INFORM
%token <integer> INTEGER
%token <string> INSERT
%token <string> LABEL
%token <string> LATEX
%token <string> LIMITS
%token <string> LINE
%token <string> MACPAINT
%token <string> MARKUP
%token <string> MAX_VALUES
%token <string> MIN_VALUES
%token <string> MINOR_CONTROLS
%token <string> MODAL
%token <string> MODELESS
%token <string> NEXT_TO
%token <string> OFF
%token <string> ON
%token <string> NOTELL
%token <string> PAGE
%token <string> PANEL
%token <string> PARAGRAPH
%token <string> PHRASE
%token <string> PLOT
%token <string> PLOT_LABELS
%token <string> PLOT_POSITION
%token <string> PLOT_TYPE
%token <string> PLOTFILE
%token <string> POSTPONED
%token <string> POSTSCRIPT
%token <string> QUERY
%token <string> QUESTION
%token <string> RASTER
%token <real> REAL
%token <real> REAL01
%token <string> RELATION
%token <string> RESPONSE_LENGTH
%token <string> SIBLING_OF
%token <string> STATE
%token <string> SELECTION
%token <string> STRUCTURE
%token <string> NESTED_LIST
%token <string> DIRECTED_GRAPH
%token <string> MATRIX
%token <string> TELL
%token <string> TEXT
%token <string> TEXTFILE
%token <string> THREE_D
%token <string> TITLE
%token <string> TREE_STRUCTURE
%token <string> TWO_D
%token <string> UPDATE
%token <string> VALIDATE
%token <string> VALUE
%token <string> VARIABLE
%token <string> XWD
%token <string> NEW_AIO /* for new aio */
%token <string> DEAD_APPLICATION
%token <string> SAMPLE
%token <string> REQUEST_STATE
%token <string> DIRECTORY

$type <multiple> Aios
#type <multiple> AioElement
#type <multiple> AioList
#type <string> AtomList
#type <behaviour> Behaviour
#type <choice_list> ChoiceList
#type <choice_list> ChoiceItem
#type <choice_list> Choices
#type <StructureType> Container
#type <aioStructure> ContainerAttributeList
#type <ControlItems> ControlList
#type <ControlItems> Control
#type <string> ControlValidation
#type <control_container> Controls
#type <multiple> CoreAio
#type <disposal> Disposal
#type <disposition> Disposition
#type <external> External
#type <format> Format
#type <aioStructure> GroupTerm
#type <string> QueryContent
#type <aioStructure> QueryTerm
#type <aioStructure> PanelTerm
#type <coordinate> PlotData
#type <coordinate> PlotDataList
#type <plotdata> PlotDetail
#type <multiple> InformTerm
#type <multiple> HypertextTerm
#type <multiple> QuestionTerm
#type <multiple> SelectionTerm
#type <selection> SelectionAttributeList
#type <selection_list> Selections
#type <selection_list> Children
#type <selection_list> Tree
#type <selection_list> SelectionItem
#type <integer> UpperLimit
#type <structure> Structure
#type <symbol> Symbol
#type <question> QuestionAttributeList
#type <string> QuestionContent
#type <labels> Labels
#type <menu_list> MenuItem
#type <menu_list> MenuList
#type <menu_list> MenuTree
#type <panel> Panel
#type <panel> PanelContent
#type <panel> PanelRow
#type <panel> PanelAio
#type <string> Pathname
#type <plottype> PlotType
#type <plot_array> PlotArray
#type <presentation> Presentation
#type <real> Real
#type <state> State
#type <string> Term
#type <string> TermList
#type <values> ValueList
#type <response> ResponseLength
#type <initial_value> InitialValue
#type <integer> AioOperation
#type <integer> Destruction
#type <string> Destruct_Items
#type <string> CommaSepList
#type <integer> Sampling
%type <integer>  StateRequest
%type <integer>  Dead_Client
%type <integer>  Set_Directory
%type <multiple> NewAio
%type <multiple> NewAioTerm
%type <new_aio> NewAioAttributeList
%type <string>  ResourceValue;
%type <string>  ResourceName;

%%

ContainerContent  : Container
                     {dpm_pplInterface($l,updateFlag); }
                     AioOperation
                     ;

AioOperation  : Destruction
                  | Sampling
                  | Dead_Client
                  | Set_Directory
                  ;

Set_Directory  : DIRECTORY '(' ATOM ',' ATOM ',' ATOM ')' 
                    { ddm_set_directory($3,$5,$7); }
                    ;

Dead_Client  : DEAD_APPLICATION
               {ddm_removeall_objects(StartOfAios,Owner);}
               ;

Destruction  : DESTROY
              {dpm_rule("destroy command");} 
              '(' Destruct_Items ')' 
              { dpm_destroy_frame(Owner,$4); }
              ;

Destruct_Items  : Term
                 | '[' '{' {strcpy($$,""});} 
                 | '[' CommaSepList ']'
                 { strcpy($$,"["); strcat($$$,$2); strcat($$$,""} 
                 ;

CommaSepList  : ATOM { strcpy($$, $1); } 
               CommaSepList ',' ATOM 
               { strcpy($$, $1); strcat($$$,""),strcat($$$,$3); } 
               ;

Sampling  : SAMPLE
          {dpm_rule("sample request");} 
          '(' Destruct_Items ',' INTEGER ')' 
          (ddm_sample(Owner,$4,$6);}
          ;

StateRequest  : REQUEST_STATE
              {dpm_rule("state request");} 
              '(' Destruct_Items ')' 
              (dpm_request_state(Owner,$4);}
              ;

Container  : GROUP { dpm_rule("group"); }
          '(' GroupTerm ',' Aios ',' ContainerAttributeList ')' 

190
PanelContent  :  VARIABLE
{ $$ = NULL; }
| Panel
|

QueryContent  :  VARIABLE
{ strcpy($$, "") ; }
| ATOM
|

ContainerAttributeList  :  Null
{ $$ = NULL; }
| '[ ' ( dpm_rule("ContainerAttributeList") ; )
| ContainerAttributes
{ $$ = tempContainer; tempContainer = NULL; NewStructure = TRUE; }
|

ContainerAttributes  :  ContainerAttribute
| ContainerAttributes ' , ' ContainerAttribute
|

Controls  :  CONTROLS { dpm_rule("controls") ; }
| ' : Controls
{ tempContainer = dpm_set_container_controls(tempContainer,$4); }
| GeneralContainerAttribute
|

Controls  :  '']}' ( dpm_rule("Control List") ; )ControlList ' ]
{ $$ = dpm_control_container($3,untouched1); }
| UPDATE ' ( '
{ dpm_rule("Update Control list") ; )
| ControlList ' ) ' )'
{ $$ = dpm_control_container($5,update); }
| Null
{ $$ = NULL; NeedControls = FALSE; }
|

ControlList  :  Control
{ $$ = $1; }
| ControlList ' , ' Control
{ dpm_add_next_control($$, $3); }
|

Control  :  ATOM ( dpm_arg($1); dpm_rule("Control element") ;
| $$ = dpm_control_button(CurrentControlPtr,$1); }
| ATOM ' ( '
{ dpm_arg($1); dpm_rule("Control element with arguments") ; )
| ControlArgList ' )'
{ $$ = dpm_control_button(CurrentControlPtr,$1); }
|
APPENDICES

ControlArgList : ControlArg
    ControlArgList "," ControlArg

ControlArg : DISPOSAL {dpm_rule("control disposal"); }
    :' Disposal
    (CurrentControlPtr =
      dpm_set_control_disposal(CurrentControlPtr,$4);)
    | VALIDATE { dpm_rule("control validate"); }
    | :' ControlValidation
    (CurrentControlPtr =
      dpm_set_control_validation(CurrentControlPtr,$4);)
    | STATE { dpm_rule("control state"); }
    | :' External
    (CurrentControlPtr =
      dpm_set_control_external(CurrentControlPtr,$4);)
    | EXTERNAL {dpm_rule("control external"); }
    | ATOM :' ATOM
    { dpm_arg($1);dpm_rule("control argument"); dpm_error_parse($1); }
    ;

ControlValidation : BOOLEAN
    | ElError
    ( strcpy($$,"false"); )
    ;

Disposal : IMMEDIATE { $$ = immediate; }
    | IF_VALID { $$ = ifValid; }
    | POSTPONED { $$ = postponed; }
    | ElError
    ( $$ = immediate; )
    ;

GeneralContainerAttribute : BEHAVIOUR { dpm_rule("behaviour"); }
    | :' Behaviour
    { tempContainer = dpm_set_container_behaviour(tempContainer,$4); }
    | TITLE { dpm_rule("title"); }
    | :' Term
    { tempContainer = dpm_set_container_title(tempContainer,$4); }
    | RELATION { dpm_rule("relation"); }
    | :' Relation
    { tempContainer =
      dpm_set_container_attachment(tempContainer,attachPtr);
      attachPtr=NULL; }
    | DISPOSITION { dpm_rule("disposition"); }
    | :' Disposition
    { tempContainer =
      dpm_set_container_disposition(tempContainer,$4); }
    | MINOR_CONTROLS { dpm_rule("minor controls"); }
    | :' Menu
    | ATOM :' ATOM
    { dpm_arg($1);dpm_rule("control argument");dpm_error_parse($1); }
    ;

Menu : ['(dpm_rule("Menu");cParent=NULL;)MenuList ']
    {tempContainer =
      dpm_set_container_menu(tempContainer,$3,untouched1);}
    | UPDATE ['-''(dpm_rule("Menu");cParent=NULL;)MenuList' ']'
    {tempContainer = dpm_set_container_menu(tempContainer,$5,update);}
APPENDICES

MenuList: MenuTree

MenuTree:

MenuItem

($$ = Modal;)

MODELESS {$$ = modeless; }

El Error ( $$ = modeless; )

Attachment

{ attachPtr = dpm_set_attachment(attachPtr, alone, NULL); }

ABOVE { dpm_rule("above"); } ...

BELOW { dpm_rule("below"); } ...

NEXT_TO ( dpm_rule("next_to"); }

INDEPENDENT { attachPtr = dpm_set_attachment(attachPtr, next_to, $4); }

Relation

{ tempContainer = dpm_set_container_relation(tempContainer, independent, NULL, NULL); }

EXTENSION_OF { dpm_rule("extension_of"); } ...

SIBLING_OF { dpm_rule("sibling"); } ...

CHILD_OF { dpm_rule("child_of"); } ...

EXPLANATION_OF { dpm_rule("explanation"); } ...

193
APPENDICES

} IN_RESPONSE_TO { dpm_rule("in response to"); }
(' Symbol ')
(tempContainer = dpm_set_container_relation(tempContainer,
in_response_to, NULL,$4 );)
}

ElError;

Symbol : HYPERTEXT_KEY { dpm_rule("hypertext key"); }
(' Destroy_Items ', ' Term ', ' ATOM ')
($$ = dpm_build_hypertext_symbol($4,$6,$8); }
| GRAPHIC_POSITION { dpm_rule("graphic position"); }
(' Destroy_Items ', ' Real ', ' Real ')
($$ = dpm_build_graphics_symbol($4,$6,$8); }
| PLOT_POSITION { dpm_rule("plot position"); }
(' Term ', '[', ' PlotArray ', ']')
($$ = dpm_build_plot_symbol($4,*$8[0],*$8[1],*$8[2]); }

PlotArray : Real ', Real ', Real
{ $$[0] = $1; $$[1] = $3; $$[2] = $5; }
| Real ', Real
{ $$[0] = $1; $$[1] = $3; $$[2] = NULL; }

Disposition : DESTROY { $$ = destroy; }
| HIDE { $$ = hide; }
| ElError { $$ = destroy; }

Panel : '{ ( dpm_rule("Panel row"); )PanelRow '}
($$ = $3; )
| Panel ' '{ PanelRow '}
(dpm_add_panel_row($$,$4,updateFlag); }

PanelRow : PanelAio
{ if (!updateFlag) $$ = $1; else $$ = NULL; }
| PanelRow ', PanelAio
{ dpm_add_item_to_panel_row($$,$3,updateFlag); }

PanelAio : TEXT { dpm_rule("text aio"); }
(' Term { dpm_arg($4); }', ' ATOM ', ' TextAttributes'
{ $$ = dpm_text_panel_item($4,$7); }
| INPUT { dpm_rule("input aio"); }
('Term( dpm_arg($4));'),'InitialValue ', 'InputAioAttributeList ')
{ $$ = dpm_input_panel_item($4,$7); }
| CHOICE { dpm_rule("choice aio"); }
(' Term { dpm_arg($4); }
', ' Choices ', ' ChoiceAttributes ')
{ $$ = dpm_choice_panel_item($4,$7); }

Aios : Null { $$ = NULL; }
| VARIABLE { $$ = NULL; }
| AioElement { if (updateFlag) $$ = NULL; else $$ = $1; }
| '{ ' ( dpm_rule("Aios"); } AioList '}
{ if (updateFlag) $$ = NULL; else $$ = $3; }

AioList : AioElement

194
if (updateFlag) $$ = NULL; )
CoreAio

AioList ',' AioElement

{ $$ = dpm_add_aio_to_end($1,$3,updateFlag); }

AioElement

{ $$ = $1; }

| Container
| NewAio /** added by Haishan */
| $$ = $1; )

CoreAio

: INFORM {dpm_rule("inform aio");}

{ InformTerm ',' Presentation ',' InformAioAttributes '}

{ $$ = dpm_inform_aio($4,$6,markup,attachPtr);attachPtr=NULL; }

HYPERTEXT {dpm_rule("hypertext aio");}

{ ' HypertextTerm ',' Presentation ',' HyperTextAttributes '}

{ $$ = dpm_hypertext_aio($4,$6,markup,attachPtr);attachPtr=NULL; }

QUESTION {dpm_rule("question aio");}

{ ' QuestionTerm ',' QuestionContent ',' QuestionAttributeList '}

{ $$ = dpm_question_aio($4,$8,$6,attachPtr);attachPtr =NULL; }

TEXT {dpm_rule("text aio");}

{ ' Term { dpm_arg($4); }',' ATOM ',' TextAttributes '}

{ $$ = dpm_text_aio($4,$7,attachPtr);attachPtr=NULL; }

INPUT {dpm_rule("input aio");}

{ ' Term { dpm_arg($4); }',' InitialValue ',' InputAioAttributeList '}

{ $$ = dpm_input_aio($4,$7,attachPtr);attachPtr=NULL; }

CHOICE {dpm_rule("choice aio");}

{ ' Term { dpm_arg($4); }

', Choices ',' ChoiceAttributes '}

{ $$ = dpm_choice_aio($4,$7,attachPtr);attachPtr=NULL; }

SELECTION {dpm_rule("selection aio");}

{ ' SelectionTerm ',' Selections ',' SelectionAttributeList '}

{ $$ = dpm_selection_aio($4,$8,$6,attachPtr);attachPtr=NULL; }

QuestionContent

: Term

{ strpy($$,"_"); } ;

ChoiceAttributes

: Null

{ '[ ' ATTACHMENT {dpm_rule("attachment");}

', Attachment ']

};

TextAttributes

: Null

{ '[ ' ATTACHMENT {dpm_rule("attachment");}

', Attachment ']

};

Choices

: VARIABLE

{ $$ = NULL; }

{ $$ = NULL; }

{ '[ ' ChoiceList ']

{ $$ = dpm_select_update_choice($2,FALSE); }

UPDATE {' Null '}

UPDATE {' '[ ' ChoiceList ']

{ $$ = dpm_select_update_choice($4,TRUE); }
ChoiceList : (dpm_rule("choiceList");)
  ChoiceItem
  { $$ = $2; }
  |    ChoiceList ',' ChoiceItem
  { dpm_add_next_choice_item($$, $$3); }

ChoiceItem : ATOM
  { $$ = dpm_set_choice($1); }
  |    ATOM
  (dpm_rule("choice item");)
  ' ( STATE ':' State ')
  { $$ = dpm_set_choice_on($1, $$6); }

InformAioAttributes : Null
  { markup = focus ; }
  |    '|' InformAttributes ']

InformAttributes : InformAttribute
  |    InformAttributes ',' InformAttribute

InformAttribute : MARKUP
  { dpm_rule("markup");
    ' :' MarkUpFormat
    |    ATOM ': ' ATOM
    (dpm_arg($1); dpm_rule("inform attributes"); dpm_error_parse($1);)
    |    ATTACHMENT
    { dpm_rule("attachment");
      ' :' Attachment

MarkUpFormat : LATEX
  { markup = latex; }
  |    FOCUS
  { markup = focus; }
  |    ElError
  { markup = focus; }

HyperTextAttributes : Null
  { markup = focus; }
  |    '|' InformAttributes ']

Selections : VARIABLE
  { $$ = NULL; }
  |    '|' (dpm_rule("selections"); cParent=NULL;)
  Children ']
  { $$ = dpm_set_update_selection($3, FALSE); }
  |    UPDATE '(' 'Null' { dpm_rule("update to selections"); } ')' 
  { $$ = NULL; }
  |    UPDATE ' {'
  Children ']
  { dpm_rule("update to selections"); cParent=NULL; }
  Children ']
  { $$ = dpm_set_update_selection($5, TRUE); }

Children : Tree
  |    Children ',' Tree
  { dpm_add_selection_item($1, $$3); }
Tree

```
SelectionItem
  ($$=$1;)
  | Null
  ($$=NULL;)
  | ' ' SelectionItem ' ')
  ($$=$2;)
  | ' ' SelectionItem ',' {
  | push_parent($2); } Children ' ')
  | dpm_link_children($2,$5); $$=$2;)

SelectionItem : Null
  ( $$ = dpm_selection_item("Nil",NULL,NULL,selectionb); )
  | INPUT {
  | dpm_rule("input as selection");
  | ' ' Term ',' InitialValue ',' InputAttributeList ' ')
  ( $$ = dpm_selection_item($4,$6,NULL,inputb); )
  | ATOM
  ( $$ = dpm_selection_item($1,NULL,NULL,selectionb); )
  | INTEGER
  ( $$ = dpm_integer_selection_item($1,NULL,NULL,selectionb); )
  | ATOM ' ')
  | dpm_arg($1); dpm_rule("Selection Args");
  | SelectionArgs ' ')
  ( $$ = dpm_selection_item($1,$2,SelectionArgStruct,selectionb);
NewSelectionArg = TRUE;)

SelectionArgs : SelectionArg
  | SelectionArgs ',' SelectionArg

SelectionArg : STATE
  ( dpm_rule("selection item state");)
  | ' ' State
  | SelectionArgStruct =
  | dpm_set_selection_arg_state(SelectionArgStruct,$4);)
  | EXTERNAL
  | dpm_rule("selection item external");)
  | ' ' External
  | SelectionArgStruct =
  | dpm_set_selection_arg_external(SelectionArgStruct,$4);)
  | ATOM ' ' ATOM
  | dpm_arg($1);dpm_rule("Selection argument");dpm_error_parse($1);)

SelectionAttributeList : Null
  ( $$ = NULL; NewSelection = TRUE; )
  | ' ' ( dpm_rule("selection attribute list");)
NewSelectionAttributes ' ')
  ( $$ = tempSelection; NewSelection = TRUE; )

SelectionAttributes : SelectionAttribute
  | SelectionAttributes ',' SelectionAttribute

SelectionAttribute : STRUCTURE
  ( dpm_rule("selection structure"); ' ' Structure
APPENDICES

{ tempSelection = dpm_set_selection_structure(tempSelection,$4); }
| LABEL
{ dpm_rule("selection label"); }

Term
{ tempSelection = dpm_set_selection_label(tempSelection,$4); }
| LIMITS
{ dpm_rule("selection limits"); }

' (' INTEGER ' UpperLimit ')' 
{ tempSelection = dpm_set_selection_limits(tempSelection,$5,$7); }
| ATOM
' ATOM
{ dpm_arg($1); dpm_rule("selectionattribute"); dpm_error_parse($1); }
| ATTACHMENT
' ATTACHMENT

' : UpperLimit : INTEGER
($$ = -1;);

Structure : NESTED_LIST
{ $$ = nestedList; }
| DIRECTED_GRAPH
{ $$ = directedGraph; }
| MATRIX
{ $$ = matrix; }
| TREE_STRUCTURE
{ $$ = treeStructure; }
| Error
{ $$ = directedGraph; }

/* InputAttributeList used for selection Aio only */

InputAttributeList : Null
| [' InputAttributes ']

InputAttributes: VALIDATE
{ dpm_rule("input validation"); }
'
Validation
| ATOM
' ATOM
{ dpm_arg($1); dpm_rule("input attribute"); dpm_error_parse($1); }

InputAioAttributeList : Null
| [' InputAioAttributes ']

InputAioAttributes : InputAioAttribute
| InputAioAttributes ',' InputAioAttribute

InputAioAttribute: VALIDATE
{ dpm_rule("input validation"); }
'
Validation
| ATOM
' ATOM
{ dpm_arg($1); dpm_rule("input attribute"); dpm_error_parse($1); }
| ATTACHMENT
' ATTACHMENT

Validation : ' (' ATOM ',' Term ',' Presentation ')

External : ICON
{ dpm_rule("external icon"); }
'
Pathname
' Format
{ $$ = dpm_set_external_icon($4,$6); }
| Term
{ $$ = dpm_set_external_text($1); }


QuestionAttributeList : Null
{ $$ = NULL; }
| ' ['
| {dpm_rule("question attributes");}
| QuestionAttributes ' ]'
| if (NewQuestion) $$ = NULL ; else { $$ = tempQuestion; NewQuestion = TRUE; }
|
;

QuestionAttributes : QuestionAttribute
| QuestionAttributes ',' QuestionAttribute
|

QuestionAttribute : VALUE {dpm_rule("question value");}
|:' InitialValue
| { tempQuestion = dpm_set_question_value(tempQuestion,$4); }
| RESPONSE_LENGTH
| {dpm_rule("question response length");}' : ResponseLength
| { tempQuestion = dpm_set_question_response(tempQuestion,$4); }
| VALIDATE
| {dpm_rule("question validation");}:' Validation
| { dpm_set_temp_question(&tempQuestion); }
| ATOM:' ATOM
| { dpm_arg($1); dpm_rule("question attribute");
| dpm_error_parse($1); }
| ATTACHMENT {dpm_rule("attachment");}' : Attachment
|

ResponseLength : PHRASE { $$ = phrase; }
| LINE { $$ = line; }
| PARAGRAPH { $$ = paragraph; }
| PAGE { $$ = page; }
| FILEINPUT { $$ = fileInput; }
| ElError { $$ = line; }
|

InitialValue : VARIABLE { $$ = NULL; }
| Term
| { $$ = dpm_set_initial_value($1,untouched2); }
| APPEND {dpm_rule("question value append");}
| ( atom )'
| ( $$ = dpm_set_initial_value($4,append); }
| INSERT {dpm_rule("question value insert");}
| ( atom )'
| ( $$ = dpm_set_initial_value($4,insert); }
|

State : ON { $$ = on; }
| OFF { $$ = off; }
| ElError { $$ = on; }
|

Presentation : VARIABLE
{ $$ = dpm_atom_presentation(" ",atom); }
| ATOM
| { $$ = dpm_atom_presentation($1,atom); }
| ( atomlist in presentation' );
| ATOMList ' ]'
| ( $$ = dpm_atom_presentation($3,atomlist); }
| TEXTFILE {dpm_rule("inform textfile");}
| ( Pathname ' ]'
| ( $$ = dpm_atom_presentation($4,textfile); }
| GRAPHIC {dpm_rule("inform graphic");}
|
APPENDICES

"(' Term ', ' Pathname ', ' Format ')
{ $$ = dpm_graphic_representation($4,$6,$8); }
| PLOT
{dpm_rule("inform plot");}
"(' Term ', ' PlotDetail ')
{ $$ = dpm_plot_representation($4,CurrentPlotPtr,$6);
    NewPlot = TRUE; }
;

PlotDetail :
| VARIABLE ( $$ = NULL; }
| PLOTFILE  {dpm_rule("inform plotfile");}
| ' Pathname '
{ $$ = dpm_plot_data_file($4); }
| '
{dpm_rule("plot data list");}
PlotDataList($3); }
|
PlotData :
| ' Real ', ' Real ', ' Real ')
{ $$ = dpm_build_plot_point($2,$4,$6); }
| ' Real ', ' Real ')'
{ $$ = dpm_build_plot_point($2,$4,0.0); }
;
AtomList :
| ATOM
{ strcpy($$, $1); }
| AtomList ', ATOM
{ strcpy($$, $1); strcat($$, " "); strcat($$, $3); }
;
PlotAttributes :
| PlotAttribute
| PlotAttributes , PlotAttribute
;
PlotAttribute :
| PLOT_TYPE {dpm_rule("plot type");}
|':
| PlotType
{ CurrentPlotPtr = dpm_set_plot_type(CurrentPlotPtr,$4); }
| MAX_VALUES {dpm_rule("plot max values");}
| ' ValueList '}
{ CurrentPlotPtr = dpm_set_plot_max_values(CurrentPlotPtr,$5); }
| MIN_VALUES {dpm_rule("plot min values");}
| ' ValueList '}
{ CurrentPlotPtr = dpm_set_plot_min_values(CurrentPlotPtr,$5); }
| PLOT_LABELS {dpm_rule("plot labels");}
| ' Labels '}
{ CurrentPlotPtr = dpm_set_plot_labels(CurrentPlotPtr,$5); }
| CURVES {dpm_rule("curves");}' INTEGER
| CurrentPlotPtr = dpm_set_plot_curves(CurrentPlotPtr,$4); }
| ATOM : ATOM
| dpm_arg($1); dpm_rule("plot attributes"); dpm_error_parse($1); }
;
PlotType :
| TWO_D ( $$ = two_d; }
| THREE_D { $$ = three_d; }
| CONTOUR { $$ = contour; }
| ElError { $$ = two_d; }
;
ValueList :
| INTEGER
| $$ = dpm_set_value_list(NULL,$1); }

200
APPENDICES

ValueList : ' INTEGER
{ $$ = dpm_set_value_list($1,$3); }

Labels : Term ',' Term ',' Term
{ $$ = dpm_set_labels($1,$3,$5); }  
| Term ',' Term
{ $$ = dpm_set_labels($1,$3," "); }  

Format : XWD { $$ = xwd; }
| POSTSCRIPT { $$ = postscript; }
| BITMAP { $$ = bitmap; }
| MACPAINT { $$ = macpaint; }
| ElError { $$ = bitmap; }

Pathname : ATOM

Term : Real
{ sprintf($$,"%f",$1); }
| INTEGER
{ sprintf($$,"%d",$1); }
| ATOM '{'TermList'}'
{ strcpy($$,$1); strcat($$," "); strcat($$,$3); strcat($$," "); }  
| ATOM
{ strcpy($$,$1); }
| '{'TermList'}'
{ strcpy($$," "); strcat($$,$2); strcat($$," "); }

ElError : Term { dpm_error_parse($1); }

GroupTerm : Term
{ dpm_get_container($1,&($$),groupc,Owner); }

PanelTerm : Term
{ dpm_get_container($1,&($$),panelc,Owner); }

QueryString : Term
{ dpm_get_container($1,&($$),queryc,Owner); }

InformTerm : Term
{ dpm_arg($1);
dpm_get_inform($1,&($$),inform); }

HypertextTerm : Term
{ dpm_arg($1); dpm_get_inform($1,&($$),hypertext); }

REAL01

201
QuestionTerm : Term  
    { dpm_arg($1);
    dpm_get_question($1,&($$)); }

SelectionTerm : Term  
    { dpm_arg($1);
    dpm_get_selection($1,&($$)); }

NewAio : NEW_AIO  
    { dpm_rule("new aio"); }
    (' NewAioId ', ' NewAioTerm ', ' NewAioAttributeList ')
    { fprintf(stderr,"end of new aio !\n");
      $$=dpm_new_aio($6,$8,attachPtr); attachPtr=NULL;tempNewAio=NULL ;}

NewAioId : Term  
    { dpm_arg($1); strcpy(new_aio_id,$1); }

NewAioTerm : ATOM  
    { dpm_arg($1); fprintf(stderr,"newAIO_Name: %s
",$1);
      tempNewAio = dpm_get_new_aio(new_aio_id,$1,&($$)); }

NewAioAttributeList : Null  
    { $$ = tempNewAio; }
    [' NewAioAttributes ']
    { $$ = tempNewAio;tempObject=NULL; }

NewAioAttributes : NewAioAttribute  
    | NewAioAttributes ',' NewAioAttribute

NewAioAttribute : ObjectName ':' ObjAttributeList

ObjectName : ATOM  
    { fprintf(stderr," object_name: %s
",$1);
      tempObject = dpm_name20bject(tempNewAio->p_objects,$1);}

ObjAttributeList : Null  
    [' ObjAttributes ']

ObjAttributes : ObjAttribute  
    | ObjAttribute ',', ObjAttributes

ObjAttribute : ResourceName ':' ResourceValue  
    { if ( tempObject != NULL )
      dpm_changeDefaultPresentation(tempObject,$1,$3); }

ResourceName : ATOM  
    { fprintf(stderr," resource_name: %s",$1); strcpy($$, $1); }

ResourceValue : ATOM
APPENDICES

{ fprintf(stderr," resource_value: %s \n",$1);strcpy($$,\$1);}  

<table>
<thead>
<tr>
<th>INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>}</td>
</tr>
<tr>
<td>NewAioAttachment : Null</td>
</tr>
<tr>
<td>[ ' ATTACHMENT { dpm_rule(&quot;attachment&quot;);}</td>
</tr>
<tr>
<td>':' Attachment ']'</td>
</tr>
<tr>
<td>Null : NIL</td>
</tr>
<tr>
<td>[ ' ' ]</td>
</tr>
</tbody>
</table>

%

A.3 Lex Program List

/* file name: mparse.lex */
%
#include <math.h>
#include <stdio.h>

#define return_token (x) return x
#define return_string_token(x) {
    strcpy(yylval.string,yytext);
    dpm_token(yytext,"x");
    return x;
}
#define return_stoken(x) {
    dpm_substitute_nl(yytext,yylval.string);
    dpm_token(yytext,"x");
    return x;
}
#define return_float_token(x) {
    yylval.real=atof(yytext);
    dpm_token(yytext,"x"); return x;
}
#define return_int_token(x) {
    yylval.integer=atoi(yytext);
    dpm_token(yytext,"x");
    return x;
}

int line_no = 1;
/* Force lex to take its input from InputPtr instead of stdin */
#undef input
#define input() 
    ( ((yytchar = yysptr > yysbuf ? U(*--yysptr) : *InputPtr++) == 10 
    ? (yylineno++,yytchar):yytchar) == EOF ? 0 : yytchar)
%

$p 5000
%a 4000
%n 1000
%e 2000
%o 4000
%

203
above 
append 
attachment 
behaviour 
below 
bitmap 
child_of 
choice 
contour 
controls 
curves 
destroy 
directed_graph 
directory 
disabled 
disposal 
disposition 
dvi 
explanation_of 
extension_of 
external 
focus 
file_input 
graphic 
graphic_position 
group 
hide 
hypertext 
hypertext_key 
icon 
if_valid 
immediate 
in_response_to 
independent 
input 
inform 
insert 
label 
latex 
limits 
line 
macpaint 
markup 
matrix 
max_values 
min_values 
minor_controls 
modal 
modeless 
next_to 
notell
A.4 AIO Data Structure Definitions

/*------------------------------------------*/
*
* Filename:  pd_struc.h

%
* Description: Contains data structures that are used by dprn to pass
* on the aios to ppl
* (c) Copyright 1993 by the FOCUS Consortium
*
*-----------------------------------------------------------------*/

#ifndef _app_structures_h
#define _app_structures_h

#define FALSE 0
#define TRUE 1
#define BOOLEAN short

typedef enum {unset1,modal,modeless} behaviour_t;
typedef enum {unset2,hide,destroy} disposition_t;
typedef enum {unset3,untouched,update} bstatus_t;
typedef enum {unset4,minorcontrol,none,control} carriage_t;
typedef enum {unset5,true,false} validate_t;
typedef enum {unset6,immediate,ifValid,postponed} disposal_t;
typedef enum {unset7,matrix,directedGraph,nestedList,treeStructure} structure_t;
typedef enum {unset8,on,off,disabled} able_t;
typedef enum {unset9,unlimited,integer} upper_limit_t;
typedef enum {unset10,line,paragraph,page,phrase,file input} response_t;
typedef enum {unset11,untouched,append,insert} value_t;
typedef enum {unset12,atom,atomlist,textfile,graphic,plot} informat_t;
typedef enum {unset13,independent,extension_of,sibling_of,child_of,explanation_of,in_response_to} relation_t;
typedef enum {unset14,selection,inform,hypertext,question,text,input,choice,group,query,panel,new aio} aio_list_t;
typedef enum {unset15,text,input,choice} element_type_t;
typedef enum {unset16,group,panel,query} container_t;
typedef enum {unset17,xwd,postscript,bitmap,cgm,macpaint} file_format_t;
typedef enum {unset18,text,graphics,icon} graphics_text_t;
typedef enum {unset19,two_d,three_d,contour} plot_type_t;
typedef enum {unset20,file,latex} plot_format_t;
typedef enum {unset21,latex,icon} markup_t;
typedef enum {unset22,selectionb,inputb} selection_type_t;
typedef enum {unset23,hypertexts,graphics2,plots} symbol_type_t;
typedef enum {unset24,next_to,above,below,alone} aio_relation_t;
typedef enum {menu,control,icon} symbol_t;

typedef struct _limit
{
    int lower;
    int upper;
} limit_t;

/* Icon is the same as graphics, but without an ID */

typedef struct _graphic_format
typedef struct _external {
    graphics_text_t graphics_text; /* text or graphics */
    union {
        term_t text; /* label to be displayed */
        graphic_format_t graphics; /* pointer to the graphics */
    } typeof;
} external_t;

typedef struct _selection_arg {
    able_t able; /* on, off or disabled */
    external_t external;
} selection_arg_t;

typedef struct _controllist {
    term_t id; /* the control id */
    Widgetid_t Handle; /* the control widget id */
    selection_arg_t *selection; /* pointer to external appearance */
    disposal_t disposal; /* hide or display */
    validate_t validate; /* validate the content if set to true */
    struct _controllist *nextcontrol; /* pointer to the next control */
} controllist_t;

typedef struct _control_container {
    Widgetid_t Handle; /* widget id of the container of the controls */
    bstatus_t bstatus; /* untouched or update */
    BOOLEAN replace; /* replace existing controls */
    controllist_t *controls; /* pointer to the first control */
} control_container_t;

typedef struct _placement_textual {
    term_t id;
    term_t key;
    term_t literal;
} placement_textual_t;

typedef struct _placement_graphic {
    term_t id; /* id of graphics */
    float normx; /* normalized x coordinate */
    float normy; /* normalized y coordinate */
} placement_graphic_t;

typedef struct _placement_plot {
    term_t id; /* id of the plot */
    plot_type_t plot_type; /* 2D or 3D plot */
    float coord[3]; /* coordinate of the point */
} placement_plot_t;

typedef struct _symbol
APPENDICES

{ symbol_type_t symbol_type; /* hypertext, graphics or plot */
  union{
    placement_textual_t textual; /* hypertext text element */
    placement_graphic_t graphic; /* position on the graphics element */
    placement_plot_t plot; /* point on the plot */
  } symbol_kind;
} symbol_t;

typedef struct _placement
{
  relation_t relation; /* independent, extension_of, .... */
  union{
    term_t aoid; /* id of the aio which is related to */
    symbol_t *symbol; /* pointer to an active area */
  } argument;
} placement_t;

typedef struct _attachment_t
{
  aio_relation_t attach; /* next_to, below or alone */
  term_t toAio; /* id of the other aio */
  struct _multiple_t *Aio; /* pointer to the other aio */
} attachment_t;

typedef struct _menu_list
{
  term_t id; /* id of the minor control */
  term_t ancestor; /* ancestor list */
  Widgetid_t Handle; /* widget id of the minor control */
  selection_arg_t *selection; /* appearance of the minor control */
  struct _menu_list *submenu; /* pointer to its children */
  struct _menu_list *nextitem; /* pointer to its siblings */
} menu_list_t;

typedef struct _menu
{
  Widgetid_t Handle; /* widget id of the container of minors */
  bstatus_t bstatus; /* untouched or update */
  menu_list_t *content; /* pointer to the first minor control */
} menu_t;

typedef struct _initial_value
{
  term_t atom; /* text to be displayed */
  value_t value; /* append or insert where the cursor is */
} initial_value_t;

/**********************************************************
* Input Aio Structure
**********************************************************/

typedef struct _input
{
  term_t id; /* id of the input aio */
  Widgetid_t Handle; /* widget id of the input aio */
  bstatus_t bstatus; /* untouched or update */
  initial_value_t *initial_value; /* initial text to display and position of it */
  validate_t validate; /* if true validate the input */
  term_t validation; /* how to validate */
  attachment_t *attach; /* how it is related to the other
APPENDICES

typedef struct _text
{
    term_t id; /* id of the text aio */
    Widgetid_t Handle; /* widget id of the text aio */
    bstatus_t bstatus; /* untouched or update */
    term_t atom; /* text to be displayed */
    attachment_t *attach; /* how it is related to the other aios */
} text_t;

typedef struct _choices
{
    term_t id; /* id of the choice item */
    Widgetid_t Handle; /* widget id of the choice item */
    bstatus_t bstatus; /* untouched or update */
    BOOLEAN replace; /* replace existing choices */
    callback_string; /* not used */
    struct _choices *next_choice; /* pointer to the next item */
} list_choices_t;

typedef struct _choice
{
    term_t id; /* id of the choice aio */
    Widgetid_t Handle; /* widget id of the choice aio */
    bstatus_t bstatus; /* untouched or update */
    list_choices_t *p_choices; /* pointer to the first choice item */
} choice_t;

typedef struct _values
{
    int value; /* maximum or minimum value */
    struct _values *next; /* next value */
} values_t;

typedef struct _coordinates_t
{
    point xyz; /* coordinate of the point */
    struct _coordinates_t *next; /* next point */
} coordinates_t;

typedef struct _plot_data
{
    plot_format_t format; /* data in a file or in the message */
    union{
        coordinates_t *data; /* pointer to the first point */
        term_t name; /* name of the file */
    } typeof;
}
typedef struct _plot {
    term_t id; /* id of the plot */
    plot_type_t plot_type; /* 2D or 3D plot */
    plot_data_t *plot_data; /* pointer to the coordinates of the
       points */
    values_t *max_values, /* maximum value for each curve on the
       plot */
    *min_values; /* minimum value for each curve on the
       plot */
    term_t *labels; /* label for each axis */
    int no_of_curves; /* number of curves on the plot */
} plot_t;

typedef struct _inform_presentation {
    informat_t informat; /* atom, atomlist, textfile, graphics or
       plot */
    union{
        term_t name; /* text to be displayed for atom or atomlist,
           if textfile this is the name of the file */
        graphic_format_t graphics; /* graphics information */
        plot_t plot; /* plot information */
    } inform_presentation_t;
} inform_presentation_t;

typedef struct _inform {
    term_t id; /* id of the inform aio */
    Widgetid_t Handle; /* widget id of the inform aio */
    bstatus_t bstatus; /* untouched or update */
    inform_presentation_t presentation; /* what is to be displayed */
    markup_t markup; /* focus or latex markup */
    attachment_t *attach; /* how it is related to other aios */
} inform_t;

/* Question Aio Structures */

typedef union _carriage_control {
    menu_list_t *p_menu; /* pointer to the minor control item */
    controllist_t *p_control; /* pointer to the control item */
} carriage_control_t;

typedef struct _question {
    term_t id; /* id of the question aio */
    Widgetid_t Handle; /* widget id of the question aio */
    bstatus_t bstatus; /* untouched or update */
    term_t label; /* question text */
    initial_value_t *initial_value; /* initial text to be
       displayed and position of it */
    response_t response; /* line, paragraph, phrase or page */
    carriage_t carriage; /* control, minor control or none */
    carriage_control_t which_control; /* the structure of the control
       that the carriage return is associated with */
    validate_t validate; /* if true validate the input */
    attachment_t *attach; /* how it is related to other aios */
} question_t;
APPENDICES

typedef struct _edges
{
    int direction;  /* direction of the edge */
    int cycle;  /* 0=no cycle; 1=cycle; 2=dummy node*/
    struct _dgLayout *thisNode;  /* the node to draw the edge to */
    struct _edges *next;  /* next node */
}edges_t;

typedef struct _dgLayout
{
    char *id;  /* id of the node */
    Widgetid_t Handle;  /* id of the widget/window */
    int width,  /* width of the node */
    height;  /* height of the node */
    int x,  /* x coordinate */
    y;  /* y coordinate */
    int hLevel;  /* horizontal position */
    int vLevel;  /* vertical position */
    int cycle;  /* 0=no cycle; 1=cycle; 2=dummy node*/
    BOOLEAN drawn;  /* TRUE if it has been drawn */
    edges_t *edges;  /* list of the edges */
}dgLayout_t;

typedef struct _selectionitem
{
    term_t id;  /* id of the selection aio */
    Widgetid_t Handle;  /* widget id of the selection aio */
    bstatus_t bstatus;  /* untouched or update */
    selection_type_t selection_type;  /* input or a selection item */
    union {
        input_t *input_struct;  /* pointer to input structure */
        selection_arg_t *args;  /* pointer to a selection item */
    }type_of;
}selection_item_t;

typedef struct _ParentSize_t
{
    selection_item_t *node;
    struct _ParentSize_t *next;
}ParentSize_t;

typedef struct _selection
{
    term_t id;  /* id of the selection aio */
    Widgetid_t Handle;  /* widget id of the selection aio */
}
typedef struct _variable
{
  term_t  v_name;
  term_t  v_value;
  struct _variable  *v_next;
}variable_t;

typedef struct _action
{
  term_t  a_name;  /* action name */
  struct _action  *a_next;
}action_t;

typedef struct _callback
{
  term_t  c_type;  /* type of callback */
  variable_t  *c_variable;
  action_t  *c_action;
  struct _callback  *c_next;
}callback_t;

typedef struct _presentation
{
  term_t  p_name;  /* name of resource */
  term_t  p_value;  /* value of resource */
  struct _presentation  *p_next;  /* next resource */
}presentation_t;

typedef struct _object
{
  term_t  o_name;  /* name of object */
  Widgetid_t  o_wid;  /* window id */
  int  o_type;  /* window type */
  Boolean  o_update;  /* update flag */
  struct _object  *o_parent;  /* parent of object */
  struct _object  *o_child;  /* the first child of object */
  struct _object  *o_brother;  /* the next brother object */
  presentation_t  *o_presentation;  /* presentation of object */
  callback_t  *o_callback;  /* callback of object */
}object_t;

typedef struct _new_aio
{
  term_t  name;  /* name of new aio */
  term_t  id;  /* id of new aio */
  bstatus_t  bstatus;  /* update */
  Widgetid_t  Handle;  /* window id */
  object_t  *p_objects;
  attachment_t  *attach;
  callback_t  *return_actions;  /* for getting newAio return
typedef struct _aio_structure {
  term_t id;        /* id of the container aio */
  term_t Owner;     /* owner of the aio */
  Widgetid_t Handle; /* widget id of the container aio */
  term_t ancestor_ids; /* ancestor ids of the aio */
  behaviour_t behaviour; /* modal or modeless */
  term_t title;      /* title of the aio */
  disposition_t disposition; /* destroy or hide */
  attachment_t *attach; /* if it is a sub container how it is related to other aios */
  placement_t *place; /* if top container how it is related to other top containers */
  control_container_t *control_container; /* list of controls */
  menu_t *p_minorcontrols; /* list of minor controls */
} aio_structure_t;

typedef struct _query {
  term_t label;    /* text of the query */
  aio_structure_t *aioStructure; /* pointer to its attributes */
} query_t;

typedef struct row {
  element_type_t bstatus;  /* input, choice or text */
  bstatus_t untouched_or_update; /* untouched or update */
  union{
    text_t *text_aio;  /* pointer to text aio */
    input_t *input_aio; /* pointer to input aio */
    choice_t *choice_aio; /* pointer to choice aio */
  } panel_kind;
  struct _row *next_item; /* pointer to the next item in the row */
} panel_row_t;

typedef struct _panel {
  panel_row_t *row; /* pointer to the first aio in the first row*/
  aio_structure_t *aioStructure; /* pointer to its attributes */
} panel_t;

* Attributes of container Aio Structure */

typedef struct _aio_structure
{
  term_t id;                        /* id of the container aio */
  term_t Owner;                     /* owner of the aio */
  Widgetid_t Handle;                /* widget id of the container aio */
  term_t ancestor_ids;              /* ancestor ids of the aio */
  behaviour_t behaviour;            /* modal or modeless */
  term_t title;                     /* title of the aio */
  disposition_t disposition;        /* destroy or hide */
  attachment_t *attach;             /* if it is a sub container how it is related to other aios */
  placement_t *place;               /* if top container how it is related to other top containers */
  control_container_t *control_container; /* list of controls */
  menu_t *p_minorcontrols;          /* list of minor controls */
} aio_structure_t;

* Query Aio Structure */

typedef struct _query
{
  term_t label;                     /* text of the query */
  aio_structure_t *aioStructure;    /* pointer to its attributes */
} query_t;

* Panel Aio Structure */

typedef struct _row
{
  element_type_t element_type;       /* input, choice or text */
  bstatus_t bstatus;                /* untouched or update */
  union{
    text_t *text_aio;              /* pointer to text aio */
    input_t *input_aio;            /* pointer to input aio */
    choice_t *choice_aio;          /* pointer to choice aio */
  } panel_kind;
  struct _row *next_item;           /* pointer to the next item in the row */
} panel_row_t;

typedef struct _panel
{
  panel_row_t *row;                 /* pointer to the first aio in the first row */
  aio_structure_t *aioStructure;    /* pointer to its attributes */
} panel_t;

* Group Aio Structures */
typedef struct _multiple {
  aio_list_t aio_list; /* selection, inform, or .... */
  bstatus_t bstatus; /* untouched or update */
  union{
    selection_t *p_selection; /* selection aio */
    inform_t *p_inform; /* inform or hypertext aio */
    question_t *p_question; /* question aio */
    text_t *p_text; /* text aio */
    input_t *p_input; /* input aio */
    choice_t *p_choice; /* choice aio */
    struct _frame_t *p_frame; /* container aio */
    new_aio_t *p_new_aio; /* added by Haishan */
  } next_block;
  struct _multiple *next; /* next aio */
} multiple_t;

typedef struct _group {
  multiple_t *aio_type; /* list of aios */
  aio_structure_t *aioStructure; /* pointer to its attributes */
} group_t;

/****************************************
* frame_t : Structure for containing
*    container aios
*/

typedef union _kind_container {
  group_t *group_container; /* group aio */
  query_t *query_container; /* query aio */
  panel_t *panel_container; /* panel aio */
} frame_type_t;

typedef struct _frame_t {
  bstatus_t bstatus; /* untouched or update */
  container_t container; /* group, panel or query */
  frame_type_t frame; /* corresponding structure */
} frame_t;

#endif

typedef struct _multiple {
  aio_list_t aio_list; /* selection, inform, or .... */
  bstatus_t bstatus; /* untouched or update */
  union{
    selection_t *p_selection; /* selection aio */
    inform_t *p_inform; /* inform or hypertext aio */
    question_t *p_question; /* question aio */
    text_t *p_text; /* text aio */
    input_t *p_input; /* input aio */
    choice_t *p_choice; /* choice aio */
    struct _frame_t *p_frame; /* container aio */
    new_aio_t *p_new_aio; /* added by Haishan */
  } next_block;
  struct _multiple *next; /* next aio */
} multiple_t;

typedef struct _group {
  multiple_t *aio_type; /* list of aios */
  aio_structure_t *aioStructure; /* pointer to its attributes */
} group_t;

/****************************************
* frame_t : Structure for containing
*    container aios
*/

typedef union _kind_container {
  group_t *group_container; /* group aio */
  query_t *query_container; /* query aio */
  panel_t *panel_container; /* panel aio */
} frame_type_t;

typedef struct _frame_t {
  bstatus_t bstatus; /* untouched or update */
  container_t container; /* group, panel or query */
  frame_type_t frame; /* corresponding structure */
} frame_t;

#endif