

1

A USER INTERFACE FOR A PROGRAMMING ENVIRONMENT

Sami Boulos

School of Computer Science
McGill University, Montréal

July 1990

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

© Sami Boulos, 1990

Abstract

Poor user interfaces in programming environments detract from environments' power and ineffectively communicate with users. Moreover, specification, development, testing, and modification of these usually large, complex, and handcrafted user interfaces are difficult, error prone, slow, and costly. An alternative is user-interface generation. This thesis deals with two primary issues: *xmupe2*, a user-friendly user interface for the MUPE-2 programming environment, and user-interface generation. Implemented in Modula-2 and C for the X Window System, *xmupe2* shows MUPE-2's character with: windows tailored to program-fragments, textual and graphical representations of fragments' contents, and editing commands fired by context-sensitive mouse-based menus. Secondly, and because of the effort in handcrafting *xmupe2*, the thesis introduces MUISL, an experimental event-driven user-interface specification language. MUISL defines user-interface objects with inheritable classes, attributes, and actions. Then presented is *mugen* — a flexible, table-driven, and MUISL-based user-interface generator. Both MUISL and *mugen* simplify user-interface development, as exemplified in the thesis.

Résumé

Dans les environnements de programmation, de pauvres interfaces usager atténuent la puissance des environnements et communiquent d'une manière inefficace avec les usagers. De plus, la spécification, le développement, le test et la modification de ces généralement larges, complexes et artisanales interfaces usager sont non seulement difficiles mais tout à la fois sources d'erreurs, lents et coûteux. Une alternative réside dans la génération d'interfaces usager. Cette thèse traite de deux sujets principaux. *xmupe2* une interface usager conviviale pour l'environnement de programmation MUPE-2 et la génération d'interfaces usager. *Xmupe2* qui est implémenté en Modula-2 et C pour le système X Window visualise les caractéristiques de MUPE-2 à l'aide: de fenêtres adaptées aux fragments de programme, de représentations textuelles et graphiques du contenu des fragments et de commandes d'éditations exécutées par l'intermédiaire de menus dépendants du contexte, basés sur l'utilisation d'une souris. D'autre part, cette thèse présente MUISL, un sous produit de la conception de *xmupe2*. MUISL est un langage de spécification d'interfaces usager expérimental contrôlé par des événements. MUISL définit des objets d'interface usager avec des classes héritables, des attributs et des actions. Puis, nous présentons *muigen* ou générateur d'interfaces usager, basé sur MUISL, flexible et contrôlé par tables. MUISL et *muigen*, comme exemplifié dans cette thèse, simplifient le développement d'interfaces usager.

Acknowledgements

I am indebted to my supervisor, Professor Nazim H. Madhavji, for his constant assistance, advice, and encouragement. His guidance was indispensable in innumerable instances.

I have had the support and friendship of the members of the MUPE-2 team: Jules Desharnais, Yuan Xiang Gu, Kamel Toubache, and Mingjun Zhang. They were always willing to spend the time to assist me in any way. For the French version of the abstract, I thank both Jean-François Girard and Kamel Toubache. For his assistance with the X Window System, I thank Alan Emtage. I also appreciate the technical and administrative support of the School of Computer Science. Finally, I thank my family for their unflagging support and encouragement throughout all my years of study; I especially thank my sister, Yola, for proofreading the initial versions of this thesis.

This work was financially supported by a scholarship from the Natural Sciences and Engineering Research Council of Canada.

Contents

Abstract	ii
Résumé	iii
Acknowledgements	iv
1 Introduction	1
1.1 Problem Definition	1
1.2 Thesis Area and Goals	2
1.3 Contributions of the Thesis	3
1.4 Methodology	5
2 Background and Related Work	6
2.1 User Interfaces in Programming Environments	6
2.2 Architectural Models of User Interfaces	9
2.3 User Interface Guidelines	11
2.4 The MUPE-2 Programming Environment	13
3 The MUPE-2 User Interface: An Overview	17
3.1 User-Interface Requirements	17
3.2 Xmupe2: The X Window System MUPE-2 User Interface	22
3.2.1 Window Structures	22
3.2.2 Design	28
3.2.3 Implementation	29
4 Unparsing and the User Interface	32

4.1	Textual Unparsing	33
4.2	Graphical Unparsing	35
5	Cursors and the User Interface	37
5.1	Cursors in Xmupe2	37
5.1.1	The Mouse and Textual Cursors	38
5.1.2	The Structured Cursor	38
5.1.3	Design and Implementation	40
5.2	Cursor Movements	42
5.2.1	User's View	43
5.2.2	Design and Implementation	46
6	Menus and the User Interface	49
6.1	Menu Design Issues	49
6.2	Menus in Xmupe2	50
6.2.1	Using the Menus	51
6.2.2	Design	52
6.2.3	Implementation	54
7	Editing Commands and the User Interface	57
7.1	Editing Scenarios	57
7.1.1	Programming-in-the-Small	58
7.1.2	Programming-in-the-Large	68
7.2	Design and Implementation	71
7.2.1	General Strategy for All Commands	71
7.2.2	Delete	72
7.2.3	Drag	72
7.2.4	Group/UnGroup	73
7.2.5	Inspect/TextEdit	74
7.2.6	Insert	76
8	User Interface Generation	77
8.1	Background and Related Work	78
8.1.1	An Introduction to User Interface Tools	78
8.1.2	Methods of Control	81

8.1.3	Approaches to Specification and Generation	82
8.2	MUISL: The McGill User Interface Specification Language	88
8.2.1	Assumptions and Scope of Work	88
8.2.2	The Language	90
8.3	Muigen: The MUISL-Based User-Interface Generator	112
8.3.1	Definitions	112
8.3.2	Initialization Files	113
8.4	Evaluation of MUISL and Muigen	116
9	Conclusions	124
A	Xmupe2 Architecture	128
B	MUISL Lexical Rules and Grammar	131
B.1	MUISL Lexical Rules	131
B.2	MUISL Grammar	132
C	Muigen Architecture and File Generation	135
C.1	Muigen Architecture	135
C.2	File Generation	137
D	Initialization Files	138
E	Sample MUISL Specifications	151
E.1	Example 1	151
E.2	Example 2	156
	Bibliography	168

List of Tables

2.1	MUPE-2 Fragtypes	15
3.1	<i>Xmupe2</i> Window Structures	23
3.2	Main Window Buttons	25
5.1	The Mouse Cursor's Shapes	39
5.2	Cursor Movement Keys for Program Structures	44
6.1	<i>Xmupe2</i> Menus	51
6.2	EditOps Menu Structure	55
8.1	Reserved Words in MUISL	91
8.2	Special Symbols in MUISL	92
8.3	Special Prefixes in MUISL	92
D.1	Attribute-Name Prefixes	140
D.2	clRoot Attributes	140
D.3	clButton Attributes	141
D.4	clMenu Attributes	141
D.5	clSimpleWindow Attributes	141
D.6	clToggleButton Attributes	141
D.7	clMenuButton Attributes	141
D.8	clSimpleMenu Attributes	142
D.9	clListMenu Attributes	142
D.10	clTextWindow Attributes	142
D.11	clScrollbarWindow Attributes	143
D.12	clBoxWindow Attributes	143
D.13	clPanedBoxWindow Attributes	144
D.14	clFormBoxWindow Attributes	144
D.15	clViewportWindow Attributes	144

D.16 clDialogueWindow Attributes	145
D.17 clItemSimpleMenu Attributes	145
D.18 Attribute Values	146
D.19 Mouse Buttons	147
D.20 Keys	147
D.21 Event Names	147

List of Figures

2.1	The Seeheim Model	10
3.1	An Overview of <i>Xmupe2</i> 's Window Structures	24
3.2	The Creation of Fragments	26
5.1	Algorithm to Update the Textual Structured Cursor	41
5.2	Cursor Movements in a PIS Window	45
5.3	Cursor Movements in a PIL Window	47
5.4	Algorithm to Move the Structured Cursor	48
6.1	EditOps Menus	53
7.1	Programming-in-the-Small Editing Scenarios	59
7.1	Programming-in-the-Small Editing Scenarios	60
7.1	Programming-in-the-Small Editing Scenarios	61
7.1	Programming-in-the-Small Editing Scenarios	62
7.2	Grouping and Deleting Program Structures	65
7.3	Programming-in-the-Large Editing Scenarios	69
7.3	Programming-in-the-Large Editing Scenarios	70
7.4	Algorithm to Drag by Mouse	73
7.5	Algorithm to Group by Mouse	75
8.1	UIMS Architecture	79
8.2	Class Hierarchy	98
8.3	Non-Operation Initialization-File Grammar	114
8.4	Operation Initialization-File Grammar	115
A.1	The User Interface and Computational Component	128
A.2	<i>Xmupe2</i> Modular Decomposition	129
C.1	<i>Mugen</i> Modular Decomposition	136
E.1	Example 1 Interface	152

E.2 Example 2 Interface	156
-----------------------------------	-----

Chapter 1

Introduction

A user interface, or human computer interface, is the user's view of a system and the domain of discourse between a user and machine [38]. Nievergelt [59] defines a user interface as consisting of an input language (user's input) and an output language (what the user sees). Hartson and Hix [27] view a user interface as the software and hardware through which a human computer dialogue, or observable two-way exchange of symbols and actions, occurs.

1.1 Problem Definition

Poor user interfaces in programming environments detract from environments' power and ineffectively communicate with users. These interfaces often fail to answer basic questions such as [59]: *Where am I? What can I do here? How did I get here? Where else can I go and how do I get there?* Other characteristics include a complex input language with convoluted and cryptic commands; a user-hostile output language having useless error messages and little or no help; a recovery mechanism lacking undo facilities and failing to confirm dangerous commands (such as a delete); an inconsistent use of windows, menus, and other objects interacting with a user; and/or incorrect state information related to the current command and data environments. An example of the final characteristic is a user interface that does not correctly reflect internal programming environment changes affecting the user interface.

In addition, another major problem is that the specification, design, implementation, testing, and modification of the usually large, complex, and handcrafted user interfaces for programming environments are difficult, error prone, often slow, and costly. Consequently,

prototyping is sometimes used to develop such user interfaces. Moreover, an alternative to handcrafting these user interfaces is generating them.

1.2 Thesis Area and Goals

This thesis deals with two different, yet related, aspects of the author's work: a user-friendly user interface for a programming environment and a specification language to generate similarly styled interfaces that are not necessarily limited to programming environments.

The first goal was to design and build a user-friendly user interface that fulfilled the requirements of the current state of the MUPE-2 programming environment [13, 16]. This thesis presents the result of this work, *xmupe2* (The X Window System MUPE-2 User Interface) — a user interface that successfully interacts with internal non-user interface MUPE-2 code previously developed by others and incorporates the principles of good user interface design.

This goal is significant because an effective user interface for a programming environment should simplify the learning and use of the environment, better communicate with the user, and reflect the internal state of the environment in a manner comprehensible at a glance. In addition, the proper design of user interfaces is essential to any programming environment. This is because a user interface is a critical component of an environment: a user often judges the environment's quality by its user interface. A poor user interface can ruin a programming environment and effectively thwart the environment's goals of providing better software tools, producing better quality software using these tools, and simplifying the process of software development. As a result of a mediocre user interface, a user will reject a programming environment — regardless of its underlying power and features that the user is unable to exploit; commit numerous and unnecessary errors; become confused and frustrated; and waste time and effort.

The second goal was the result of the slow, difficult, and tedious work of handcrafting *xmupe2*. A simpler and quicker method of specifying, building, and testing user interfaces is to generate them from specifications in a user-interface specification language. Consequently, the goal was to use the valuable experience and knowledge gained from building *xmupe2* in devising an experimental language for the specification of user interfaces, similar to an *xmupe2*-style of interface but not limited to programming environments. The language is experimental, and not a comprehensive production language, because of time and scope

constraints. The intention was only to experiment with key characteristics of a language capable of specifying user interfaces. A secondary goal was to build a sample generator of user interfaces from specifications in this language. This thesis also presents the results of author's work in these areas, mainly MUISL (The McGill User Interface Specification Language) — a *programmer's* language for the specification of user interfaces, and *muigen* (The MUISL-Based User Interface Generator) — a program to generate user interface code from MUISL specifications.

The second goal is significant because traditional user interface software is often large, complex, and difficult to create, debug and modify. A user interface specification language and the generation of a user interface from this language have positive implications for the developers of user interfaces. The first is to remove the concern with low-level details of user interaction and permit the concentration on the design of the high-level form and functionality of a user interface. The second is to simplify and speed-up the implementation, testing, and modification of user interfaces.

This thesis does *not* deal with human-computer dialogue in depth, general man-machine communication issues, behavioral and cognitive aspects of user interface development, psychological models of users, user interface evaluation, command languages, detailed descriptions of MUPE-2 concepts, and descriptions of internal (non-user interface) MUPE-2 algorithms and data structures (referred to as the *computational component*, in the rest of this thesis). Although this thesis gives a brief overview of MUPE-2 concepts, further details can be found in the relevant papers cited in later sections.

1.3 Contributions of the Thesis

The first contribution of this thesis is in the complete design and implementation of *xmupe2*. The work on this user interface reflects the current state of the MUPE-2 computational component.¹ *Xmupe2* shows the character of a user interface for MUPE-2. This character is part of the program's contributions, and any extensions to the program would add to the implementation, and not to the results of the thesis: the author has laid the foundations in the current version of *xmupe2*. *Xmupe2* has validated MUPE-2 concepts and illustrates that they are effective, practical, and easy to use. Some of these concepts include: multiple

¹The author did not implement any portion of the computational component, but solely designed and implemented *xmupe2*.

windows tailored to program-fragments, the association of screen objects to internal MUPE-2 data structures, the graphical display of node hierarchies and their textual representations, the textual display of other internal program structures, the display of textual and graphical cursors, the management of cursor movements on textual and graphical structures, and editing commands fired by context-sensitive mouse-based menus.

The approximately 14,000 lines of *zmupe2* are written in the programming languages C [40] and Modula-2 [85], using the X Window System [70] (Version 11, Release 4, with the Athena Widget Set [7] and X Toolkit Intrinsics [50]), and running on a Sun-3/50 workstation with Sun UNIX 4.2. The program is consistent in its design and implementation, and consists of a window-system-independent Modula-2 layer that acts as a buffer between the remaining code of user interface and internal (non user-interface) MUPE-2 code; and C code that interfaces with both Modula-2 layer and the window system. C was chosen for its flexibility in interfacing with the X Window System, and because of the local lack of Modula-2 libraries for interfacing with this window system. The X Window System was chosen for its flexibility, portability, availability, and widespread use.

The second contribution of this thesis is the design of MUISL. Though user interface specifications have existed for a number of years, MUISL was designed with the requirements of a typical window-based user-interface, such as *zmupe2*, in mind. It is intended for a programmer and can be used by software developers such as those in the MUPE-2 group. MUISL combines some ideas from other languages and systems, such as Smalltalk [23], the University of Alberta UIMS [25], and the Sassafras UIMS [33], among others. Features of the language include definitions of user interface objects (windows, menus, buttons, and so on), attributes for these objects, and actions that act on them. It also includes powerful features such as classing, and inheritance of attributes and operations. However, MUISL is only an experimental language intended to illustrate basic features of a specification language for user interfaces.

Part of the second contribution of this thesis is to show the viability of MUISL as an experimental specification language, by using it to generate samples of user interface code. This generation is a result of the modular design, and implementation of *mugen*, whose approximately 8,000 lines are written in C and run under Sun UNIX 4.2. Although *mugen* is table driven and is intended to generate code independent of a target programming language or window system, the implemented version was tested for C and the X Window System as the sample target programming language and window system, respectively. Sample user

interface specifications were written in MUISL, passed through *muigen*, and the resultant user interface source code was successfully tested.

1.4 Methodology

Why was *xmupe2* handcrafted instead of generated? The further development of MUPE-2 was dependent on the immediate design and implementation of a user interface. There were real, practical constraints and requirements that necessitated the development of a “real” system such as *xmupe2*: its building could not wait until the possible realization of MUISL and implementation of *muigen*. The solution was to simultaneously handcraft *xmupe2* and learn about user interface generation. As a consequence, *xmupe2* was not delayed, and the experience and knowledge gained from *xmupe2* about user interfaces was useful in designing MUISL and implementing *muigen*. Another factor was the amount of interaction that *xmupe2* required with MUPE-2’s internals. About half of *xmupe2*’s code acts as an interface to the rest of MUPE-2; the other half deals directly with the window system. The first half of *xmupe2* could not have been generated because of its interaction with MUPE-2.

Chapter 2 presents background and related work on user interfaces and programming environments; Chapter 3 discusses user interface requirements and overviews *xmupe2*; Chapter 4 covers unparsing issues relevant to *xmupe2*; Chapter 5 examines the role of cursors and their movements in *xmupe2*; Chapter 6 investigates the role of menus in *xmupe2*; Chapter 7 discusses MUPE-2 editing commands in the context of *xmupe2*; Chapter 8 presents an overview of user interface generation, discusses MUISL and *muigen*, and evaluates them; and Chapter 9 concludes the thesis.

Chapter 2

Background and Related Work

Because this thesis mostly deals with a *user interface* for the MUPE-2 *programming environment*, it is necessary to present related work for user interfaces of programming environments, architectures of user interfaces, user interface guidelines, and a brief overview of the MUPE-2 programming environment. Relevant MUPE-2 concepts not covered in this chapter are presented in other chapters.

2.1 User Interfaces in Programming Environments

This section surveys the user interfaces of selected programming environments. The survey is not exhaustive; other papers such as Nørmark's [60], discuss programming environments in more depth.

Smalltalk [81] is an environment for Smalltalk-80, an object-oriented language [23]. This graphical, integrated, and interactive programming environment incorporates a window manager and a mouse-based user-interface that contains pop-up menus to execute operations. The environment is user friendly because it considered user interface issues in its design. For example, the environment provides "explain" and "example" facilities and on-line documentation. The basis of integration in the Smalltalk environment is its conceptual model of the screen as a desk with sheets of paper represented by one window per program. Easy movement among windows facilitates activities such as mouse-based modeless editing, browsing, debugging, and program execution.

XS-1 (An EXperimental Integrated Interactive System, Version 1) [2] has facilities such as a tree editor, tree file-system, and a kernel with a front-end central dialogue-processor

that handles and redirects user input to other components. The result is a simple and uniform user interface that can answer questions such as “Where am I?” and others posed in Chapter 1. XS-1 utilizes a bit-mapped screen divided into five size-adjustable and non-overlapping windows always visible in any part of the screen. Data in these screens answer the previously posed questions.

Emily [26] is a template-based and syntax-directed editor — an editor that ensures the syntactic integrity of a program by tightly adhering to a grammar. The user interface permits a user to select a programming-language construct from a menu that contains all possible derivations of the current nonterminal. The system then substitutes this construct for the program structure on which the cursor is located. Although Emily is not a programming environment, it is the precursor of systems such as the Cornell Program Synthesizer and MENTOR [13].

The Cornell Program Synthesizer (CPS) [77] is a syntax-directed editor and programming environment for PL/CS. CPS is a hybrid of a pure syntax-directed editor and text editor: expressions are treated as text instead of tree elements (after the expression is edited as text, it is parsed), and templates are used to generate program constructs that are filled by templates or text. The user does not choose templates from a menu, as in Emily, but displays them by typing the dot character followed by the construct name. Placeholders identify locations where insertions (in templates) are permitted. The first character of the cursor’s current position is highlighted; cursor movements use function keys; and editing operations include delete, insert, and clip. The syntax-directed editor is the core of the programming environment and creates an internal representation that can be used by multiple tools such as a compiler and debugger.

PECAN [68], a programming environment generator for block structured languages, exploits the graphics facilities of workstations. PECAN has features such as semantic and syntactic checking while editing, template- or text-based syntax-directed editing, menus for most commands, and multiple overlapping windows to visualize different processes. The environment uses multiple views of programs and data structures including: a pretty-printed view, Nassi-Schneiderman structured flowchart [57], or graphical module-interaction interconnection diagram.

Cedar [79] is a programming environment whose sophisticated user interface is visually oriented. Design principles [12] underlying this environment include: the Law of Least

Astonishment (the user, who is usually right, should be able to predict a program's behavior), The Principle of Non-preemption (the system should not usurp a user's attention and prerogatives), and fast turnaround (think-bound, not compute-bound, programming is preferable). Icons emphasize user interactions and represent: data structures, text documents, tools and services. The screen is divided into tiles — non-overlapping windows called viewers. A viewer, which can be closed into a labeled icon, contains a fixed menu of buttons that invoke associated operations. Viewers allow concurrency: the user can start one task (per viewer) before finishing the current task, and switch back and forth among tasks such as editing and compiling. Cedar's user interface supports multiple character fonts and graphics facilities, similar to those of the Xerox Star [75]. The environment also contains a structured editor that uses templates and the mouse.

IPSEN's (Integrated Programming Support Environment) [15] user interface [14] reflects the environment's support of the software development process. Unlike other environments, its tools (such as those for static analysis, execution, and editing) are not centered around the syntax-directed editor. Instead, they are a highly integrated set of equivalent tools having a common internal representation [41], which supports the integration and ease of use of these tools. IPSEN avoids overloading the screen by using as few windows as possible in its structured screen layout. A tool in IPSEN presents one or more views (cutouts of external representations of an internal representation), each of which is represented by a window. For example, there are views for execution and editing. A structured cursor highlights the current internal structure of interest, called the internal increment. Cursor movements use the mouse, instead of the keyboard. IPSEN uses menu windows to display the list of valid commands. The keyboard can also be used to input commands. By allowing the application of textual editing at all syntactical levels, the syntax-directed editor increases its user-friendliness.

Like other environments, Magpie (Magnolia Pascal Integrated Environment) [11] has a user interface based on a bit-mapped display. Code Browsers — overlapping windows in which the user can develop programs — display a program's declarations and statements. Editing follows the text model and avoids the often inflexible user interface of the template model. The simplicity and uniformity of the user interface reflect the small number of editing commands. Each of the debugging facilities is window based.

The Display-Oriented Programmer's Assistant [78] provides a user-friendly interface for Interlisp [80] (a programming environment for a dialect of LISP), especially in terms of the

editor. The user interface consists of multiple overlapping windows and pull-down menus for editing, electronic mail, program debugging and execution, and other tasks. The mouse is used to choose menu items and select parts of structures to delete, copy, or insert.

The toolkit-based UNIX programming environment [39] has a multitasking capability, a combination of tools with redirection and pipes, and a file system that supports directory hierarchies and treats files uniformly as a sequence of bytes. The shell, UNIX's command-language interpreter, interacts with the user. In spite of its extensive set of powerful tools, UNIX's unfriendly user interface, manifested by the shell's cryptic commands, has been viewed by some as a drawback of the environment. However, others view UNIX's current user interface as an advantage of the environment.

Xmupe2 has been influenced by some environments, such as Cedar, IPSEN, and Magpie, however, it has novel features of MUPE-2. Like most systems, *xmupe2* exploits a workstation's bit-mapped capabilities to display multiple and independently manipulable windows that can be resized, moved, (de)iconized, and so on. Moving the mouse from one window to another allows the user to effortlessly move from one task to another. Window structures reflect the underlying internal environment structures. Some windows show both textual and graphical views of program structures, whereas others display only textual views. Constantly-displayed menus and pull-down menus are used for non-editing tasks. Context-sensitive and mouse-based pop-up menus display lists of valid editing commands. Most commands operate on the structured cursor and are selected from editing menus. Some commands require additional mouse or keyboard input. *Xmupe2* supports both the structured and textual models of editing that MUPE-2 espouses. By highlighting a cursor per relevant window, *xmupe2* focuses the user's attention on program structures. Finally, *xmupe2* permits key-based cursor movements to support both the editing and browsing of these structures.

2.2 Architectural Models of User Interfaces

Architectural models of user interfaces relate user interfaces to the rest of the application. One of these is the *Seeheim model* [65,25,24], a run-time architectural model of human-computer dialogue. Figure 2.1 [25] shows the three components of this model. The *presentation component* is responsible for the physical representation of the user interface and deals with device dependencies and interaction styles. The *dialogue control component* controls

the processing, sequencing, and structure of the dialogue between the user and the application program. This component acts as a mediator between the other two components, by interpreting events in the presentation component and translating them into events for the application interface model, and vice versa. The *application interface model* defines the interface between the user interface and the rest of the application. Communication to the application is via procedure calls and data structures.

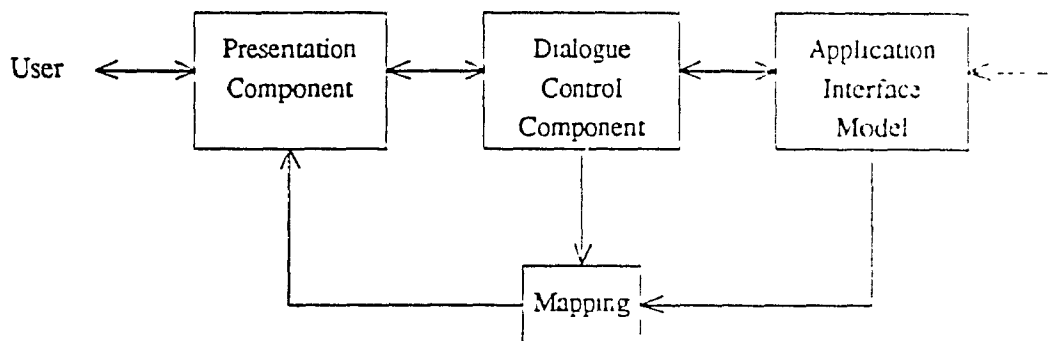


Figure 2.1: The Seeheim Model

The Seeheim model allows the interface to be independent of its application. This independence promotes coarse-grained control over the narrow communication paths between components. In this type of control, an application and its interface rarely communicate. Applications that require frequent fine grain communication may not be suited to this model. For example, a direct manipulation [73] application that tracks a mouse in order to determine semantically legal operations on the object under it, requires constant interface-application communication to provide this semantic feedback.

The Dialogue Management System [28] provides a different architectural model. The system logic of an application in this system has an architecture centered about a *global control component* which has bidirectional communication with each of a *dialogue component* and *computational component*. The global control component is responsible for the correct sequence of invocations of the other two components. The computational component does not deal with dialogue, but deals with the semantic computations of an application. The dialogue component is responsible for dialogue control, dialogue transactions, dialogue-related computations (such as input validation), display of output messages, and so on.

The *dialogue socket model* [9] is centered about a *dialogue socket* --- a high-level abstraction that connects to one or more *dialogue handlers*, on one side, and one or more

applications on another. The user deals with one of these handlers, which is designated to be the current dialogue handler. The dialogue socket acts as a virtual user to an application and maps the lexical and syntactic elements of a dialogue (gathered by the current dialogue handler) to the application's operations.

Xmupe2 follows the Seeheim model. The presentation component of *xmupe2* is responsible for the creation and management of windows, menus, and text and graphics with which the user interacts. Part of the dialogue control component consists of X Window System library routines that detect user events, sequence them, and call user interface routines associated with these events. Other routines of this component call the appropriate application interface component routine and communicate information to presentation component routines that update the display. The application interface component is the only one that directly calls non-user interface MUPE-2 code, the computational component, which deals with internal computations.

The Seeheim model is well suited to *xmupe2* because the actions of responding to a user event, interacting with the computational component, and updating the display, correspond to the components of the model. An important advantage of the model is that it isolates, in one layer, the user-interface code dependent on the computational component. This model also simplifies the front end of the user interface, which displays information without concern for its semantics. The semantics are ensured with the correct calls to the computational component. Appendix A describes *xmupe2*'s architecture in terms of its modular decomposition.

2.3 User Interface Guidelines

What makes a good user interface? A good user interface should make a program easy to learn and use. Schneiderman [72] presents some rules for user interface design, some of which are: consistency at all levels, such as in commands, terminology, menu and display layout, and responses; shortcuts, such as abbreviations, special keys, and macros to accommodate experienced users; undo facilities to reverse actions and protect users from their mistakes; simple error handling and design of the system to prevent users from making a serious error (this includes features such as the confirmation of dangerous commands — delete, erase, and so on); informative feedback for every action; and reduction of short-term memory load by using simple displays, on-line access to relevant documentation, and so on.

Hansen [26] discusses *user engineering principles* that apply to the design of user interfaces and illustrates them with examples from the Emily system. Some principles include know the user, minimize memorization, optimize operations, and engineer for errors. Minimizing memorization involves: selection instead of entry (select, instead of having to type a character string or operation name), names instead of numbers (be able to select from a list of items by name), predictable behavior, and access to system information. Optimization of operations stresses the modes and speeds of user interactions and attempts to reduce the user's interaction effort. This optimization includes the rapid execution of common operations, the display of status messages for lengthy operations, and display inertia which changes the display as little as necessary when carrying out a request. To engineer for errors includes: good error messages to train the novice and remind the expert, the ability to engineer out the common errors, reversible actions, redundancy (back up a powerful operation with combinations of simpler operations), and data structure integrity to guard against loss of valuable user data.

Other authors [59,10,29] provide guidelines for information density, state knowledge, command languages, and color, among other features of user interfaces. For example, they recommend the display of information only necessary to the user. Interim data, such as some messages, should be removed from the screen once no longer needed. Knowledge of the current state is also critical: the user should know the current data environment (what data are affected by the commands entered currently) and the current command environment (what commands are active).

A good command language should accommodate both the novice and expert by providing full menus, or typeaheads and abbreviations to menus. The *input language*, or set of commands, should be simple and consistent. For example, there should be consistency in the behavior of commands to quit a system. The *output language*, or the system's responses, communicates with the user at the key-press level (by echoing), lexical level (for example recognizing a parameter of type string), syntactic level (for example, recognizing a certain command), and semantic level (for example, processing or completing a command). A system's responses should be informative and tell the user what it is doing and why. Clear and consistently placed error messages and different help levels are also recommended.

Color, if possible to use, allows the inclusion of much more data in a single image without confusion — if it is presented in controlled dosages, rather than in a mosaic. For example colors, such as red, can be used for dangerous situations.

Dialogue independence in user interfaces [27] involves separation of the software that deals with the human-computer dialogue — the dialogue component — from the computational component, which deals with an application's internal computations. The computational component needs only a set of valid inputs, and is neither concerned with the method of their collection, nor with user interaction. The result is easier modification and maintenance of software, especially if the user interface is developed using iterative refinement techniques. The dialogue component can concentrate on human factors, be changed without affecting the computational component, and provide multiple interfaces for the same computational component.

Separation has its drawbacks: it is not always easy to achieve, can result in decreased efficiency as a result of increased inter-component communication, and may need separate data structures for the dialogue component. For providing semantic feedback, the dialogue component may sometimes have to be aware of an application's semantics.

2.4 The MUPE-2 Programming Environment

MUPE-2 is a fragment-based, integrated programming environment [43,46] for Modula-2. The environment focuses on the design, documentation, coding, testing, implementation, and maintenance phases of reasonably large modular programs. MUPE-2 falls into the category of programming environments, such as IPSEN and Smalltalk, that enable the integrated development of software in an incremental and modular fashion, by using various tools and techniques.

MUPE-2 supports programming-in-the-small and programming-in-the-large activities in a uniform manner [44], and integrates them in an enlarged scope, called *programming in the-all* [43]. *Programming-in-the-small* (PIS) deals with activities concerned with smaller granules of a program, as declarations, and program flow. *Programming-in-the-large* (PIL) deals with activities concerned with program units or modules and their interrelationships.

A primary feature of MUPE-2 is its view of a program as a composition of program fragments. A *fragtype* is a specific type associated with a fragment [45,42]. Fragtypes, shown in Table 2.1, form the building blocks of software: appropriately typed fragments can be used to assemble well-formed software. Fragments are retrieved from and saved to *FragLib*, an integrated library of fragments.

MUPE-2 enforces rules geared towards software development; for example, it allows

the building of isolated Declarations or Statements fragments. Because fragtypes form the basis of compatibility rules that drive the machinery to build a program, these rules define the legal set of operations on a typed fragment. For example, they can permit *fragtype transitions* that change the type of a fragment. Not only do such rules permit flexibility in software development, but they also provide a user with protection during software development since they ensure only a legal set of operations.

A recent detailed description of fragtypes has appeared in [42]. However, the rest of this section explains relevant parts of Table 2.1, which shows all the fragtypes currently supported by the computational component. In this table, a *phrase* is a natural language statement. Commands [44] that operate on fragments — in the context of a fragment-based editor, are explained in later chapters, as part of the user interface's view of them.

The structure of a fragment has significant implications on *xmupe2*. A window representing a fragment must correspond to the structure dictated by the fragtype. For example, a PIS fragment requires a simple window-representation, the multiple portions of a PIL node necessitate a multi-paned window to represent each portion, and the hierarchy of nodes in a PIL fragment requires a graphical display of this hierarchy.

Fragments of fragtype Abstract, Declarations, Exports, Expression, Header, Imports, and Statements are programming-in-the-small (PIS) fragments, whereas fragments of other fragtypes are programming-in-the-large (PIL) fragments. Nodes such as SuperModules, DefImpModules, ProgramModules, and Procedures are referred to as *PIL nodes*.

The DefImpModule encapsulates a Modula-2 DEFINITION and IMPLEMENTATION module pair. Its DefImpHeader contains the module's name, the DefImpDescription, any number of comments; the DefImport, the DEFINITION module's import lists; the DefExport, the DEFINITION module's export lists; the DefDecls, the DEFINITION module's declarations; the ImpImport, the IMPLEMENTATION module's import lists; the ImpDecls, the IMPLEMENTATION module's declarations; and the ImpStats, the IMPLEMENTATION module's statements.

In a SuperModule, the SuperModuleHeader contains the module's name; the SuperModuleDescription, any number of comments; the SuperModuleImports, the module's import lists; and the SuperModuleExports, the module's export lists.

A ProgramModule's ProgHeader contains the module's name and priority; the ProgDescription, any number of comments; the ProgImports, the module's import lists; the ProgDecls, the module's declarations; and the ProgStats, the module's statements.

Notation

$::=$	$=$	is composed of
$\{x\}$	$=$	0 or more occurrences of x
$ $	$=$	OR

Fragtypes

<i>Abstract</i>	$::=$	{Phrase}
<i>Declarations</i>	$::=$	{Declaration}
<i>Exports</i>	$::=$	{Export}
<i>Expression</i>	$::=$	Expression
<i>Header</i>	$::=$	DefImpHeader SuperModuleHeader ProgHeader ProcHeader
<i>Imports</i>	$. =$	{Import}
<i>Statements</i>	$::=$	{Statement}
<i>Modules</i>	$::=$	SuperModule DefImpModule ProgramModule
<i>Procedures</i>	$::=$	Procedure
<i>Program</i>	$::=$	ProgramModule {SuperModule DefImpModule}

Nodes

<i>DefImpModule</i>	$::=$	DefImpHeader DefImpDescription DefImport DefExport DefDecls ImpImport ImpDecls ImpStats
<i>SuperModule</i>	$::=$	SuperModuleHeader SuperModuleDescription SuperModuleImports SuperModuleExports
<i>ProgramModule</i>	$::=$	ProgHeader ProgDescription ProgImports ProgDecls ProgStats
<i>Procedure</i>	$. =$	ProcHeader ProcDescription ProcImports ProcDecls ProcStats

Table 2.1: MUPE-2 Fragtypes

In a Procedure, the ProcHeader contains the procedure's name, parameter list, and possibly, the function type; the ProcDescription, any number of comments; the ProcImports, the procedure's import lists; the ProcDecls, the procedure's declarations; and the ProcStats, the procedure's statements.

Chapter 3

The MUPE-2 User Interface: An Overview

Xmupe2 was designed and implemented with a number of requirements, some of which include purely MUPE-2-related ones, and others apply to the design of any user interface. The first section of this chapter deals with such requirements; the second presents an overview of *xmupe2*.

3.1 User-Interface Requirements

A list of requirements for *xmupe2* is as follows:

1. Multiple fragments

- a. **Requirement:** Create and operate on multiple fragments. Keep track of each window representing a fragment and associate it to the proper computational component structure.

Rationale: A program can be incrementally developed by synthesizing fragments of various fragtypes.

- b. **Requirement:** Manipulate window representations of fragments (resize, move, (de)iconize, hide/unhide, and scroll).

Rationale: Each fragment is independent of another. Screen space is limited.

2. Information associated with a fragment

Requirement: Maintain and be able to quickly retrieve the information associated with a fragment window (text or graphics displayed, editing menu, window coordinates of a location to highlight, and so on) Update the information after any fragment-window manipulation or change in the state of the underlying fragment.

Rationale: The information associated with a fragment is not static; operations on a fragment change its internal state (as maintained by the computational component), and consistency between this state and the window representation is essential.

3. Structure of fragment windows

- a. **Requirement:** Properly label a fragment window with the fragtype and a unique identifier.

Rationale: A fragment should be unique and easily identifiable.

- b. **Requirement:** Represent a PIS fragment as a simple window with text

Rationale: A PIS fragment contains just text of PIS program structures.

- c. **Requirement:** Represent a PIL fragment as a complex window, with a graphical hierarchy of internal PIL nodes, and a container of windows defining the textual representation of each node.

Rationale: A PIL fragment contains one or more PIL nodes arranged in a hierarchy.

- d. **Requirement:** Represent the textual representation of a PIL node as a multipaned window with each pane corresponding to the parts of a PIL node, and containing the proper textual program structures.

Rationale: Each textual representation of a PIL node has subdivisions, based on the type of node (see Section 2.4).

- e. **Requirement:** Restrict the textual representation of a PIL node to the window representing the parent PIL fragment.

Rationale: The PIL node is a child of a PIL fragment and is not an independent entity.

4. Location of internal structures from the screen

Requirement: Based on the window in which the mouse cursor is positioned, locate the corresponding computational component fragment or PIL node; this is straightforward for the simple window representing a PIS fragment but more difficult for the PIL nodes, because the screen presents a *flat* representation of the internal hierarchy of these nodes.

Rationale: The user should always be operating on the correct fragment or PIL node.

5. Display of text and graphics

- a. **Requirement:** Map to linear text (compatible with window-system routines to display text), the computational component's flat representation of program structures in windows representing PIS fragments or textual representations of PIL nodes.

Rationale: The computational component's representation of textual program structures is not suitable for quick display nor is it compatible with window system routines.

- b. **Requirement:** Support the creation, update, and display of a graphical hierarchy of objects representing PIL nodes. Manipulate graphics primitives such as lines and rectangles in order to draw simple diagrams. Associate a graphical object representing the PIL node with the corresponding computational component node.

Rationale: The computational component's representation of a node hierarchy is not suitable for display on the screen — it has no window-relative coordinates per node.

6. Display of structured cursors

- a. **Requirement:** For each fragment, map the internal coordinates of and display in reverse video, a structured cursor, which refers to a program structure instead of fine-grained constructs such as characters.

Rationale: The coordinates of a structured cursor refer to internal, hierarchical program structures, and cannot directly be highlighted on a *flat* screen.

- b. **Requirement:** Distinguish between structured cursors referring to textual program structures, and others referring to PIL nodes having graphical screen representations.

Rationale: The type of screen coordinates and display of each type of structured cursor is different — textual display expects rows and columns, but graphical display expects initial (x, y) -coordinates and the dimensions of a rectangle

7. Display of the mouse cursor

Requirement: Vary the shape of the mouse cursor based on the location of the mouse, and the type of system activity.

Rationale: The user should be given mouse-location and system status feedback

8. Cursor movements

Requirement: Support the screen movement of a structured cursor on program structures represented either textually or graphically. Detect the press of legal cursor-movement keys and call their respective computational-component cursor movement routines. After a successful internal cursor movement, unhighlight the old cursor and highlight the new one.

Rationale: The computational component implements movements of the structured cursor, based on the press of certain keys.

9. Editing menus

Requirement: For each fragment and textual representation of a PIL node maintain a window-system-specific hierarchical menu of *legal* editing commands. Associate the items of this menu to those of the corresponding menu maintained by the computational component. For every cursor movement or editing command properly update the former menu to maintain the correctness of this association. Use each menu item to fire the appropriate computational-component editing command.

Rationale: For each fragment and part of a PIL node, the computational component maintains a window-system-independent editing menu that fires the appropriate editing command and is updated with every editing command or cursor movement.

10. Editing commands

- a. **Requirement:** Fire and interact with the editing commands of the computational component's structured editor. Reflect editing command changes, whether they are textual, graphical, or changes of the fragtype of a fragment.

Requirement: For editing commands requiring mouse or keyboard manipulations, translate the results into a form understandable to the computational component

Rationale: Editing commands manipulate computational component structures not visible to the user.

- b. **Requirement:** Provide a pop-up window with textual editing capabilities.

Rationale: The TextEdit command supports the textual editing of PIS program structures.

- c. **Requirement:** Provide a readonly pop-up window with the capability to view program structures stored in a computational component buffer.

Rationale: The Inspect command requires that the user be able to view this buffer.

11. Modifiability

Requirement: Make the user interface easily modifiable by using a modular design and localizing the user interface – computational component communication to a few modules.

Rationale: The computational component does not remain static and changes in it should not affect the whole user interface.

12. Portability

Requirement: Encapsulate window-system-specific code to certain modules, and use a window system that is portable across different architectures.

Rationale: The user interface should be able to run on different architectures.

13. User interface guidelines

Requirement: Follow the user interface guidelines in Section 2.3.

Rationale: A user interface should make software easy to use and learn.

3.2 Xmupe2: The X Window System MUPE-2 User Interface

This section presents an overview of *xmupe2*. Section 3.2.1 discusses the windows in *xmupe2*, Section 3.2.2, issues in the design of *xmupe2*; and Section 3.2.3, issues in the implementation of *xmupe2*. Reference to requirements enumerated in Section 3.1 is made with the notation (*R*#), where # indicates the number of a requirement enumerated in that section. For example, (*R* 6) applies to all the requirements of the sixth item and (*R* 6 a) applies only to requirement *a* of the sixth item.

3.2.1 Window Structures

As a result of the multiple-fragment requirement, *xmupe2* can display multiple windows (*R* 1.a). Each is created dynamically, represents a fragment (*R* 1.b), and is independently movable, resizable, scrollable, and iconizable. The ability to resize or reduce a window into a shrunken representation, or icon, allows the user to reduce screen clutter, decreases time to find a certain window, and saves screen space -- a valuable resource in light of the multiple windows that *xmupe2* can create (*R* 13). Other manipulations of a window, such as its movement, also assist the user in managing screen space. Systems such as the Xerox Star [75] and Cedar programming environment [79], also use icons.

All independently-manipulable windows contain a top title-bar with the name of the window (*R* 13) and two squares for iconizing/moving/resizing the window. Both the title bar and window border are highlighted when the mouse moves within the window. This highlighting assists the user in indicating the current focus of interest (*R* 13). It is especially helpful if the mouse cursor is within a window which is partly obscured by another.

Table 3.1 enumerates the different types of windows that *xmupe2* can display. Names followed by an asterisk (*) indicate multiply-occurring windows. Literal names are shown in *italics*. The initials *C.M.* indicate the CreateFragment Menu; *E.M.* indicate an EditOps Menu which is used to fire editing commands (*R* 9). Section 6.2 further discusses menus in *xmupe2*.

Figure 3.1 presents an overview of most of the types of windows in *xmupe2*. The top left window, labeled *MUPE-2*, is the Main Window. Below it are two PIS Windows representing Statements and Declarations fragments, respectively. To the right of the Main Window are two PIL Windows, representing Modules and Program fragments, respectively. Note the

Name	Title	Contents	Creation
Main Window	<i>MUPE-2</i>	Main Button Window Main Messages Window	<i>xmupe2</i>
Main Button Window		Command & menu buttons	
Main Messages Window		Messages	
PIS Window*	<i>Fragment #</i> Fragtype	PIS structures	C M.
PIL Window*	<i>Fragment #</i> Fragtype	PIL Graphics Window PIL Container Window	C M
PIL Graphics Window		PIL-node hierarchy	
PIL Container Window		PIL-Node Text Windows	
PIL-Node Text Window*	PIL-node name	DefImpModule, SuperModule, ProgramModule, and/or Procedure	E.M
TextEdit Window	<i>TextEdit</i>	TextEdit Button Window TextEdit Editing Window	E.M.
TextEdit Button Window		Command buttons	
TextEdit Editing Window		Structured-cursor	
Inspect Window	<i>Inspect</i>	Inspect Button Window Inspect Viewing Window	E.M.
Inspect Button Window		Command buttons	
Inspect Viewing Window		Anonymous Buffer	

Table 3.1: *Xmupe2* Window Structures

two titled icons at the lower right corner of the figure. These represent a PIL Window (a Procedures fragment) and a PIS Window (a Statements fragment). The contents of each type of window are explained further on. The TextEdit and Inspect Windows which are associated with PIS Windows, are not shown in this figure. They are explained in Section 7.1.1.

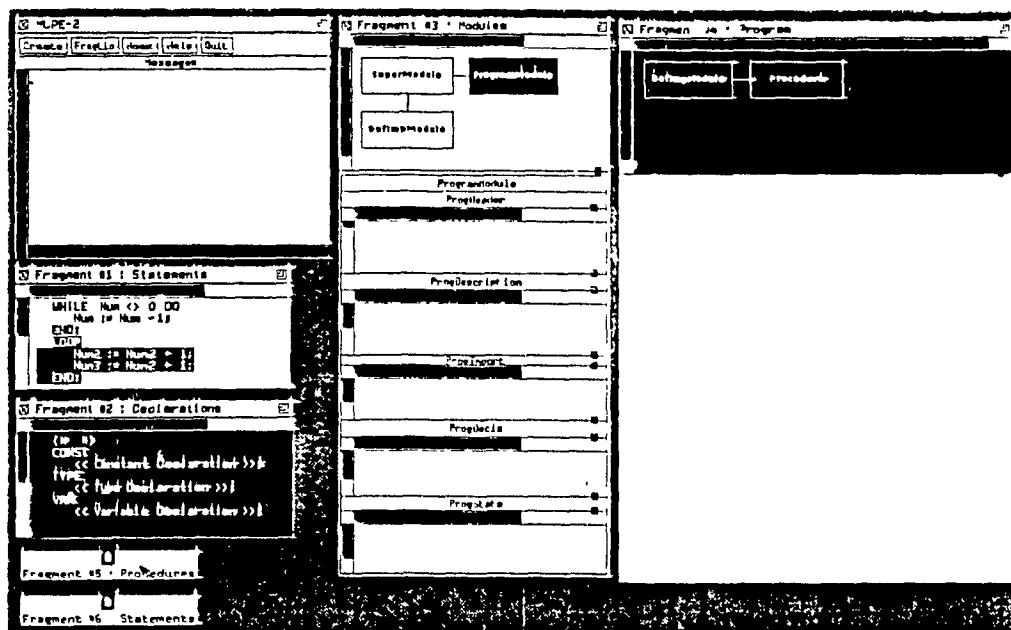


Figure 3.1: An Overview of *Xmupe2*'s Window Structures

The Main Window, shown in Figure 3.1 as the window labeled *MUPE-2*, is the initial window that appears when *xmupe2* is invoked. This window acts as *xmupe2*'s "control center" to: drive the creation of fragments, obtain general help, and quit the program. The Main Messages Window has vertical and horizontal scrollbars for viewing messages not fitting the window. Using the Main Messages Window as a central location for all of *xmupe2*'s user-directed messages, focuses the user's attention and avoids a plethora of confusing messages scattered among different windows (R 13). A new message first erases the currently displayed message.

Table 3.2 explains the function of each button in the Main Button Window. Menu buttons pop up a menu, when pressed with any mouse button; command buttons carry out

an action, when pressed with the left mouse button.

Title	Type	Purpose
Create	Menu Button	Pop-up CreateFragment Menu
FragLib	Command Button	Retrieve fragment from fragment library
Hook	Command Button	Save fragment to fragment library
Help	Menu Button	Pop-up Help Menu
Quit	Menu Button	Pop-up Quit Menu

Table 3.2: Main Window Buttons

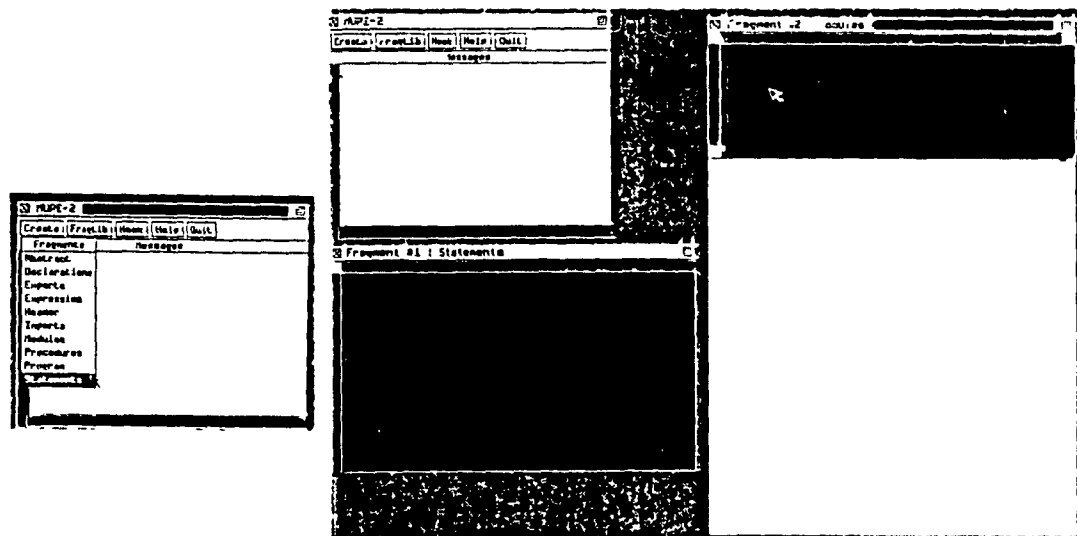
Windows representing fragments are of two types (R 3.b, R 3.c): those for PIS fragments — *PIS Windows*, and those for PIL fragments (Modules, Program, Procedures) — *PIL Windows*. Each of these types of fragment windows is created and pops up when the user selects the corresponding item in the CreateFragment Menu. A newly created window is labeled with the fragtype name and a unique number identifying the fragment (R 3.a). Such a labeling allows the user to distinguish among different fragments, especially those of the same fragtype. After the independent manipulation of each type of fragment-window on the screen, its associated contents are refreshed (R 2). Note that *xmupe2*'s screen layout supports MUPE-2 activities: PIS and PIL fragments are created independently and thus require independent windows. Each type of window reflects the data associated with the underlying fragment.

Movement of the mouse inside a window changes the shape of the mouse, based on the type of window. The shape of the mouse cursor also changes when *xmupe2* is performing an internal activity (R 7).

Figure 3.2 animates a series of actions to create fragments. Frame (a) shows the Main Window with a pull-down menu (the CreateFragment Menu), and the Statements item selected. The result, in Frame (b), displays the created Statements fragment (the PIS Window labeled with *Fragment #1 : Statements*). This frame also shows that the user intends to create a Modules fragment. The next frame (Frame (c)) contains the newly created Modules fragment (the PIL Window labeled with *Fragment #2 : Modules*).

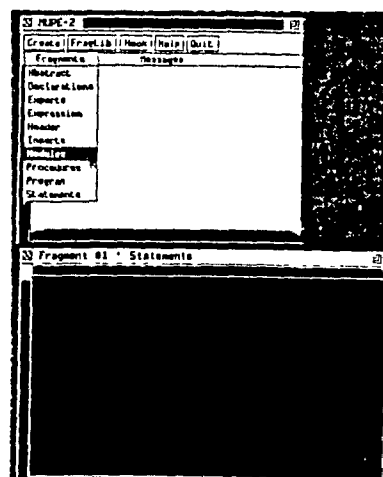
Windows for Programming-in-the-Small

A PIS Window is the container of linear text representing PIS program structures (R 5.a). Horizontal and vertical scrollbars allow the display of text that does not fit the window.



(a)

(b)



(b)

Figure 3.2: The Creation of Fragments

Each PIS Window displays, in *reverse video*, the external representation of a structured cursor referring to textual program structures (R 6). Certain keys are used for cursor movements on program structures (R 8). For each window, mouse buttons are used to pop-up contained WindowOps and EditOps Menus (R 9). Editing commands (R 10) that change the contents of a PIS Window are fired with the EditOps Menu; some, such as the Group and Drag commands, can also be completed with the mouse or keyboard.

Part of Figure 3.1 shows two PIS Windows representing Statements and Declarations fragments, respectively. The former is labeled as *Fragment #1 : Statements* and the latter, as *Fragment #2 : Declarations*. Both windows contain PIS program structures. The highlighted area in each represents a structured cursor: in the Statements PIS Window, it is on an internal program structure (the LOOP statement); however, in the Declarations PIS Window, it is on the entire fragment.

Windows for Programming-in-the-Large

A PIL Graphics Window contains a graphical representation of the PIL node-hierarchy (R 5.b) contained in the PIL fragment and represented by its parent PIL Window. Because the graphical hierarchy can become intricate, horizontal and vertical scrollbars in the PIL Graphics Window respectively permit horizontal and vertical scrolling of the contained diagram. A PIL Graphics Window also displays in reverse video, the external representation of a structured cursor referring to a PIL Node (R 6.a). Certain keys move this cursor from one PIL node to another (R 8). Editing commands (R 10) to insert or delete PIL nodes from a PIL Graphics Window use the EditOps Menu (R 9) for that window.

A PIL-Node Text Window is the textual representation of its respective PIL node displayed in a PIL Graphics Window (R 3.c). The former window is a multi-paned window in which the panes correspond to the divisions of a PIL node (R 3.d). Each pane, or sub window, contains scrollbars to control the display of text representing internal program structures, an EditOps Menu, and square grips to resize one pane at the expense of the other. A PIL Graphics Window and PIL Container Window can also be similarly resized. The capability to resize windows allows the user the flexibility to hide one subwindow while viewing the other (R 13). Because it is a child of the PIL fragment represented by a PIL Window, a PIL-Node Text Window is restricted in movement (R 3.e) to its parent PIL Container Window. When a structured cursor is on a certain PIL node in a PIL Graphics Window, the corresponding PIL-Node Text Window appears in the PIL Container window

of the current PIL Window.

Part of Figure 3.1 shows two PIL Windows representing Modules and Program fragments, respectively. The former is labeled as *Fragment #3 : Modules* and the latter, as *Fragment #4 : Program*. Both windows contain a hierarchy of PIL nodes in the upper PIL Graphics Window of each. The highlighted area of each PIL Graphics Window represents a structured cursor: in the Modules PIL Window, it is on an internal PIL node (the ProgramModule node); however, in the Program PIL Window, it is on the PIL Graphics Window. Note the display of the PIL-Node Text Window corresponding to the highlighted ProgramModule node of the Modules PIL Window. If the user changes the size of the Modules PIL Window, the size of this PIL-Node Text Window automatically changes to fit the parent PIL Container Window. The figure shows that the Modules PIL Window has been resized to exhibit this effect.

3.2.2 Design

The modular architecture of *xmupe2* permits easy modifiability of the program (R 11) by isolating one set of functions per module layer. One set of modules is solely responsible for the display of windows and menus on the screen; another, for handling user input and firing the proper routines in other modules; and the final, for directly interacting with the computational component. *Xmupe2*'s routines, which need to communicate with the computational component, make calls to the appropriate routines in its set of modules that is responsible for direct calls to the computational component. Changing the interface to a computational component routine necessitates change(s) just to the call(s) made in one set of modules, without affecting the other modules outside this set. Further details of *xmupe2*'s architecture are in Appendix A.

Xmupe2 attempts to be as ignorant as possible about the semantics of the computational component routines it calls. This ignorance simplifies *xmupe2* and also isolates it from changes in the computational component. For example, *xmupe2* simply maps computational component text and graphics into a form suitable for window display (R 5) and displays them without regard for the significance of their contents. Another example is *xmupe2*'s interaction with cursor movements: *xmupe2* updates the display of its representation of a cursor, based on the success of an internal cursor movement and on the cursor's coordinates retrieved and translated from the computational component. At no time is *xmupe2* cognizant about the reasons for the success or failure of a cursor movement in the

computational component, nor does it care about the rationale behind the current position of the structured cursor. Note that *xmupe2* only acts as a driver of the computational component. *Xmupe2* then reflects the internal changes, based on the information retrieved from the computational component.

User interaction with *xmupe2* is based on *event handler* and *callback* procedures that respond to input events or actions such as the press of a key, and call the appropriate routines in other *xmupe2* modules. There is no need to poll constantly for events and dispatch them to the correct procedure: the window system is responsible for this polling and other facets of event management. *Xmupe2* thus uses an *external* method of control, explained in Section 8.1.2. An advantage is to simplify the coding of the user interface and reduce its dialogue-control routines to a collection of event handlers and callbacks.

Shifting responsibility to the window system is not only used with event handlers and callbacks, but also used with the display of text or graphics. Once *xmupe2* sends it displayable text or graphics, the window system is responsible for their proper display and fitting within a window. The result is simpler code in *xmupe2*.

Xmupe2 does not take control of the screen, but coexists with other X Window System applications that the user may be running. The coexistence of independent programs is in accordance with the style of typical X Window-System applications, such as *xman* (a manual-page browser), and *xterm* (a terminal emulator). Some advantages of this style are that the user can interact with other programs and manipulate windows to manage screen space.

3.2.3 Implementation

Xmupe2 is implemented to run in the X Window System, a widely available and portable window system (R.12). The capability to manipulate windows (R.1.b) requires that a window manager be running under the X Window System, before invoking *xmupe2*. Using an available window manager saved considerable time in the implementation of *xmupe2*.

To locate fragments or PIL nodes displayed in windows (R.4), *xmupe2* maintains a list of nodes, called the Window List. Each window, which represents a PIS or PIL fragment or subwindow of a PIL-Node Text Window, has such a node. A pointer to the fragment's abstract syntax tree (AST) structure is associated with each node, and consequently, window descriptor. The usage of a node per subwindow of a PIL-Node Text Window effectively linearizes the hierarchical structure of the corresponding PIL nodes. This was done to

simplify the implementation and search of the Window List. Each node in the Window List also contains attributes associated with its respective window such as (R 2): the window descriptor, window name, text (for a PIS Window, and subwindow of a PIL-Node Text Window), coordinates of the structured cursor as displayed on the window, menus associated with each mouse button, and graphics tree (for a PIL Graphics Window)

The invocation of *xmupe2* calls routines to initialize variables and data structures, and creates the Main Window. Callbacks, associated with the menus and buttons of the Main Window, are procedures that are automatically called by the window system when the user selects a menu item or presses a button, respectively. *Xmupe2* then releases control to a X Window System main interaction loop that detects events and dispatches them to the appropriate event handlers or callbacks. Once these routines finish executing, control is returned to the main interaction loop.

For the creation of a fragment, *xmupe2* creates the window corresponding to this fragment and a Window List node associated with this window. It also defines the event handlers to trap the pressing of the mouse buttons and keys, the manipulation of the window, and the entrance of the mouse inside the window. It then calls the computational component routine to create a fragment internally, retrieves the coordinates of the structured cursor from the computational component, and displays the contents of the fragment in its window.

Manipulations of a window result in a call to event handlers that refresh the current contents of a window. These contents are stored in the Window List node for that window. Recall that *xmupe2* itself does not make this call — the main interaction loop in the X Window System is responsible for the call.

Whenever the user moves the mouse into a PIS Window, PIL Graphics Window, or a subwindow of a PIL-Node Text Window, *xmupe2* searches the Window List for the node whose window descriptor is equal to that in which the mouse is located. Once it finds the node, it informs the computational component of the associated current AST node. The drawback of this method is the search of a linked list, which can grow with an increased number of windows.

Whenever the user presses the right mouse-button while the mouse cursor is in a fragment window, *xmupe2* pops-up the associated EditOps Menu. If a menu item is selected and an editing command is to be executed, *xmupe2* calls the computational component editing routine. *Xmupe2* then retrieves the program structures to be displayed in the fragment, maps them to a suitable form, and shows them. It also retrieves the current structured

cursor and uses it to display the representation of this cursor. Finally, *xmupe2* updates the EditOps Menu based on the corresponding menu structure in the computational component. Control is then released to the main interaction loop.

A legal keystroke in a fragment window results in a call to the appropriate cursor-movement routine in the computational component. If this routine returns a successful result, *xmupe2* retrieves the structured cursor's new coordinates from the computational component, unhighlights the old cursor (whose coordinates are stored in a Window List node), paints the new cursor, and stores its coordinates.

Specific requirements not mentioned in this section are shown to have been satisfied in the following chapters that deal with unparsing, cursors, menus, and editing commands in the user interface. However, references to specific requirement numbers are not made.

Chapter 4

Unparsing and the User Interface

An *unparser* is a program that maps an internal abstract structure to text on a physical area [52]. The internal abstract structure is an object, such as an AST, which is hierarchical and intricate: its pictorial representation is not suitable for display on a screen. The text to be displayed is the flat representation of the AST's concrete syntax. Problems such as screen size and display formats must be addressed, in order to provide the user with a clear view of the edited program. Screen size problems can be alleviated with the use of scrollbars that give the user a movable viewport into the full text.

The unparser usually has some arbitrary rules for its display format. For example it could place only one statement per line and keywords in certain positions, and decide the spacing between entities. The user interface is not concerned with issues, such as formatting unparsed text, which are the computational component's responsibility.

Unparsers can be nonincremental or incremental. A *nonincremental unparser* regenerates the unparsed form of the whole program, for every change. This method is reasonable for small programs, but is less efficient for larger programs. The Cornell Program Synthesizer has such an unparser. An *incremental unparser* regenerates only the relevant parts of the AST for every change. This is more efficient, but more complex to implement than nonincremental unparsing. Rice University's Programming Environment for Fortran [4] has an incremental unparser.

Unparsing internal MUPE-2 structures is a critical operation that provides the user with a correct view of the edited internal structures. The AST in MUPE-2 is unparsed into two kinds of buffers: a *textual unparsed buffer* and a *graphical unparsed buffer*. The former is used for PIS fragments and the textual representations of PIL nodes, and the

latter, for PIL fragments. Although the computational component is responsible for maintaining these two kinds of buffers, a brief discussion is necessary to explain how *xmupe2* interacts with them. However, the algorithms to translate the AST to any of the unparsed buffers are of no concern to *xmupe2* and thus are not discussed. The structure of the AST and other computational component data structures of MUPE-2 are extensively treated in [66]. Because the computational component maintains an unparsed buffer for each fragment, *xmupe2* maintains its representation of the unparsed buffer corresponding to each computational component's unparsed buffer.

The rest of this chapter discusses relevant issues in textual and graphical unparsing, within one fragment and from *xmupe2*'s perspective.

4.1 Textual Unparsing

Performed for PIS fragments and the textual representations of PIL nodes, textual unparsing results in a textual display of internal program structures. For example, part of Figure 3.1 shows two PIS Windows with the results of textual unparsing. The Statements PIS Window exhibits statements formatted by the unparser of the computational component and displayed by *xmupe2*. The Declarations PIS Window shows similarly displayed templates of declarations.

In the computational component, the window-system-independent textual unparsed buffer consists of a doubly linked list of Line nodes, with each Line pointing to a doubly linked list of Line-part nodes, each of which contains items such as reserved words, expressions, and start/end columns of text. Each AST points to the first and last Line nodes of its textual unparsed buffer; conversely, each Line-part node points to its related AST node. As a result, the structured cursor (discussed in Section 5.1) moving on the AST, can be easily mapped to the corresponding unparsed buffer nodes, and consequently to row and column coordinates within a window. Some editing commands need a reverse mapping: screen coordinates are first associated to corresponding unparsed buffer nodes, and thus, to AST nodes. In such a mapping, the window containing a representation of the textual unparsed buffer, is viewed as a two-dimensional virtual array of rows and columns.¹

The computational component must maintain consistency between the AST and textual unparsed buffer. For example, for partial modifications of a line, the computational

¹“Virtual” because there is no real array data structure.

component unparses only the appropriate parts of the AST, and replaces the corresponding parts of the textual unparsed buffer.

The computational component's textual unparsed buffer is not directly suitable for window-display by X Window System routines. These expect a continuous string, whereas the unparsed text is spread over Line-part nodes scattered among Line nodes. To facilitate the quick initial display and efficient refresh of the textual unparsed buffer *xmupe2* maintains its own representation of the buffer of each PIS Window and subwindow of the PIL-Node Text Window. This representation is a dynamically-changing, continuous, one dimensional C array of characters — a string, to be called the C string

The *xmupe2* algorithm that translates, or *maps*, the contents of a textual unparsed buffer to this C string, traverses this buffer from a starting Line node to an ending Line node, both retrieved as parameters from the computational component. For each Line node the algorithm retrieves the text within each Line-part node and transfers it to the current position in the C string. The algorithm adds blanks at the start of each line and within a line, based on the starting and ending column numbers it obtains from each Line-part node

The traversal of the textual unparsed buffer and retrieval of text, and column information from each Line-part node, depends on results returned by calls to *xmupe2* Module 2 routines. Only these routines directly interact with the computational component code dealing with the textual unparsed buffer. Consequently, C code is insulated from the details of this buffer, the interface between *xmupe2* and the computational code is a clean one and it is easier to modify one part of code without affecting the other.

Xmupe2 calls its textual-unparsed-buffer mapping algorithm after the completion of each user-initiated editing command that is successful and alters this buffer. *Xmupe2* has no knowledge of the nature of the changes in, nor the contents of, this buffer. its only function is to display the unparsed buffer in a form suitable for a window. The initial costs of this algorithm are in the traversal of the appropriate parts of the textual unparsed buffer, and the transferral of text in each Line-part nodes to the C string. Nonetheless, this algorithm was chosen for its simplicity, not its efficiency. It easily allows *xmupe2* to maintain consistency between the textual unparsed buffer and the text displayed in a window. Although the algorithm is parameterized to unparsed a range of Line nodes, it is used to map the whole textual unparsed buffer. When there are no editing commands that change the textual unparsed buffer, the window displaying a mapping of this buffer, may still have to be refreshed as a result of manipulations such as a window's resizing or scrolling

Usage of the C string enables a quick refresh of the window, without having to re-map the textual unparsed buffer. This refresh, and the original drawing of just-mapped unparsed-buffer text, do not concern themselves with whether or not the text fits the containing window. *Xmupe2* simply informs the X Window System of the text it wishes to display, and the window system handles the actual display and fitting of the text onto a window — including when the window is scrolled. The shifting of responsibility from *xmupe2* to the window system has greatly simplified implementation of the display of unparsed text.

4.2 Graphical Unparsing

Performed for PIL fragments, graphical unparsing results in a graphical display of the PIL-node hierarchy specific to a PIL Graphics Window. For example, part of Figure 3.1 shows two PIL Windows with the results of graphical unparsing in their respective PIL Graphics Windows. Representing a PIL node, a rectangle contains the node's type. Lines connect either immediate siblings to each other or a parent to its first child. Any other children of this parent node are connected to their siblings, instead of the parent. This was done to simplify the algorithm that positions the rectangles. In the figure, the Modules PIL Window contains three PIL nodes: a SuperModule, ProgramModule, and a DefImpModule. The first two are siblings and the SuperModule is the parent of the DefImpModule. If the SuperModule had a second child, it would have been shown as a sibling of the DefImpModule. The figure also shows a Program PIL Window with DefImpModule and Procedure nodes as siblings.

The computational component's graphical unparsed buffer is a tree, each of whose nodes contains a pointer to the corresponding PIL AST node, the name (such as DefImpModule, SuperModule, and so on) to display for that node, and pointers to the left and right nodes in the tree. Unlike a textual unparsed buffer, it has no formatting details, because the coordinates of the graphical objects to be displayed depend the dimensions of these objects and other factors specific to the window system. However, the position of each node in the buffer indicates the hierarchical position of the node in a window.

For each hierarchy to be displayed, *xmupe2* recursively traverses the computational component's graphical unparsed buffer and maps it to a corresponding tree, the *xmupe2* graphical tree. The latter tree contains formatting details such as the (x, y) -coordinates of the top left corner of the rectangle representing each PIL node. Each node in this tree also contains a pointer to the corresponding PIL AST-node; this pointer is critical in associating

a PIL Graphics Window screen location with a PIL AST-node. This association is needed for graphical cursor movements, discussed in Chapter 5. To locate a PIL AST-node from a window location (x, y) relative to a PIL Graphics Window, *xmupe2* traverses the associated graphical tree until it finds a tree node whose corresponding rectangle coordinates contain (x, y) .² If no such node is found, the search fails; otherwise, *xmupe2* returns the graphical-tree node-field containing a pointer to the PIL AST node.

The mapping from a computational-component graphical unparsing buffer to the associated *xmupe2* graphical tree is done at the end of each editing command on PIL nodes within a PIL fragment. The disadvantage of the timing of the mapping is that it is nonincremental, however, it need not worry about how and when the computational component's graphical unparsed buffer is extracted from the AST. As with the mapping of the textual unparsing buffer, *xmupe2* code is also not directly cognizant of the names of the computational component's buffer fields. Refresh of a PIL Graphics Window, which is triggered by events such as scrolling or resizing of the window, uses the associated *xmupe2* graphical tree, instead of re-mapping the computational component's graphical unparsing buffer.

Once *xmupe2* builds a graphical tree, it draws it in the corresponding PIL Graphics Window. As with textual unparsing, *xmupe2* draws without worrying about which graphical structures are visible in the window: the X Window System clips them to the size of the window. The window system manages scrolling, but *xmupe2* must traverse a graphical tree in order to refresh it in a PIL Graphics Window.

²To simplify implementation, *xmupe2* assigns all rectangles the same width and height

Chapter 5

Cursors and the User Interface

An image on the display screen, the cursor is used to select information and provide location and program-activity feedback. Programming environments view the cursor in different ways. In Emily, a purely syntax-directed system, the cursor can only represent a program structure, such as a statement. In Magpie, whose editor follows a text model, the cursor represents just characters, instead of structures. The cursor in the Cornell Program Synthesizer, whose editor is a hybrid between a tree editor and text editor, can represent both characters or structures. In this editor, templates are inserted with commands and expressions and assignments are typed character by character. MUPE-2's structured editor supports both the structured and textual editing of program structures. Accordingly, its cursor can represent both structures and text.

This chapter first examines the types of cursors the user sees in *xmupe2*. It then discusses cursor movements as applied to *xmupe2*.

5.1 Cursors in Xmupe2

The cursor takes on different shapes in *xmupe2*: the *mouse cursor*, the *textual cursor*, and the *structured cursor*. The first two types of cursors concern the user interface; refer to text or a screen position, respectively; and are not affected by the computational component. The third type of cursor reflects a program structure, which is maintained by the computational component and is displayed and translated by *xmupe2*.

5.1.1 The Mouse and Textual Cursors

The mouse cursor refers to the shape of the image that corresponds to the absolute pixel screen-position given by the mouse. This type of cursor serves several purposes in *xmupe2*, the first of which is to provide the user with visual location feedback: moving the mouse changes its screen location. The mouse cursor can also be used to select information. For example, the mouse cursor is useful in assisting the user to select a menu item, by highlighting successive items as the mouse is moved.

Feedback describing a system's current activity is also important in a user interface. One method of such feedback is by changing the shape of the mouse cursor when a certain action is being performed. For example, immediately after the user selects an item from an EditOps Menu, *xmupe2* displays a watch cursor, which is restricted to the current window and indicates that the user needs to wait. After the internal *xmupe2* computations are completed, the mouse cursor reverts to its original shape and is freed from any restrictions of movement.

The mouse cursor also changes its shape, depending on the type of object — such as a window, menu, or button — in which it is positioned. For example, in the TextEdit Editing Window, the mouse cursor is pencil-shaped, indicating that the user can directly input text using the keyboard. Because the InspectBuffer Viewing Window is readonly, the mouse cursor is 'I'-shaped to indicate no action possible within the window. In a scrollbar, the mouse cursor assumes an arrow pointing to the appropriate direction. Table 5.1 shows the mouse cursor's shapes associated with different menus, windows, and buttons.

The second type of cursor, the textual cursor, refers to a character position within a TextEdit Editing Window and is shown as a caret-like structure. This cursor provides the point at which the user can edit text; it moves either by typing or by the mouse.

5.1.2 The Structured Cursor

MUPE-2's structured editor uses the structured cursor to manipulate program structures. Whereas the structured cursor in the computational component, denoted by the internal structured cursor, is of one type, *xmupe2* distinguishes between textual and graphical structured cursors, both of which are displayed in reverse video. The *internal structured cursor* points to an internal program-structure within a fragment (for example, to a simple expression or a program statement), an entire fragment (for example, to a Statements fragment),






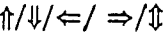

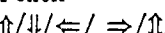
Object	Mouse Cursor	Purpose
PIS/PIL Window	Watch	Wait
Menu Button		Pull-down a menu
Command Button		Execute action
Main Messages Window	'I'	No action in window
PIS Window		Pop-up menu(s)
	Crosshair	Execute Drag/Group
PIL Graphics Window		Pop-up menu(s)
PIL-Node Text Window		Pop-up menu(s)
Resizing Grips		Resize window
Any label	'I'	No action in window
EditOps Menu		Select menu item
InspectBuffer Viewing Window	'I'	No action in window
TextEdit Editing Window	Pencil	Edit text
Scrollbars		Scroll

Table 5.1: The Mouse Cursor's Shapes

or PIL node (for example, to a Supermodule node). A *textual structured cursor* highlights textual program structures on which the internal structured cursor is located. A *graphical structured cursor* highlights a (graphical representation of a) PIL node in a PIL Graphics Window and causes the display of the corresponding PIL-Node Text Window in a PIL Container Window.

Each type of structured cursor focuses the user's attention to a certain area (of a PIS or PIL window) which corresponds to a program structure. The structure is either program text, in a PIS Window or subwindow of a PIL-Node Text Window, or a PIL node represented by a graphical node in a PIL Graphics Window. Another function of the structured cursor is to act as the operand for many of the MUPE-2 structured editor's commands. For example, when the cursor is on an assignment statement, and the user selects the Delete command from an EditOps Menu, the program structure to which the structured cursor refers, is deleted.

Figure 3.1 shows two textual structured cursors: one for the Statements PIS Window and another for the Declarations PIS Window. In the former window, the textual structured cursor highlights the LOOP statement; in the latter window, the cursor highlights the entire fragment, represented by the Declarations PIS Window.

In addition, Figure 3.1 shows two graphical structured cursors: one for the Modules PIL Window and another for the Program PIL Window. In the former window, the graphical

structured cursor highlights the ProgramModule node in the PIL Graphics Window. Note the appearance of the node's corresponding PIL-Node Text Window in the PIL Container Window. A PIL-Node Text Window appears *only* for the PIL node on which the graphical structured cursor is positioned. Consequently, the user can focus on a single PIL-Node Text Window instead of being confused with a cluttered PIL Container Window. MUPE 2 semantics also dictate that the user be able to edit only the node on which an internal structured cursor is located.

The other graphical structured cursor of Figure 3.1 is shown positioned on the entire Program fragment. Only the PIL Graphics Window is blackened because it is the one which represents the PIL fragment; the PIL Container Window is a container of windows and has no corresponding representation in the computational component.

5.1.3 Design and Implementation

Mouse cursors are easily implemented: the required types of cursors are created when *xmupe2* performs its initializations. The appropriate mouse cursor is attached to each newly created object such as a window, menu, or button. The X Window System is then responsible for displaying the correct mouse cursor shape when the mouse moves in an object, whether or not a shape was previously associated with that object. Textual cursors displayed in a textual editing window are created and managed by the window system. Note how *xmupe2* avoids managing the mouse and textual cursors by taking advantage of the window system's capabilities.

Xmupe2's design strategy for structured cursors is similar: let the computational component do the work. *Xmupe2* is only aware that after each editing command or cursor movement, it must: retrieve the internal structured cursor's coordinates from the computational component, associate them with the *xmupe2*-maintained text or graphics to display, and display them. At no time is *xmupe2* aware of the rationale for an internal structured cursor's coordinates.

In the implementation, *xmupe2* contains code that acts as a buffer between two data structures: the computational component's data structure that records the coordinates of the internal structured cursor and *xmupe2*'s corresponding data structure recording the window-specific coordinates of this cursor. This code maps an internal structured cursor's position to a window position. Recall that the movement of the mouse inside a window such as a PIS Window or PIL Window, triggers a search of *xmupe2*'s Window List; this search

results in an association between the current window and corresponding AST structure of the computational component. The internal structured cursor's coordinates are then guaranteed to apply to the correct window.

Each window containing a textual structured cursor has a data structure giving the cursor's window-relative coordinates, a boolean value to indicate if the cursor is on the entire window, and pointer to the contents of the window text to be highlighted. Figure 5.1 shows the algorithm to update a textual structured cursor after each editing command or cursor movement. In this figure, the *current window* is that in which the mouse is located, and the initials (CC) indicate a call to a computational component routine. The computational component translates the locations of the Line and Line-part nodes, which the internal structured cursor spans, to numerical coordinates. When *xmupe2* retrieves these coordinates, it receives numbers giving the cursor's first row, number of rows it spans, initial column in the first row, final column in the last row, and a boolean value indicating whether the internal structured cursor is on the entire fragment. *Xmupe2* then uses these coordinates to retrieve the corresponding text from the Window List node of the window in which the mouse cursor is located. This text is highlighted as the cursor; *highlighting* draws the text, from a start row and column to an end row and column, in reverse video. The structured cursor's window-coordinates are saved in the current Window List node because they are used in refreshing a cursor, after a window is manipulated. The advantage of this approach is to obviate the need for constantly requesting the internal structured cursor's coordinates from the computational component.

```

Retrieve numerical internal structured cursor's coordinates (CC)
Retrieve cursor text to which new coordinates point
If old cursor is not on whole fragment
    Unhighlight old cursor in current window
Else
    Paint window background white; and its text, black
If new cursor is not on whole fragment
    Highlight new cursor in current window
Else
    Paint window background black; and its text, white
Save new window-coordinates of cursor

```

Figure 5.1: Algorithm to Update the Textual Structured Cursor

The cursor coordinates of each graphical structured cursor include the window relative (x, y) -coordinates of the top left corner of the rectangle representing the PIL node. Because the width and height of all rectangles are equal, there is no need to store these values per node. Recall that each graphical PIL node is associated with a corresponding AST node, as a result, it is straightforward to map an internal structured cursor, which is on an AST node, to the correct graphical node. The process of highlighting/unhighlighting a graphical structured cursor is similar to the textual structured cursor's, except that a rectangle is highlighted. This highlighting draws a rectangle starting from the above (x, y) coordinates and spanning the rectangle's width and height. When the structured cursor is on the entire PIL fragment, *xmupe2* draws the background of the corresponding PIL Graphics Window in black, and the contained graph, in white.

5.2 Cursor Movements

Cursor movements aim to position the *structured* cursor on the desired structure, for editing, browsing, and so on. Moving the structured cursor through textual program structures presents its problems since program structures are represented as an AST that has been unparsed into a *flat* representation on the screen. The objectives in cursor movements on the textual representation of a program are to: make the cursor movements on the AST look natural on the screen, and minimize the number of movements to a destination.

Some programming environments support purely structured movements matching the program's syntactic structure, others support purely textual movements, and yet others combine both. Systems, such as MENTOR [13] or Gandalf [61], use highly structured, or hierarchical, cursor movements that seem unnatural in editing or browsing the flat screen representation of a program. The Cornell Program Synthesizer, which treats expressions textually, uses textual non-hierarchical cursor movements on expressions, and structured movements on program structures that it views structurally. In contrast and by treating programs textually, Magpie has cursor movements similar to a text editor's. Some systems such as Magpie and PECAN use both the keyboard and mouse for cursor movements. Others, such as IPSEN, provide cursor movements just by the mouse.

MUPE-2's cursor movements [47,48] are keyboard-based and centered on two types of structures: either the graphical PIL-node hierarchy in a PIL Graphics Window; or a textual display showing details of program structures in a PIS Window or PIL-Node Text Window

Cursor movements on the graphical hierarchy are highly structured movements from node to node. Because the textual display is a flat representation of an AST, textual cursor movements in MUPE-2 share the goals for such movements, which were mentioned at the beginning of this section.

Although the rest of this section mainly concentrates on cursor movements on the textual representation of a program, it gives a brief example of movements on graphical structures. Detailed theoretical principles of cursor movements are further discussed in [47]. The author participated in the strategy for cursor movements in the computational component, but did not implement these internal movements. The author's contribution is in the complete design and implementation of the user interface to internal cursor movements.

5.2.1 User's View

Structured tree-like cursor movements on program structures may seem unnatural or difficult, and purely flat textual-movements may not conform to a program's syntactic structure. MUPE-2 attempts to solve this dilemma by using *semi-structured cursor movements* [48], which move on fine-grained program constructs, such as expressions and CASE labels, and avoid their individual characters, or entire program structures.

Cursor movements in MUPE-2 need to consider program *partitions* (parts of a program on which the cursor can be positioned), and *streams* (paths or sequences of partitions that a cursor follows). The movements intend to simplify and assist the user in both the editing and browsing of a program. Partitioning the program according to the grammar results in highly structured movements; for example, placing the cursor on the reserved words `POINTER TO` in a record declaration is difficult. The user would be unable to easily delete these two words. Thus, *tool requirements* may dictate that some cursor positions make editing or browsing more efficient. For example, MUPE-2 deviates from the grammar and allows the cursor on the above two keywords, but is careful to preserve syntactic correctness.

MUPE-2 selects streams in such a way as to avoid unlikely candidates for editing (such as entire partitions), and to permit the user to move on partitions more likely to be edited or browsed, such as identifiers. Vertical streams include identifiers on the left hand side in declarations or statements, or labels in a CASE statement; horizontal streams include partitions in declarations or procedure declarations.

The user moves the cursor within a window by using one of two sets of keys, the first for

structured movements, and the second for semi-structured movements. Structured movements move the cursor either next or previous along outermost constructs within the parent construct, out onto the parent, or in on the first construct enclosed within the parent. Semi-structured movements move the cursor onto partitions along one of four directions. Table 5.2 shows *xmupe2*'s binding of cursor movements to specific keys. The notation *ctrl- $\langle key \rangle$* means: press the control key with the specified $\langle key \rangle$. For both PIS and PIL fragments, respective cursor-movement keys are the only ones that affect the state of the displayed fragment; other keys are ignored.

Category	Key	Cursor Movement
Structured movements	Ctrl-n	Next
	Ctrl-p	Previous
	Ctrl-o	Out
	Ctrl-i	In
Semi-structured movements	←	Left
	→	Right
	↑	Up
	↓	Down

Table 5.2: Cursor Movement Keys for Program Structures

Figure 5.2 shows how *xmupe2* displays cursor movements used in browsing a sequence of statements in a Statements PIS window. These movements were accomplished using the ↓ key. They show the cursor following a downward stream, without concern for a program's AST, and cutting through construct boundaries (such as the keywords) to allow positioning on structures most likely to be browsed. As a result of cutting through such boundaries, browsing program structures is faster.

Cursor movements in MUPE-2 also support editing. For example, the cursor is moved with the → key when initially on the left hand side of the following declaration:

```
TextArray = ARRAY [1..10] OF CHAR;
```

The second movement is from the left side to the subrange within the right side, instead of the entire right side. This is an example of a tool's requirements overriding the program's syntactic structure: the user is more likely to edit the innards of the right hand side, instead of the entire side. Editing of the entire right hand side is still possible, by using the control-o key.

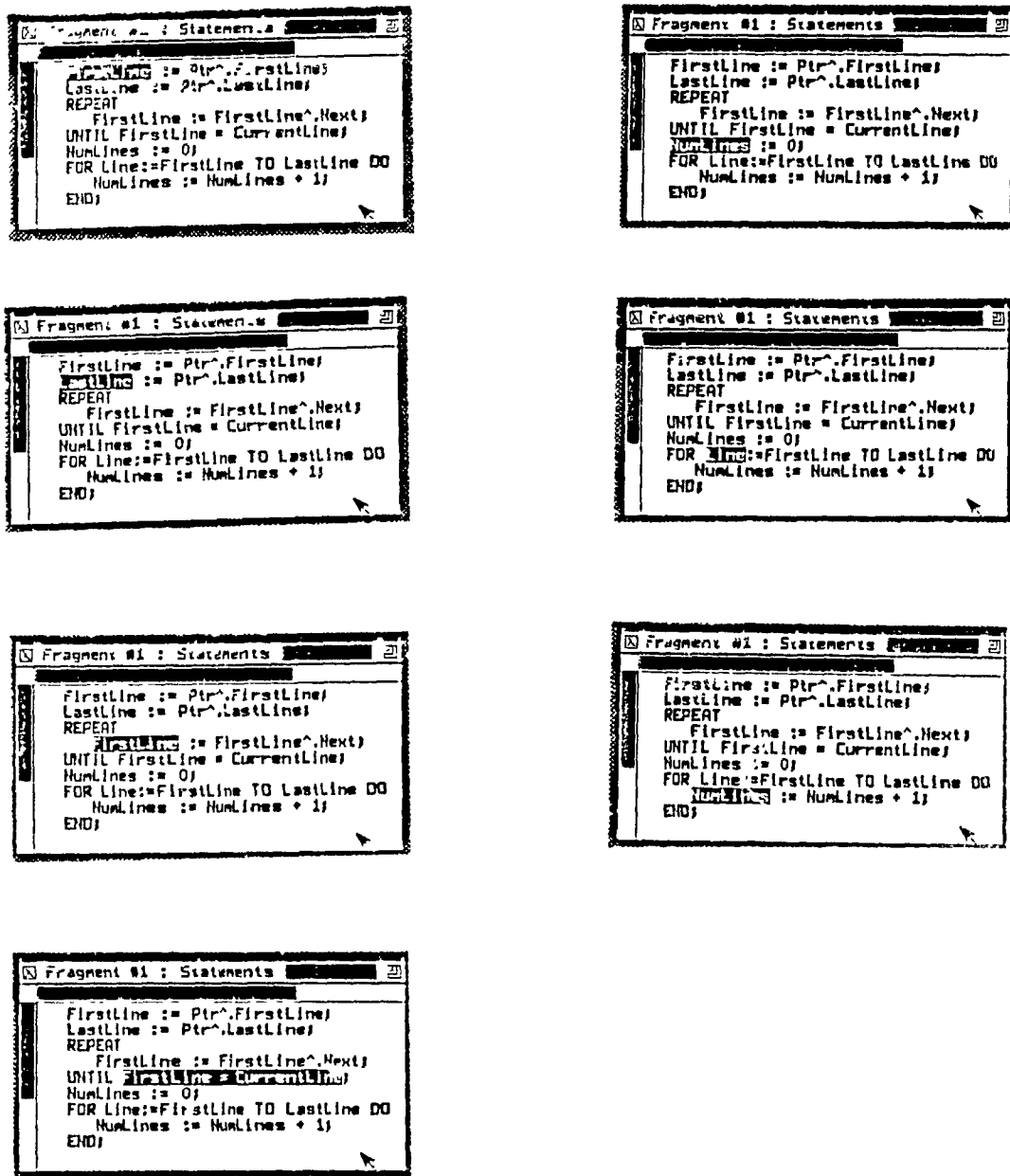


Figure 5.2: Cursor Movements in a PIS Window

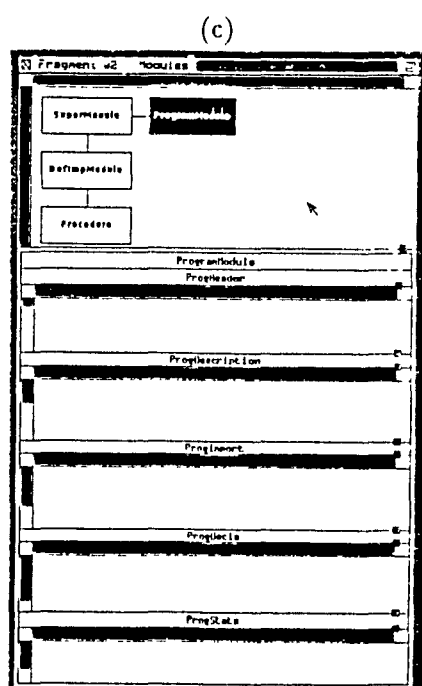
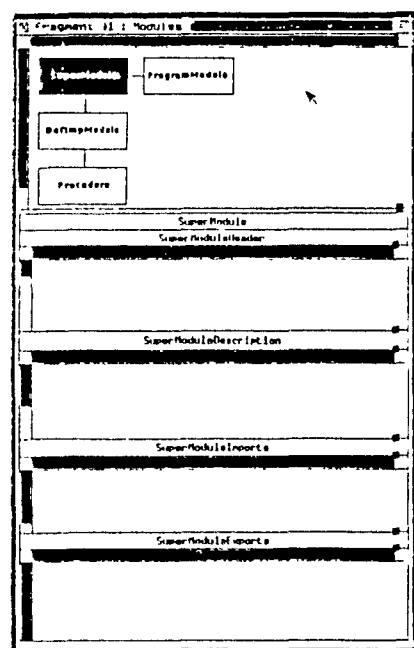
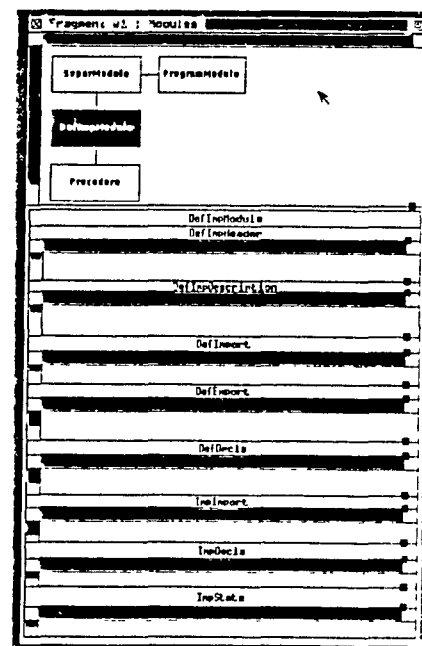
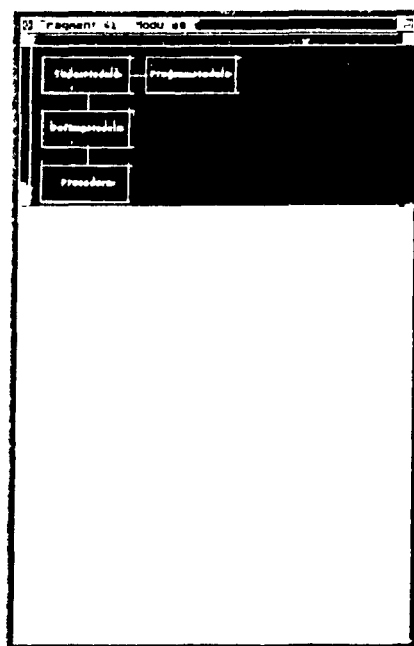
The highly structured graphical cursor movements can only use the structured movement keys of Table 5.2. Figure 5.3 animates such graphical cursor movements. Each frame shows, in the PIL Container Window, the PIL-Node Text Window corresponding to the PIL node of the current structured cursor. In Frame (a), the structured cursor is on the entire Modules PIL Window (that is, on the entire fragment). Two ctrl-i key sequences first move the cursor inside and position it on the SuperModule node (see Frame (b)) and then on its immediate child, the DefImpModule node (see Frame (c)). Frame (d) shows the cursor back on the SuperModule node, as a result of a ctrl-o. Moving the cursor from this node to the next adjacent node, the ProgramModule node, requires the ctrl-n keys (see Frame (e)). Similarly, to move the cursor from the ProgramModule node to the SuperModule node, the user must press the ctrl-p keys together.

5.2.2 Design and Implementation

Xmupe2 is responsible for mapping the results of successful (internal) window-independent cursor movements onto the window in which the mouse cursor is located. *Xmupe2*'s main design strategy for cursor movements, as previously mentioned, is not to be aware of the significance of a particular cursor movement. It is the responsibility of the computational component to determine the success or failure of a cursor movement.

The *xmupe2* algorithms to drive cursor movements in a window are not concerned with the visibility of the structured cursor as the result of a successful movement. The X Window System determines the visibility of a cursor: this is because a window acts as a scrollable viewport into the displayed text or graphics: the user can use the scrollbars to view a cursor not immediately visible. Resizing each window displaying a cursor can also usually achieve the same effect.

The computational component's internal cursor movements are independent of any window system, but *xmupe2*'s algorithm that drives cursor movements on text differs slightly from the one that drives graphics movements: the former paints a cursor containing just text, whereas the latter paints a graphical object. Otherwise, both algorithms are essentially similar and are outlined as one algorithm shown in Figure 5.4 (lines marked with a (CC) indicate calls to computational component routines). When the mouse cursor is in a PIS Window, PIL Graphics Window, or subwindow of a PIL-Node Text Window and the user depresses a key, the X Window System detects this event. It automatically calls *xmupe2*'s correct cursor-movement driver routine.



(b,d)

(e)

Figure 5.3: Cursor Movements in a PIL Window

```
success = false /*boolean*/
Retrieve key pressed
If key is for a legal cursor movement
    success = Perform cursor movement (CC)
If success /*successful cursor movement*/
    Retrieve internal structured cursor's coordinates (CC)
    Paint cursor
    Update editing menu for this fragment (CC)
    Update window's EditOps Menu
Else
    Inform user that cursor movement failed
```

Figure 5.4: Algorithm to Move the Structured Cursor

Chapter 6

Menus and the User Interface

The problem with command-line user interfaces is that the user has to remember the syntax and semantics of commands. Such interfaces are prone to errors in the entry of sometimes cryptic or complex commands and data. Menus, however, display the appropriate commands and options, encouraging a structured approach. They rely on recognition rather than recall, eliminate memorization of complex command sequences, require little or no prior knowledge or training, and hasten the learning of a system. In user interfaces of programming environments, menus provide cognitive assistance to different types of users: the novice user is unlikely to remember all system options and invocations, and the expert user may forget infrequently used commands and options. By ensuring properly structured and parameterized commands, menus act as a shield between the user and the system. But, an expert user may sometimes find menus a hindrance, and instead prefer menu accelerators or a command-language interface.

This chapter first discusses issues in menu design. It then examines the usage, design, and implementation of menus in *xmupe2*.

6.1 Menu Design Issues

Two menu design issues are menu organization and item presentation sequencing. Schneiderman [71] classifies menus according to semantic organization. Some types include single menus, linear sequences of menus, and tree structured menus.

Single menus contain multiple items and can extend to more than one screen. *Linear sequences* of menus consist of a series of interdependent menus which guide the user through

a series of choices. Presenting one decision at a time, these types of menus should allow the user to go back and view the results of previous choices. *Tree-structured* menus partition collections of items into groups — usually of logically similar and mutually exclusive items — of menus at different levels. Such menus must consider the depth (number of levels) of the menu tree versus its breadth (number of menu items per level). Decreasing the number of items per menu reduces the display time and screen clutter of the menu, but adding more items per menu reduces the number of menus and deepens a menu hierarchy. However, a deeper menu hierarchy increases search time or navigation of a menu.

Menus that appear with a mouse click and that are navigated by the mouse, seem to offer a compromise: they save screen space by appearing when needed and are quickly traversed. Yet, the user must remember which button to depress, keep it depressed, move the mouse to select a menu item, and then release the button. Expert users may become annoyed with this multi-step process and instead prefer accelerators or menu typeheads which use a mouse click or keystroke to execute the same command.

Items in a menu can be ordered in various methods, such as chronological, alphabetical, numerical, by similarity (functional grouping), or by frequency/importance of use [71]. Functional grouping is ideal for programming environment menus; for example, declaration templates can be grouped in one menu, statement templates in another, and so on.

6.2 Menus in Xmupe2

Menus used in structured or syntax-directed editors contain the commands applicable at any point. For example, Emily uses a fixed menu to display the legal constructs that can be inserted at any point of editing a program. The user uses a light pen to select a construct. Smalltalk extensively uses menus for execution of operations, PECAN uses menus for most commands, and IPSEN uses menu windows for the same purpose.

Xmupe2 uses: fixed single menus in the form of a list of menu and command buttons, single pull-down menus, and tree-structured pop-up menus for editing commands. All menus have clear and understandable titles and item names. The user utilizes the appropriate mouse buttons to display a non-fixed menu and traverses it by moving the mouse cursor over the items. Pop-up and pull-down menus appear only when needed, saving screen space. Releasing the mouse button on a highlighted menu item selects that item, for hierarchical menus, selection of an item occurs when the release is over a leaf menu item. All non-fixed

menus pop down when the depressed mouse-button is released; releasing the mouse button when the mouse cursor is outside the menu does not select any item in that menu. This behavior allows the user to gracefully exit from a menu.

Table 6.1 enumerates the different menus available in *xmupe2*. An asterisk (*) following a menu name indicates the menu can have multiple occurrences. The first three menus are pull-down menus, and the rest are pop-up menus. Italicized names are literal ones used in the menus.

Name	Parent	Contents
CreateFragment Menu	Main Button Window	Fragtype names
Help Menu	Main Button Window	Help items
Quit Menu	Main Button Window	<i>Cancel, Confirm</i>
WindowOps Menu*	PIS Window/ PIL Graphics Window	<i>Help</i>
EditOps Menu*	PIS Window/ PIL Graphics Window/ PIL-Node Text-Window subwindows	Editing commands

Table 6 1: *Xmupe2* Menus

6.2.1 Using the Menus

A pull-down menu is displayed by depressing a menu button in the Main Button Window. Pull-down menus are used for the creation of fragments, help, and quitting *xmupe2*. The CreateFragment Menu, as shown in Figure 3.2, allows the user to create a fragment of the appropriate fragtype. The Help Menu of the Main Button Window, Help Buttons in the TextEdit and Inspect Windows, and Help item in each WindowOps Menu, illustrate the principle of providing help at all levels. The Confirm item in the Quit Menu shows the principle of allowing the user to confirm dangerous commands. Quitting *xmupe2* destroys all associated windows and the structures maintained by the computational component.

A tree-structured EditOps Menu pops up with a mouse-button press and allows the user to execute an editing command, such as the insertion of PIS program structures or PIL nodes. There is an EditOps Menu for every PIS Window, PIL Graphics Window, and subwindow in a PIL-Node Text Window. To indicate that the menu is active, the mouse cursor has a temporary left-arrow shape while an EditOps Menu is visible. A menu item in an EditOps Menu has a submenu if a right-arrow is displayed at the rightmost side of

that item; menu items with no arrows are terminal or leaf items, which fire commands. Popping-up a submenu requires that the mouse cursor be moved to its arrow. The user is alerted to the existence of an EditOps Menu by a message that appears in the Main Messages Window, when a fragment is created.

A novel feature about an EditOps Menu is that its contents change after an editing command or cursor movement, as determined by the corresponding menu structure in the computational component. The rationale behind these context-sensitive changes is a property of the computational component: *xmupe2* only reflects these changes.

Figure 6.1 shows two different EditOps Menus resulting from cursor movements: each frame shows the complete set of options per menu. An interesting aspect of Frames (a) and (b), is how each EditOps Menu reflects the context of the structured cursor. In Frame (a), the structured cursor is on the WHILE statement. The EditOps Menu shows that the user can *Delete*, *Drag*, *Fold*, *Group*, or *Textually edit* the WHILE statement, *Inspect* the (unparsed) contents of the Anonymous Buffer (to which deleted ASTs are moved), or *Insert* other statement templates *around*, *after*, *before*, or *inside* the WHILE statement. In Frame (b), the cursor has been moved to the expression placeholder of the WHILE statement. Only the *Delete*, *Inspect*, and *TextEdit* options are shown in the EditOps Menu.

Note how each menu of Figure 6.1 has a title, separated from its menu items by a line, which clearly indicates the purpose of the menu. An EditOps Menu currently does not support the interruption of a selected editing command. Aborting a selected editing command is possible only if the command itself provides for this: for example, the *Drag* and *Group* commands can be aborted, *after* they have been selected from an EditOps Menu.

6.2.2 Design

Each pull-down menu contains items organized alphabetically to allow a menu item to be quickly located. One alternate ordering of a *CreateFragment* Menu's items would have been to group items by PIS or PIL fragment type, instead of alphabetically. No item in a pull-down menu is used more frequently than another, this is why an ordering of items by frequency of use was not considered.

Item presentation sequence in each EditOps Menu is not controlled by *xmupe2*, but is a function of the sequence in a corresponding computational component menu, which is organized by editing commands and their options. *Xmupe2*, however, does reflect this functional organization: related items are grouped together in a menu, and items per menu

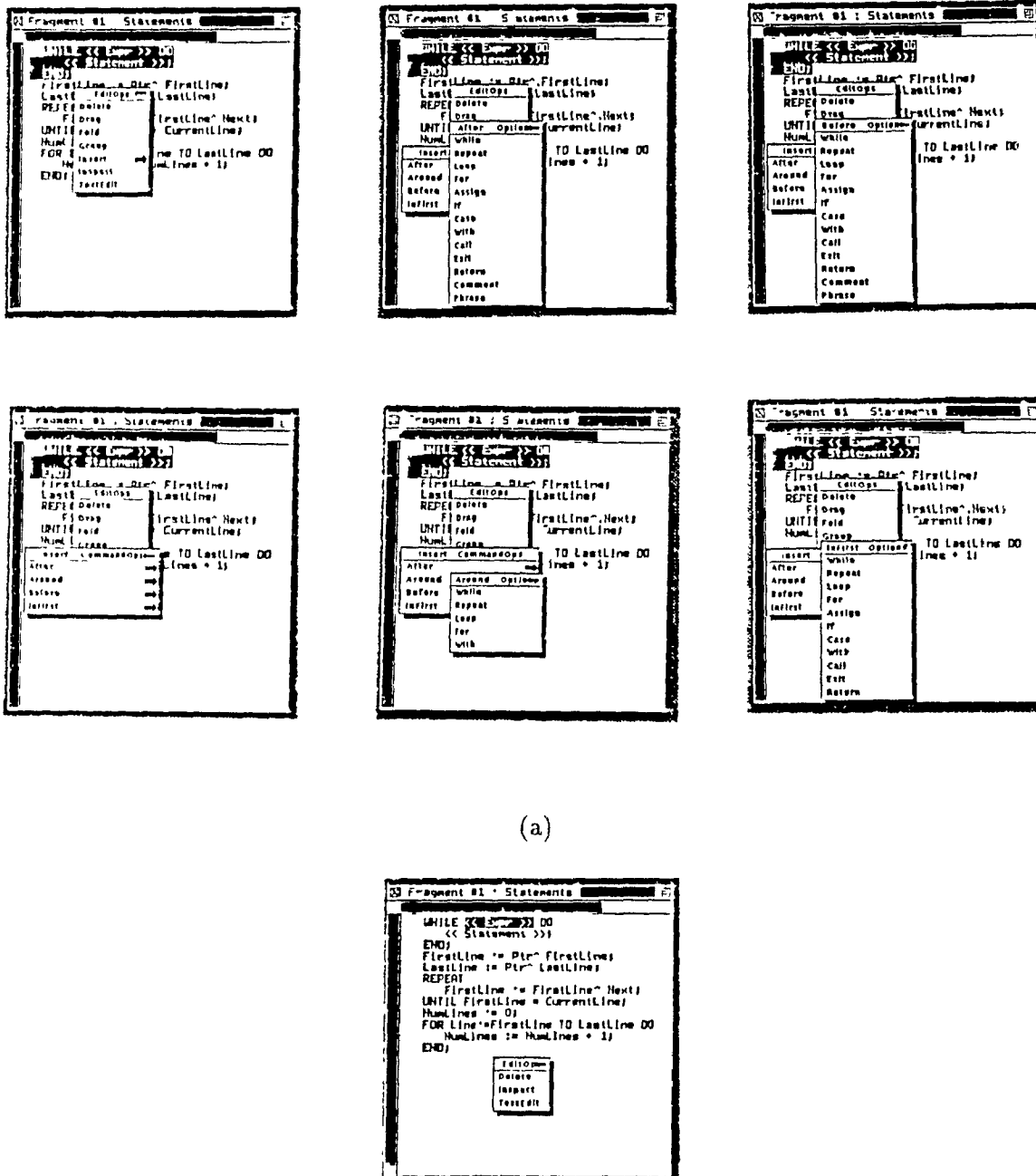


Figure 6.1: EditOps Menu

are organized alphabetically. For example, options to a first level item are grouped in second and third level submenus of this item. Items in an EditOps Menu are not organized in an alternate ordering, such as frequency of use, because this menu is constructed on the fly.

EditOps Menus were designed without typeahead or accelerator capability. The user has to navigate through the entire menu hierarchy in order to select a menu item. Nevertheless, these menus are easy to use, quickly popped up, displayed, navigated, and popped down. Because an EditOps Menu dynamically changes after cursor movements or editing commands, a pop-up menu, not visible to the user, is the best type of menu to use.

Table 6.2 contains the list of *all* possible items in an EditOps Menu, as determined by the computational component. The first level, in this tree structured menu, is that of editing commands. The second and third levels contain options to the corresponding first level item. An editing command is fired when the user selects that command, with all its options, from an EditOps Menu.

An EditOps Menu usually does not contain all the items of Table 6.2, but only those retrieved from the corresponding computational component menu. The computational component — not *xmupe2* — performs the role of context-sensitive menu filtering and prevents errors in program entry. An alternate method of display would have been to show all possible items, and gray out the illegal ones. A problem with that method is that the set of all editing-command options is large, and the resulting EditOps Menu could not possibly fit on the screen. The user would be frustrated with navigating through many illegal gray menu-items and could be confused.

Xmupe2 has simple, general routines to set an arbitrary hierarchical menu structure for display. This structure is independent of the organization of a computational component menu and thus insulates *xmupe2* from the computational component. Furthermore, in building an EditOps Menu from the corresponding computational component menu, *xmupe2* is not aware of the reasons underlying the legality of menu items. *Xmupe2*'s construction of a menu becomes a simple mechanical process.

6.2.3 Implementation

The *xmupe2* data structure to store each EditOps Menu is a tree of dynamically created nodes. Each node represents a menu item and contains: pointers to the next/previous nodes, a pointer to the submenu tree for that item, a back pointer to the parent menu — if it exists, the name of the item to display, a pointer to the function to be fired (if item is a

Command	Options	Options
Delete Drag Fold Group Insert	(After, Around, Before, InsideFirst, InsideLast)	(While, Loop, If, ElseIf, CaseElement, With, Exit, Comment, ConstBlock, VarBlock, ConstDecl, VarDecl, Export, RenameClause, Qualident, Enumeration, Pointer, Array, FixedField, CaseVariantElement, Expression, SuperModule, ProgramModule, Repeat, Assignment, IfElement, Case, ElseCase, Call, Return, Phrase, TypeBlock, ProcedureHeading, TypeDecl, Opaque, Import, Id, Subrange, ProcedureType, Record, Set, CaseVariant, CaseLabel, ActualParameter DefImpModule, Procedure)
Inspect TextEdit UnFold UnGroup		

Table 6.2: EditOps Menu Structure

leaf item), and a window-system window structure, among other fields. *Xmupe*'s window system-specific data structure is more space efficient than the corresponding window-system independent computational-component data structure storing the menu. The latter structure is a fixed three-dimensional array, structured similarly to Table 6.2. This structure also contains a field to indicate if a menu item is valid; the computational component sets this flag, based on its internal information. In contrast, *xmupe2*'s menu structure stores only the legal menu items. *Xmupe2* needs its own menu structure because the computational component menu structure is not suitable for display on the screen.

After a cursor movement or editing command, *xmupe2* calls computational component code to update its editing menu; and then calls *xmupe2* routines to translate this computational component menu to an EditOps Menu. The translation ensures that *xmupe2* provides the user with displayable menus that correctly reflect currently legal editing commands. It also ensures that *xmupe2* calls the correct computational component routine to fire the editing command: in fact, selecting a menu item calls an intermediate *xmupe2* routine which then calls the appropriate computational component routine. This buffering provides a modular structure, and prevents changes in one layer of code, from affecting others.

Chapter 7

Editing Commands and the User Interface

The user edits a fragment and its contents by using the MUPE-2 structured editor [8], which is implemented by the computational component. For programming-in-the-small, the editor supports both the structured and textual editing of program structures. Editing programming-in-the-large structures is purely structured.

The computational component currently implements a subset of MUPE-2's editing commands [12]. Most implemented commands are for the manipulation of program templates or PIL nodes. *Xmupe2*'s role is to fire editing commands from EditOps Menus and reflect the commands' results in the appropriate window. Thus, its contribution is to show the character of these commands and present a user-friendly interface to them.

This chapter first presents scenarios illustrating the effects of editing commands on the user interface. These scenarios serve to explain the editing commands, from a user's perspective, and to display the user's actual interaction with them. Secondly, the chapter discusses design and implementation issues facing the user interface of editing commands.

7.1 Editing Scenarios

Programming-in-the-small editing scenarios are presented in Section 7.1.1; Section 7.1.2 discusses scenarios for programming in the large. The scenarios in these sections do not show one aspect of *xmupe2*'s interaction with editing commands: while *xmupe2* performs the internal computations associated with an editing command, it changes the shape of the

mouse cursor to a watch, writes a suitable message in the Main Messages Window, and restricts the mouse cursor to the invoking window. The first two actions serve to remind the user that an internal program calculation is in progress and that waiting is necessary. The third action prevents the user from executing another command while the current one is being completed, and focuses the user's attention to the current command. Once the current command completes its execution, *xmupe2* releases the mouse cursor which returns to its original shape.

7.1.1 Programming-in-the-Small

Currently implemented commands for programming-in-the-small include: Group/UnGroup, Fold/UnFold, Delete, Inspect, Insert, TextEdit, and Drag. Figure 7.1 animates editing commands operating on a Statements fragment labeled as *Fragment #1*.

Group/UnGroup

Frame (a) shows the structured cursor on an entire FOR-loop. The user has depressed the right mouse button, chosen the Group item from the EditOps Menu, and is about to execute the command by releasing the button. The Group command combines adjacent structures starting from the initial position of the structured cursor to a target structure. The user selects a target structure by moving the mouse cursor to a window row on or within this structure and by clicking the left mouse button.

In Frame (b), the user is ready to perform the Group command. *Xmupe2* has printed a help message in the Main Messages Window and the mouse cursor has changed shape to a crosshair that is restricted within the Statements fragment. The help message indicates that *xmupe2* supports grouping by key or mouse; the former is less user friendly than the latter and is thus not discussed. Note also that the user is able to abort the Group command by depressing the middle or right mouse buttons. In such a case, a suitable message appears in the Main Messages Window.

Frame (b) also shows that the user has positioned the mouse cursor on the window row of the assignment statement *EndCol := LinePart^.StartCol*.¹ Pressing the left mouse button executes the Group command, the source and target operands are the first FOR-statement (the current structured cursor) and the above assignment statement, respectively.

¹Any window column is permitted

```

Line := FirstLine;
WHILE Line < NIL DO
  REPEAT
    FOR i:=1 TO EndCol - StartCol + 1 DO
      LinePart := LinePart + NextPart;
    END;
    StartCol := LinePart + EndCol;
    LinePart := LinePart + NextPart;
    EndCol := LinePart + StartCol;
    FOR i:=1 TO EndCol - StartCol + 1 DO
      NumSpaces := NumSpaces + 1;
      NumLeft := NumLeft + 1;
    END;
    WITH Menu(CadType) DO
      WITH (Menu(CadType)) DO
        WITH Options(OpType) DO
          NumLinks := NumLinks - RegularIndex;
        END;
      END;
    END;
  UNTIL LinePart = NIL;
  Line := Line + NextLine;
END;

```

(a)

```

Line := FirstLine;
WHILE Line < NIL DO
  REPEAT
    FOR i:=1 TO EndCol - StartCol + 1 DO
      LinePart := LinePart + Text(i) < SpecialChar;
    END;
    StartCol := LinePart + EndCol;
    LinePart := LinePart + NextPart;
    EndCol := LinePart + StartCol;
    FOR i:=1 TO EndCol - StartCol + 1 DO
      NumSpaces := NumSpaces + 1;
      NumLeft := NumLeft + 1;
    END;
    WITH Menu(CadType) DO
      WITH (Menu(CadType)) DO
        WITH Options(OpType) DO
          NumLinks := NumLinks - RegularIndex;
        END;
      END;
    END;
  UNTIL LinePart = NIL;
  Line := Line + NextLine;
END;

```

(c)

Messages

- to group by KEY
 - Press "N (NEXT) or "P (PREVIOUS) to move cursor to end-of-group structure
 - To execute group operation
 - Press RETURN key
 - Text from the initial cursor position to group end will be grouped
 - To abort group operation
 - Press ESC key
- to group by mouse
 - Move mouse cursor to end-of-group structure
 - To execute group operation
 - Press left mouse button
 - To abort group operation
 - Press any other mouse button

```

Line := FirstLine;
WHILE Line < NIL DO
  REPEAT
    FOR i:=1 TO EndCol - StartCol + 1 DO
      LinePart := LinePart + Text(i) < SpecialChar;
    END;
    StartCol := LinePart + EndCol;
    LinePart := LinePart + NextPart;
    EndCol := LinePart + StartCol;
    FOR i:=1 TO EndCol - StartCol + 1 DO
      NumSpaces := NumSpaces + 1;
      NumLeft := NumLeft + 1;
    END;
    WITH Menu(CadType) DO
      WITH (Menu(CadType)) DO
        WITH Options(OpType) DO
          NumLinks := NumLinks - RegularIndex;
        END;
      END;
    END;
  UNTIL LinePart = NIL;
  Line := Line + NextLine;
END;

```

(b)

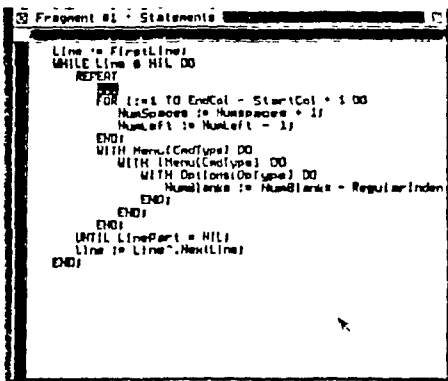
```

Line := FirstLine;
WHILE Line < NIL DO
  REPEAT
    FOR i:=1 TO EndCol - StartCol + 1 DO
      LinePart := LinePart + Text(i) < SpecialChar;
    END;
    StartCol := LinePart + EndCol;
    LinePart := LinePart + NextPart;
    EndCol := LinePart + StartCol;
    FOR i:=1 TO EndCol - StartCol + 1 DO
      NumSpaces := NumSpaces + 1;
      NumLeft := NumLeft + 1;
    END;
    WITH Menu(CadType) DO
      WITH (Menu(CadType)) DO
        WITH Options(OpType) DO
          NumLinks := NumLinks - RegularIndex;
        END;
      END;
    END;
  UNTIL LinePart = NIL;
  Line := Line + NextLine;
END;

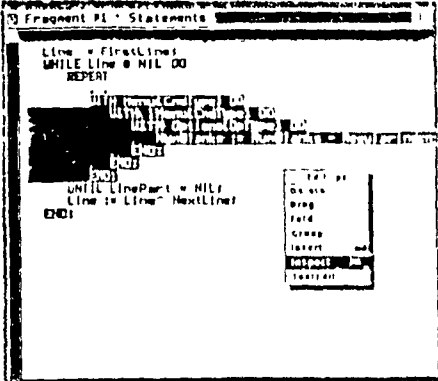
```

(d)

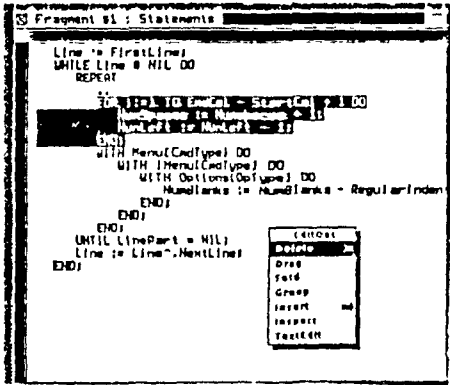
Figure 7.1: Programming-in-the-Small Editing Scenarios



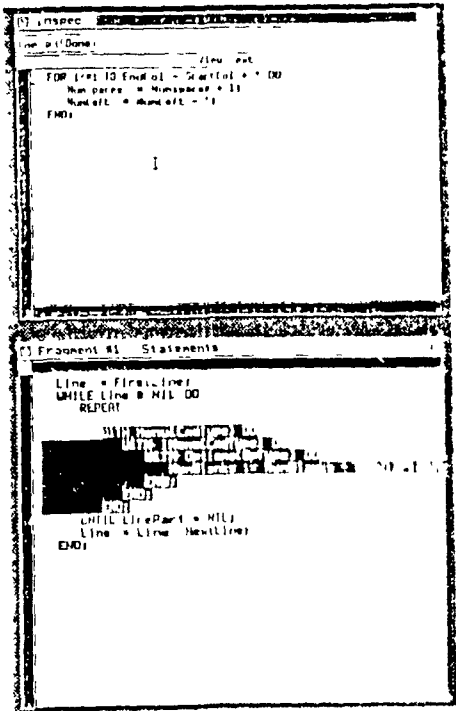
(e)



(g)

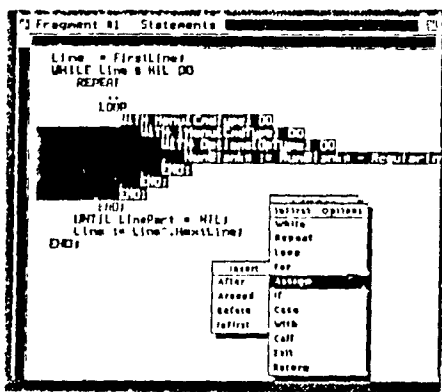


(f)

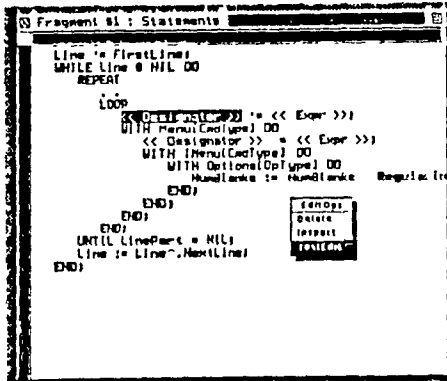


(h)

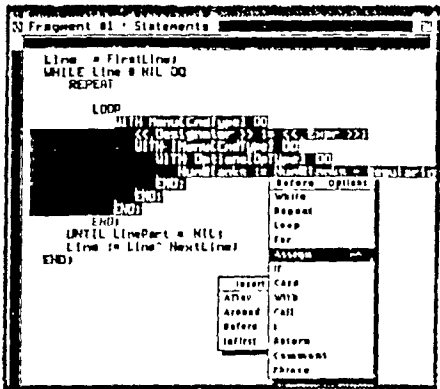
Figure 7.1: Programming-in-the-Small Editing Scenarios



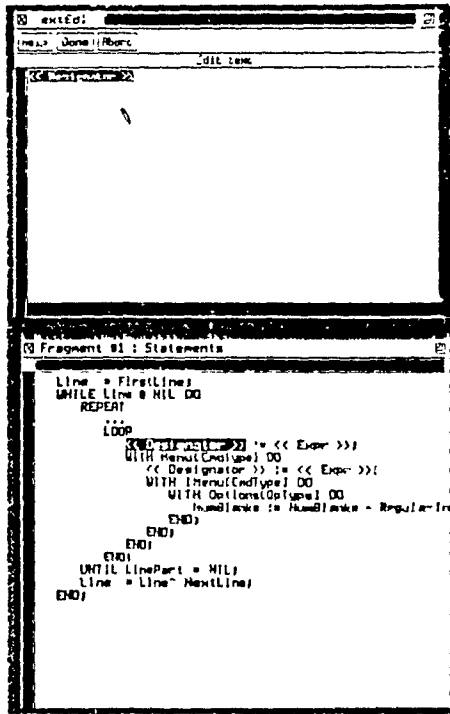
(i)



(k)



(j)



(l)

Figure 7.1: Programming-in-the-Small Editing Scenarios

```

Fragment 01 : Statements
Line := FirstLine;
WHILE Line <= NIL DO
  REPEAT
    ...
    LOOP
      LeftNum := << Expr >>
      WITH MenuCmdType1 DO
        << Designator >> := << Expr >>
        WITH MenuCmdType2 DO
          WITH OptionsType1 DO
            Humblanks := Humblanks + RegularLine
          END
        END
      END
    UNTIL LinePart = NIL;
    Line := Line*.NextLine;
  END
END

```

```

Fragment 01 : Statements
Line := FirstLine;
WHILE Line <= NIL DO
  REPEAT
    ...
    LOOP
      LeftNum := << Expr >>
      WITH MenuCmdType1 DO
        << Designator >> := << Expr >>
        WITH MenuCmdType2 DO
          WITH OptionsType1 DO
            Humblanks := Humblanks + RegularLine
          END
        END
      END
    UNTIL LinePart = NIL;
    Line := Line*.NextLine;
  END
END

```

(m)

(o)

Messages

- To abort drag operation:
Press ESC key or **MOUSE** mouse button or **RIGHT** mouse button
- To drag by **MOUSE**:
Move mouse to desired column and press **LEFT** mouse button
- To drag by **KEY**:
Press a succession of one or more **LEFT** or **RIGHT** arrow keys
Execute drag by pressing **RETURN** key

```

Fragment 01 : Statements
Line := FirstLine;
WHILE Line <= NIL DO
  REPEAT
    ...
    LOOP
      LeftNum := << Expr >>
      WITH MenuCmdType1 DO
        << Designator >> := << Expr >>
        WITH MenuCmdType2 DO
          WITH OptionsType1 DO
            Humblanks := Humblanks + RegularLine
          END
        END
      END
    UNTIL LinePart = NIL;
    Line := Line*.NextLine;
  END
END

```

(n)

Figure 7.1: Programming-in-the-Small Editing Scenarios

Frame (c) shows the structured cursor positioned on the resultant grouped structure, now surrounded by braces. The structured cursor has been moved to the second FOR-statement (see Frame (d)) in order to show the distinctive gray font of the grouped structure

To reverse the effects of the Group command, the user can choose the UnGroup command, taking Frames (a) and (c) in reverse shows the ungrouping effect of the latter command

Fold/UnFold

A powerful feature of a grouped structure (see Frame (c)) is that it can act as a single entity, which can be the operand of another command, such as a Fold. Frame (e) shows the effects of the Fold command on the structured cursor of the Frame (c).

The Fold command elides (holophrasts or selectively hides) program structures referenced by the structured cursor. In Frame (e), the ellipsis indicates folded program structures and represents more than one line. Elision is useful for condensing large programs and makes space a premium, unlike unparsing which stresses format.

The sequence of the Frames (c) and (e) shows *user controlled* elision, the user can elide (or unelide) program structures at will. COPE and the Cornell Program Synthesizer also feature this type of elision. The latter has CONDENSE (condenses the innermost expanded unit containing the cursor line) and EXPAND (expands the outermost condensed unit identified by the cursor line) commands. In contrast, PDE1L — a program development environment for PL/1 [53], has *automatic* elision. In PDE1L, the system identifies one or more foci of interest; text in the neighborhood of the foci is displayed, but text at some distance away is elided.

The UnFold command is the reverse of the Fold: the contents of a folded structure are unelided and appear as they were before folding. For example, taking Frames (c) and (e) in reverse shows the effects of the UnFold command.

Delete

Frame (f) shows the structured cursor on the FOR-loop and the Delete command about to be executed. The Delete command destroys the contents of the structured cursor. For example, to delete an entire fragment, the structured cursor first has to be moved on the whole fragment. In Frame (g), the FOR-loop has been deleted and the structured cursor has moved to the WITH-statement.

Most commands, such as the Delete, operate on the structured cursor, which can enclose structures within a fragment or an entire fragment. This is how commands can operate uniformly and offer the user a simple interface.

A method of quickly deleting multiple adjacent structures is to first group them, then delete the grouped structure, when referenced by the structured cursor. As an aside, consider Frames (a)-(d) of Figure 7.2. These successively show, the selection of the Group item from the EditOps Menu, the positioning of the crosshair on the correct window row, the successful execution of the Group command, and the deletion of the grouped structure.

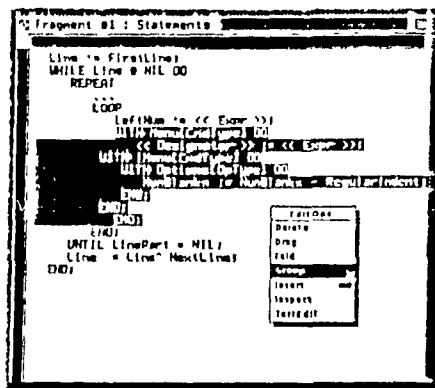
Inspect

The computational component moves a deleted PIS structure, except an entire fragment, to the *Anonymous Buffer*. For example, Frame (g) shows that the user has deleted the FOR-loop, but now wants to use the Inspect command to view the contents of this buffer. Choosing the Inspect menu-item causes the pop-up of the Inspect Window, shown in Frame (h) and labeled with *Inspect*; this window consists of an upper Inspect Button Window (containing Help and Done buttons) and a lower readonly and scrollable Inspect Viewing Window. The Help Button directs a help message to the Main Messages Window. The Inspect Viewing Window displays the current contents of the Anonymous Buffer, a FOR loop. Frame (h) also shows the shape of mouse cursor as an 'I', indicating the lack of editing actions in the Inspect Viewing Window. This cursor is also restricted within the Inspect Window until the user selects the Done button. Such a selection pops down the window and does not affect the contents of the Anonymous Buffer.

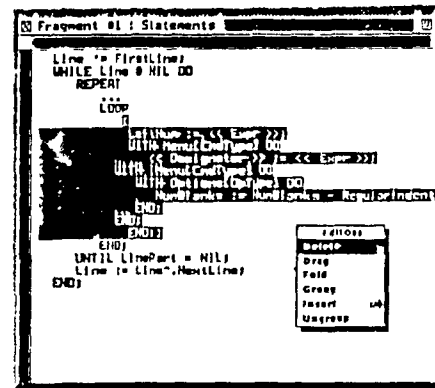
Insert

Frames (i)-(k) animate some variations of the Insert command. This command's four variations correspond to adjacent (Insert After and Insert Before), top-down (Insert Inside First/Last), and bottom-up insertions (Insert Around). All Insert commands center about the structured cursor: structures are inserted after, before, inside, or around, the structure referenced by the structured cursor. Inserted structures are either PIL nodes (see Section 7.1.2) or templates of PIS program structures.

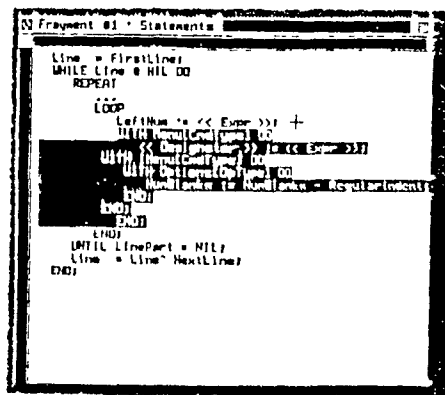
Insert InsideLast and Insert After are not shown here. The top-down Insert InsideLast is similar to the Insert InsideFirst, except that the former inserts a structure as the last



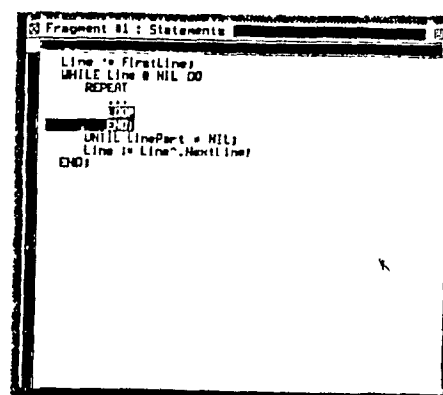
(a)



(c)



(b)



(d)

Figure 7.2: Grouping and Deleting Program Structures

one inside the container structure referenced by the structured cursor. The adjacent Insert After is analogous to the Insert Before, except that it inserts structures after the structured cursor.

Frame (i) shows the result of the insertion of a LOOP template around the WITH statement. The structured cursor remains on the WITH statement before and after the LOOP insertion. Frame (i) also shows the selection of an Insert InsideFirst command to fire the insertion of an assignment-statement template inside the WITH statement. As a result, the WITH statement then contains, as its first statement, an assignment template with Designator and Expr placeholders on the left and right sides of the assignment symbol, respectively (see Frame (j)). This frame also shows the selection of the Insert-Before (Assignment) item from the EditOps Menu. The result of this selection is to insert an assignment-statement template before the structured cursor, currently on the WITH statement. After this insertion, the structured cursor moves onto the newly inserted assignment-statement template. In Frame (k), the user has then moved the structured cursor inside this template and onto the Designator placeholder. This frame also shows the selection of the TextEdit item from the EditOps Menu.

TextEdit

The TextEdit command allows the user to textually edit the program structure referenced by the structured cursor. Such a command is useful either for the replacement of placeholders with identifiers or expressions or for circumventing the rigidity of pure structured-editing.

In Frame (k), the user intends to textually edit the Designator placeholder and replace it with a variable. After selecting the appropriate item of the EditOps Menu, the TextEdit Window appears (see Frame (l)). Labeled with a *TextEdit*, this window consists of an upper TextEdit Button Window (containing Help, Done, and Abort Buttons) and a lower TextEdit Editing Window (containing a mouse-based scrollable text-editor). The latter window displays the contents of the structured cursor, the Designator placeholder, and the textual cursor (the ^). The user can textually edit the TextEdit Editing Window's contents and press the Done Button to indicate that the result is to be passed to the incremental compiler. If there are no detected errors, the structured cursor is replaced with edited text. Unlike insertion by templates, which automatically maintains program integrity, an incremental compiler is needed to ensure the syntactic integrity of program text. Using the Abort Button abandons the textual editing session, and leaves the structured cursor's

contents — the Designator placeholder in the Statements fragment — untouched. Pressing the Help Button displays a help message in the Main Messages Window.

Note that during a textual editing session, the mouse cursor changes its shape to an inclined pencil and is restricted to the TextEdit Window, until the user presses the Done or Abort Buttons. The restriction of the mouse cursor's movements is intended to focus the user's attention on the current session and prevent multiple simultaneous textual editings of the same structured cursor.

Frame (m) shows that the Designator placeholder has been replaced with the identifier LeftNum. Between Frames (l) and (m), the user had typed the identifier LeftNum in the TextEdit Editing Window and had pressed the Done button.

Drag

In a PIS fragment, the computational-component's unparsing algorithm may either split a construct into separate lines for formatting purposes or display deep indentation. The user may not like the unparsing's formatting, and the Drag command's purpose is to accommodate individual formatting tastes. This command allows the user to horizontally move the structured cursor contents a number of spaces leftward or rightward.

In Frame (m), the nested WITH-statements are deeply indented, and the right side of the assignment to the identifier NumBlanks is not completely visible. There are three methods of dealing with this: scroll to the right, resize the Statements window, or use the Drag command. In Frame (n), the user has positioned the structured cursor on the WITH-IMenu statement and has chosen the Drag item from the EditOps Menu. A help message has been shown in the Main Messages Window and the mouse cursor has changed shape to a crosshair that is restricted within the Statements fragment. As with the Group command, a Drag can be performed by mouse or key. The latter, for similar reasons as with the Group, are not discussed. The Drag command, like the Group command, can be aborted.

Note the position of the crosshair in Frame (n): the user has moved it to the target column,² which is the first *O* of the LOOP identifier. A press of the left mouse button then displays the result of the Drag command: the WITH statement referenced by the structured cursor has been horizontally displaced to the left (see Frame (o)).

²Which window row is irrelevant because the drag applies to the structured cursor.

7.1.2 Programming-in-the-Large

The computational component currently supports just the Insert and Delete commands for PIL structures. The scenario in this section, shown in Figure 7.3, is based firstly on the structured insertion of PIL nodes and secondly, on their deletion. As with PIS editing, both the Insert and Delete commands operate on the structured cursor.

Insert

Frames (a) to (e) of Figure 7.3 animate a series of insertions in a Modules fragment, labeled as *Fragment #1*.

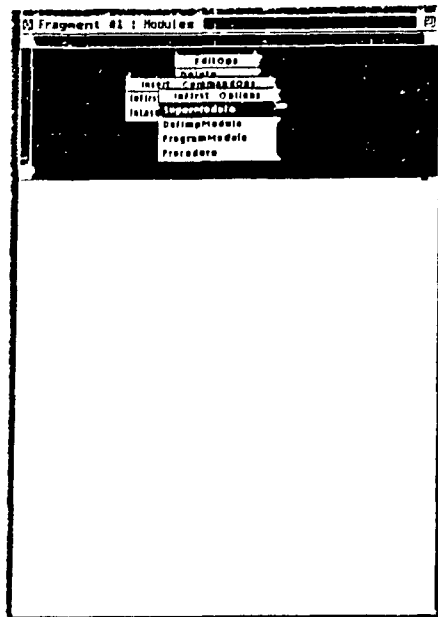
Frame (a) shows the structured cursor initially on the entire empty Modules fragment and the PIL Graphics Window is blackened to reflect this. By pressing the right mouse button, the EditOps Menu appears and the user can execute an Insert InsideFirst of a SuperModule. Release of the mouse button displays a SuperModule node in the PIL Graphics Window and the structured cursor remains on the Modules fragment.

In Frame (b), the user has first moved the structured cursor inside the PIL Graphics Window and onto the SuperModule. Note that this module's corresponding PIL Node Text Window appears in the PIL Container Window. The insertion of a ProgramModule after the SuperModule is shown in Frame (c). Because the ProgramModule was inserted adjacent to the SuperModule, both nodes are displayed as siblings at the same horizontal level. If the user had chosen an Insert Before command, the positions of the nodes in Frame (c) would have been reversed.

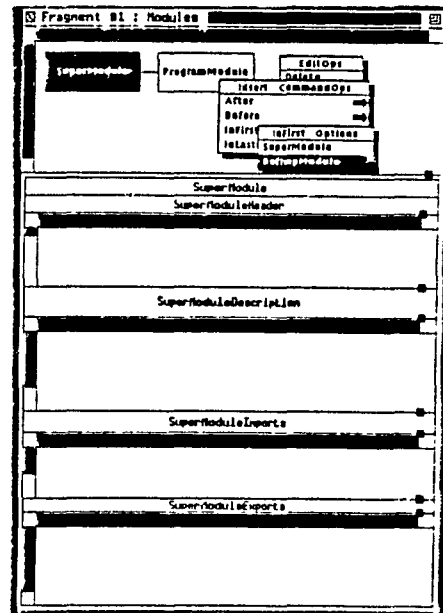
After choosing an Insert InsideFirst of a DefImpModule, the user sees a DefImpModule node displayed below the SuperModule node (see Frame (d)). The DefImpModule node is shown one vertical level below the SuperModule node because the former was inserted as a child of the latter. Frame (e) shows the result of a final insertion: an Insert InsideLast of a SuperModule node. The newly-inserted SuperModule node and the DefImpModule node are both siblings because they were inserted inside the SuperModule node of the structured cursor.

Delete

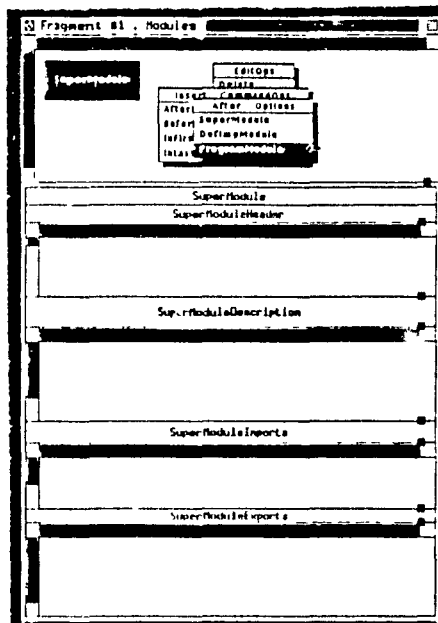
If certain nodes, or their children, are to be removed, the Delete command can be used. Frames (f) to (h) animate a sequence of deletions within the same Modules fragment.



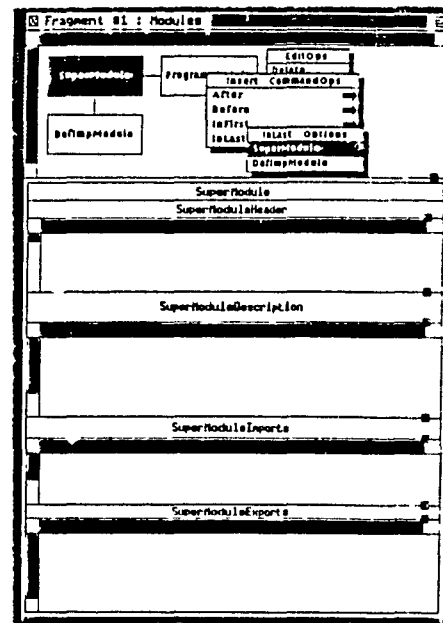
(a)



(c)

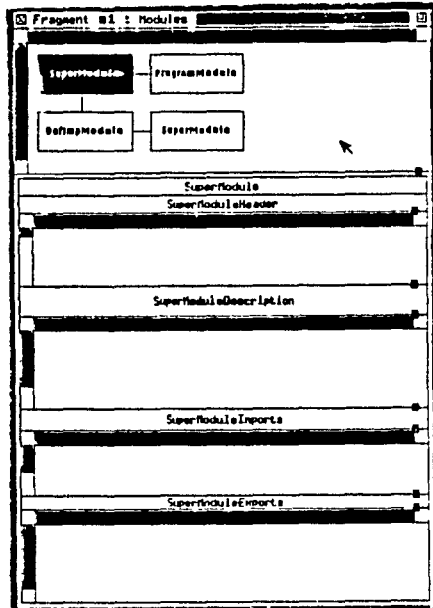


(b)

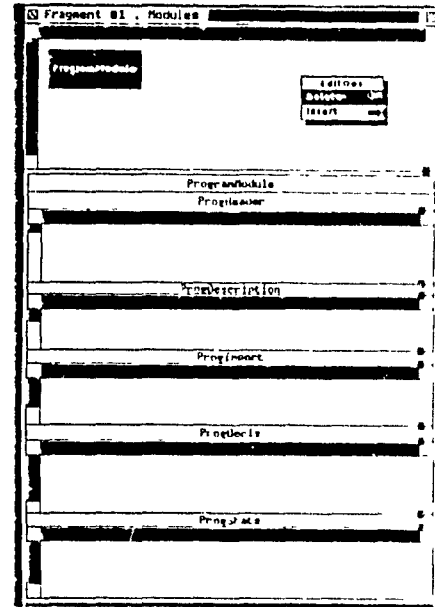


(d)

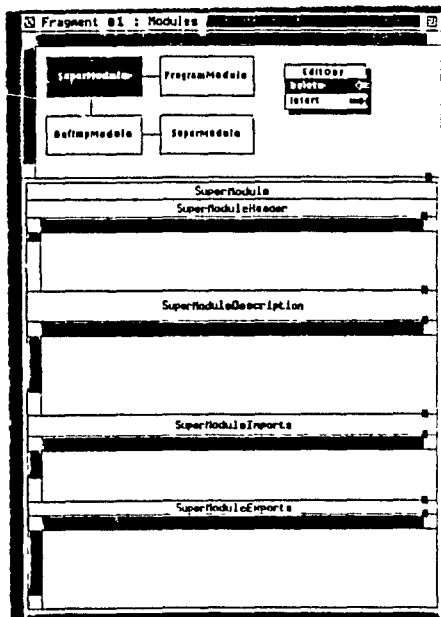
Figure 7.3: Programming-in-the-Large Editing Scenarios



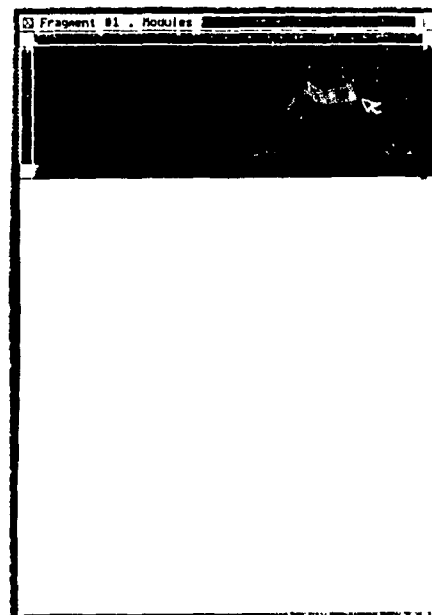
(e)



(g)



(f)



(h)

Figure 7.3: Programming-in-the-Large Editing Scenarios

Frame (f) shows that the SuperModule node (and all its children) are to be deleted. The result, in Frame (g), displays only the ProgramModule node — the sibling of the just-deleted SuperModule node. Deleting a hierarchy of nodes by removing their root makes Delete a powerful and flexible editing command. Because the structured cursor of Frame (g) is on the ProgramModule node, the node's textual representation is displayed in the PHL Container Window. Finally, Frame (h) shows the deletion of the ProgramModule node. The structured cursor is now on the entire Modules fragment, as represented by a blackened PHL Graphics Window. To delete the fragment, the user can choose the Delete item from the EditOps Menu.

7.2 Design and Implementation

Xmupe2's general strategy for the design and implementation of its interface to all editing commands is first discussed in Section 7.2.1. The subsequent sections investigate other interesting design and/or implementation issues specific to some commands. Not all commands are mentioned because the interface to them is similar.

7.2.1 General Strategy for All Commands

Xmupe2 fires an editing command by calling the appropriate computational component editing routine. Such a routine does not only execute the appropriate editing command, but also updates the coordinates of the internal structured cursor. *Xmupe2* then: (a) calls computational component code that updates the component's internal editing menu, (b) updates the current EditOps Menu based on this internal menu, (c) retrieves either the unparsed text or PHL graph, (d) obtains the internal structured cursor's coordinates, and (e) updates the proper window.

In general, *xmupe2*'s design strategy for interfacing with editing commands is to be unaware of the nature of the resulting changes. This is accomplished by relying on the information retrieved from the computational component after each editing command. For example, whether or not the cursor moves as a result of an editing command is determined solely by the computational component. *Xmupe2* only has to reflect the current position of the structured cursor. Another example is the interface to the Fold command: *xmupe2* is not aware that a folded structure exists — it just updates its representation of the textual unparsed buffer. This update shows the folded program structures replaced with an ellipsis.

Ensuring the integrity of options to a command, reflected in the correct items of an EditOps Menu, is critical. For example, textual editing of the structured cursor's contents is possible when the structured cursor is *inside* a PIS fragment, and located on an appropriate structure. The computational component is responsible for this checking and provide the correct options to its editing menu. *Xmupe2* is only aware that the computational component's editing menu must be updated per command. By updating an EditOps Menu based on this internal menu, *xmupe2* ensures the presentation of proper commands and their options at any point.

7.2.2 Delete

In the computational component, the Delete command is uniform because it removes just AST nodes. However, *xmupe2* must treat the deletion of an entire fragment (that is, when the structured cursor is on the entire fragment) differently from the deletion of its contents. For the deletion of program structures inside a PIS fragment, *xmupe2* simply updates its representation of the computational component's textual unparsed buffer and is not aware of the deleted structure. Deletion of a PIL node is similar, with respect to the graphical display in a PIL Graphics Window; however, *xmupe2* must also delete the PIL-Node Text Windows representing the PIL node and its children, and all their associated data structures. Deletion of an entire PIS fragment requires the destruction of the corresponding PIS Window, similar deletion of a PIL fragment requires the destruction of all PIL-Node Text Windows within the PIL fragment.

7.2.3 Drag

Part of *Xmupe2*'s implementation of the mouse-based Drag command is in an event handler routine that traps the press of a mouse button. Figure 7.4 outlines *xmupe2*'s algorithm to Drag by mouse. Part of it shows the C event handler that is called when the X Window System detects a mouse-button press. Once the mouse-cursor pixel-column position has been translated to a character-column number, the result becomes a horizontal offset (the drag count) that is passed as a parameter to the computational component routine executing the Drag command. The drag count is positive for a rightward move and negative for a leftward one. There is no need to access the computational component's unparsed buffer because the internal structured cursor already points to the current unparsed buffer line node(s); only the horizontal position of the underlying text must be changed. The algorithm shown

is an amalgamation of C and Modula-2 code: lines marked with (C) denote C code and those with a (CC) indicate calls to a computational component routine. *Horizontal_offset* is an integer variable indicating the net horizontal offset. *Button_x* is an integer variable indicating the pixel *y* coordinate in which the mouse button was pressed. *Xmupe2* maintains, for each window, a *cursor_x_position* value, which is the pixel *x* coordinate of the top left corner of the internal structured cursor. *Xmupe2* translates one of the internal structured cursor's coordinates to this pixel value after each cursor movement.

```

horizontal_offset = 0
status = fail (C)
Get the pressed mouse button (C)
If mouse button == Left mouse button (C)
    /*Get column from pixel */
    status = success
    horizontal_offset =
        (button_x - cursor_x_position) / width of text font displaying text

If status == success and horizontal_offset <> 0
    Drag with drag-count == horizontal_offset (CC)
    Refresh window text /*call to C code, made from Modula-2*/

```

Figure 7.4: Algorithm to Drag by Mouse

7.2.4 Group/UnGroup

Xmupe2 does not understand the significance of a grouped structure because this structure is a property of the computational component. Nonetheless, it knows that any text enclosed within braces must be displayed in a gray font. As a result, *xmupe2* does not have to remember which text is to be displayed in gray. The computational component inserts the braces around a grouped structure and is knowledgeable about their purpose.

When the Group command fails, an error message appears in the Main Messages Window, and the contents of the PIS Window are unaffected. Failure can be the result of a user's aborting of the command or an invalid end-of-group specification. The computational component does not allow the grouping of non-adjacent structures (those which are at different nesting levels) such as a WHILE statement and a statement inside an adjacent REPEAT statement. *Xmupe2* only acts as the messenger of the command's source and

target operands: it is not aware of the legality of grouping.

The implementation of grouping by mouse presents the problem of correctly mapping a window's flat coordinates to the corresponding structures of the computational component. Figure 7.5 shows elements of *xmupe2*'s algorithm to group by mouse. This algorithm is spread over two routines: the first is a C routine that handles retrieval of the mouse cursor position in a PIS Window; and the second, a Modula-2 routine that uses this position to interact with the computational component code. Lines marked with a (C) indicate C code; those marked with a (CC) are *xmupe2*'s calls, in Modula-2 code, to computational component routines, and unmarked lines represent *xmupe2* code written in Modula-2. *CurrentLine* is a pointer to the current textual unparsed-buffer Line node, *done*, a boolean variable, and *window_row*, an integer variable that translates a mouse cursor's *y*-pixel-coordinate to a text row. The algorithm returns a boolean result to its C caller: if the return value is true, the Group command succeeded in the computational component. In this case, *xmupe2* first updates its translations of the unparsed buffer and structured cursor, and then refresh the screen; otherwise, *xmupe2* makes no changes to its displayed text.

The algorithm in Figure 7.5 does not maintain any complex screen maps—it simply relies on taking advantage of the computational component's textual unparsed buffer. It also does not necessarily traverse every unparsed buffer Line node, but jumps from each set of Line nodes referenced by the temporary movement of the structured cursor. If every Line node contains a different construct that does not contain any other constructs (for example, an assignment statement), the algorithm degrades to a linear traversal of the internal unparsed buffer.

Finally, *xmupe2*'s interface to the UnGroup command is simple, because the computational component is responsible for removing the surrounding brackets. When *xmupe2* displays its translation of the unparsed text, ungrouped text is shown in a normal font because of the disappearance of the brackets.

7.2.5 Inspect/TextEdit

Both the Inspect and TextEdit Windows are created once: displayed when the Inspect and TextEdit commands are invoked, respectively, and hidden when the user chooses to end the textual editing or inspecting sessions. An alternative would have been to dynamically create and destroy each window. The first method is faster, but requires more space than the second. A further consideration is the user's interaction with both commands: a user

```

While there is no mouse-button press (C)
  If there is a middle or right mouse-button press (C)
    Return false (C)
  Else /*there is a left mouse-button press*/
    window_row = y pixel coordinate of mouse/height of text font (C)
    /*No need for column*/
  If window_row is within current internal structured cursor's coordinates
    Return true /*grouping current internal structured cursor*/
  If window_row > first row of current internal structured cursor
    Remember to use move-previous cursor movement routine
    CurrentLine = first unparsed-buffer Line-node to
                  which internal structured cursor points
  Else
    Remember to use move-next cursor movement routine
    CurrentLine = last unparsed-buffer Line-node to
                  which internal structured cursor points
done = false
While not done and CurrentLine is not NIL
  If internal cursor movement is successful (CC)
    Get Line node(s) to which internal structured cursor points (CC)
    If window_row is within current internal structured
      cursor's coordinates
      done = true
    Else if using move-previous
      Update CurrentLine to point to Line node just before
        first Line-node of internal structured cursor
    Else /*using move-next*/
      Update CurrentLine to point to the Line just after the
        last Line of internal structured cursor
    Else /*Internal cursor movement failed*/
      CurrentLine = NIL
Return done

```

Figure 7.5: Algorithm to Group by Mouse

often needs to textually-edit structures such as placeholders or to delete program structures and inspect them later.

Of course, *xmupe2* ensures the correspondence of the Inspect and TextEdit Window contents with the Anonymous Buffer and the current structured cursor, respectively. Representing the contents of the structured cursor and the Anonymous Buffer is essentially a process of textual unparsing (see Section 4.1).

The Anonymous Buffer, a structure of the computational component, is initially empty. The computational component fills this buffer with a structure just deleted from a PIS fragment. Successive deletions of PIS structures destroy the current Anonymous Buffer and replace it with the newly deleted structure. Commands such as Insert Buffer — not yet implemented by the computational component — could then insert the contents of the Anonymous Buffer into another location in the current fragment, or into another fragment. This usage of the Anonymous Buffer is envisioned to allow *interfragment operations*.

7.2.6 Insert

MUPE-2's computational component uniformly views the insertion of PIS and PIL structures: there is no barrier between PIL and PIS insertions, which are both manipulations of the AST representing a program. However, *xmupe2* must distinguish between the insertion of PIS structures and PIL nodes, because the former involves textual updating of window contents, whereas the latter requires a graphical updating.

In the context of a single fragment, *xmupe2* cares little about the difference between, say, an Insert InsideFirst or InsideLast. It only needs to fire the correct routine (already bound to the corresponding EditOps Menu item) and redisplay the unparsed text or graphics after the execution of an Insert command.

Chapter 8

User Interface Generation

User interface software can be difficult to create, modify, test, and maintain. A user interface must control or respond, as quickly as possible, to devices such as the keyboard and mouse. The programmer must effectively choose from an increasingly diverse array of graphics, window systems, interaction styles, input devices, among others. For example, *direct manipulation* [73] interfaces, in which the user can select and manipulate the visual representation of an object — typically by using a mouse, are easier to use, but more difficult to create than command-line based user interfaces. The user interface also communicates with two entities: the user, and the application's computational component, which contains internal code with which the user does not come in contact. A successful user interface must be both user friendly and correctly reflect the internal state of an application.

Amupeg2 is an example of the difficulties encountered when creating handcrafted user interfaces. Consisting of approximately 14,000 lines of code, its intricacy and coding partly involved knowledge of the computational component's structures, their manipulations, and reflections of these manipulations on the screen. Considerable effort was put into insulating one layer of *amupeg2* code from another and from details of the computational component. There were innumerable challenges to overcome, such as: the graphical representation of PII nodes, the translation and display of unparsec text, the creation and updating of the EditOps Menus, management of the contents of multiple windows, and the location of internal — often hierarchical — fragments as a result of the mouse's movement into a window.

Despite the effort of creating a user interface following the guidelines of good user interface design, there is no guarantee that the resultant interface is easy to use or learn.

For example, an interface that is easy to use for one user, may be a cause of frustration for another. This may lead to repeated modification of user interface software as it is developed. Such an *iterative design* methodology relies on testing prototypes with users and modifying the design of an interface based on user feedback.

Modification of complex user interface code is not always a simple task and may often require substantial effort. For example, although modifications and extensions to *xmupe*¹ are not difficult *per se*, they do require knowledge of both the X Window System and interaction between the user interface and the computational component code.

The time and effort expended in the design and implementation of *xmupe*² were the motivations for the author's work in the design of MUISL (The McGill User Interface Specification Language) — a programmer's experimental language for the specification of user interfaces, and the design and implementation of *mugen* (The MUISL-Based User Interface Generator) — a program to generate user interface code from MUISL specifications. This work attempts to answer the following questions: (a) Is it possible to easily design a simple experimental language for the specification of user interfaces? (b) Can a tool be developed to generate code from this specification? and (c) Can the combination of specification language and generator facilitate the development of sample user interfaces?

The rest of this chapter is organized as follows. Section 8.1 presents background and related work; Section 8.2 discusses MUISL's features, Section 8.3 describes *mugen*, and Section 8.4 evaluates both MUISL and *mugen*.

8.1 Background and Related Work

This section introduces user interface tools, discusses control methods in tools, and surveys selected approaches and tools to the specification and generation of user interfaces.

8.1.1 An Introduction to User Interface Tools

User interface tools [55,27] attempt to automate or ease the representation, design, implementation, execution, and modification of user interfaces. Some tools output executable user interface program code or declarative descriptions, such as database records, that are interpreted to produce a user interface. A *User Interface Management System* (UIMS) [65,27] is a tool or set of tools to help a programmer design, prototype, execute, and maintain a user interface. A UIMS usually integrates these activities under a single development

interface. The intent is to allow a user-interface specifier to concentrate on the higher level aspects of a user interface, instead of the low-level details. Typical functions of a UIMS include handling and validation of input, display of output, performing screen management, and refresh, and handling errors.

Taken from [27], Figure 8.1 shows the basic structure of a typical UIMS and its interaction with the proper developers. The application programmer implements the application's computational component. Interacting with this programmer is the dialogue developer, who uses dialogue development tools to implement the dialogue component. This component conducts an internal dialogue with the computational component and handles interaction techniques and event sequencing. To analyze and evaluate the user interface, the evaluator relies on data from stored guidelines, saved user interactions (for example, mouse button and key presses), and so on. Note that the dialogue developer and application programmer need not be different people.

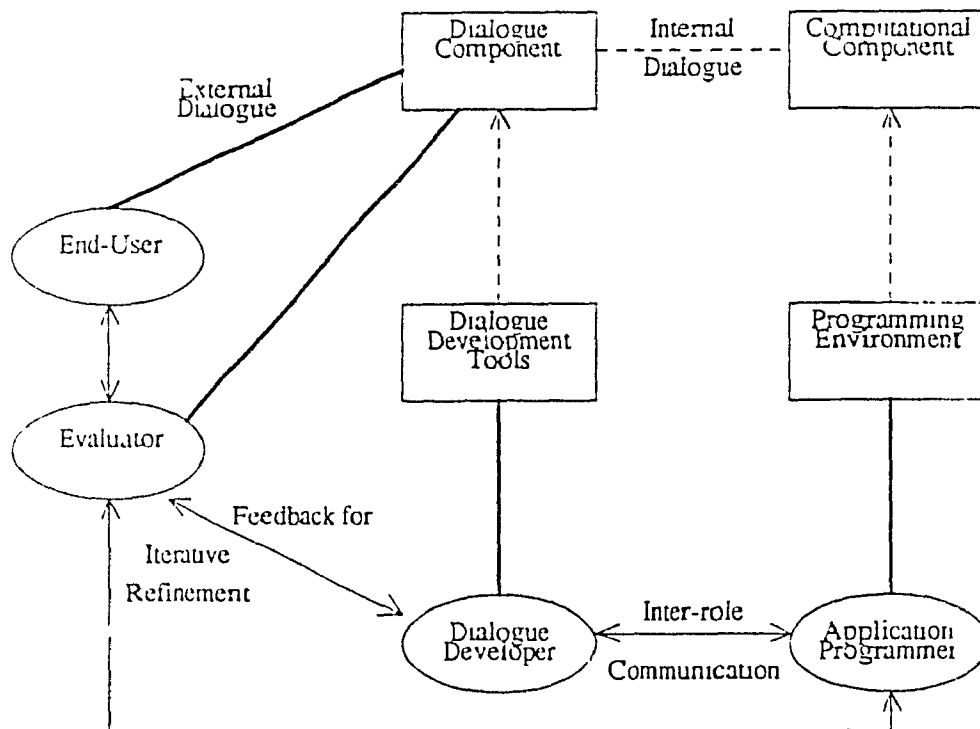


Figure 8.1: UIMS Architecture

Among the requirements for a user interface tool, such as a UIMS, are [27] its *functionality*, or what a tool can do, in terms of produced interfaces and techniques, and usable

input/output devices in these interfaces. The greater the functionality the better. Another requirement is a tool's ease of use or *usability*: tools more usable and easier to understand than others simplify a developer's task. The *completeness* requirement is a difficult one to satisfy. For example, the complete specification of a date field requires numerous details such as leap year information. The *extensibility* requirement allows a tool or the interface representations it produces, to be easily modified. Such a requirement compensates for the inability to attain absolute completeness in a tool. A tool lacking certain features and not providing for extensibility should be *escapable*: the tool should allow the developer to use regular programming when its features do not suffice. Given a set of user interface tools, an *integrated* and *uniform* interface to these tools is a desirable requirement. Another requirement is *locality of definition*, in which a local definition applies to most or all of the interface. For example, window title format can be represented once, yet apply to all instances of created windows. Changing the window title format in one localized definition automatically changes those of all created windows.

User interface tools can produce better, more extensible and maintainable interfaces [55]. These tools support rapid prototyping, allow separation of user interface code from an application's computational component, and can result in multiple interfaces per application. Some tools used for the specification and design of user interfaces, generate user interface code from these specifications. Specifications can be verified and validated and non-programmers can be involved in the design of a user interface the emphasis is not purely on implementation details.

User interface tools are not a panacea and exhibit problems. Firstly, the tools themselves may not be easy to build. Secondly, the advantage of consistency in interfaces may not appeal to those designers who seek control over all aspects of software requiring a unique feel and look. In addition, a user interface tool adds another layer of software that may slow down a program. Moreover, some tools are neither portable nor commercially available others are not easy to use and learn. Graphical tools are the easiest ones to use, especially for non-programmers, but many are mostly experimental. Finally, although many tools excel at providing fancy menus, windows, and other methods of interaction, their functionality may extend little beyond that.

8.1.2 Methods of Control

Communication between a user interface tool and an application may follow internal, external, or mixed methods of control [69,27].

In *internal* (application) control, the application calls dialogue, or user interface, functions for output and input. Compared to others, this model of control is efficient in execution, but difficult to modify a system's sequencing because of the application's control.

External (interface) control relies on user interface, not application, control. Sequencing is a function of user inputs, consequently, it is the user interface that calls application routines and handles this sequencing and scheduling. The application is viewed as a set of semantic routines, appropriately called by the interface. For these reasons, external control is prevalent among UIMSs.

In contrast to internal control, external control is better able to exploit prototyping. This is because external control gives a dialogue-oriented simulation of an application's behavior, and the programmer can easily provide application procedure stubs while the interface is being prototyped. But, lexical and syntactic handling are sometimes interleaved with global control code in the dialogue component, which invokes an application's routines. This interleaving makes the separation of dialogue and computational code more difficult.

User interface tools utilizing external control, communicate with the application using methods such as callback procedures, event handlers, and shared memory. *Callback procedures* are application procedures that are registered with a user interface tool. The interface calls the callback procedure at an appropriate time, such as the selection of a menu item.

Event handlers are procedures called when certain device-dependent events occur, such as the motion of a mouse cursor over an object. Event handlers specific to an application usually have to be registered with a user interface tool. The interface calls the appropriate application routines once a specific event occurs.

A problem with using event-based mechanisms is the need for often incorporating application semantics in a user interface. This is a problem because of the violation of the separation between interface and application components. A classic example of application semantics in a user interface is in direct manipulation interfaces such as the Macintosh's [34] or NeXT's [1]. In such interfaces, a file can be deleted by pressing the mouse button and dragging the file's icon to the trash can icon. The dragged file icon can pass over a folder icon, which is then highlighted to indicate a semantic relationship between folders and files. To highlight the folder icon, the interface must be aware of this relationship. Releasing the

mouse button over the folder icon deposits the file in the folder.

Shared memory between the user interface tool and the application is another method of communication. The application and interface poll the shared memory in order to check for changes. Alternately, changes can automatically notify the appropriate code. The shared memory method can be efficient, at the cost of extra memory.

Another method of control is *mixed control*, in which both the user interface tool and computational component can call another. For example, dialogue can be invoked from each of the computational or interface components. Although mixed control is flexible, dialogue independence is difficult to maintain. Mixed control adds more requirements to an interface and makes it more complex.

8.1.3 Approaches to Specification and Generation

Methods and tools for the specification and generation of user interfaces vary. The purpose of this section is to survey language-based, graphically-based, and other methods and tools [55,27].

Language-Based Methods

Language-based methods utilize specification languages for user interfaces. Variants of these encompass grammars in the form of BNF, state transition diagrams, object oriented languages, declarative languages, and event-based languages.

BNF

Backus-Naur Form (BNF) [58] is one of the language-based methods for the specification of dialogue and user interface syntax [67]. Terminals in the grammar are the input tokens representing a user's actions. Terminals combined by grammar productions, form nonterminals. Collections of productions in the grammar define the language the user employs in interacting with a computer. Attachment of program actions with each of the grammar's productions is also possible.

BNF has some disadvantages: it cannot explain the nature of human computer interactions, is restricted to context-free languages, has a fan-out problem because of their highly-structured nature, and is sometimes difficult to understand. However, BNF has been used as a syntactic notation to represent instances of human-computer dialogues.

The SYNGRAPH (SYNtax directed GRAPHics) [63] automatic generator of interactive systems uses an extended LL(1) grammar for interface representation. Input to SYNGRAPH are an extended BNF grammar describing the interface and the Pascal routines that are to be called to perform semantic actions. SYNGRAPH outputs, in Pascal, a screen manager, scanner, and recursive descent parser for the interface representation language.

The Abstract Interaction Tool (AIT) [82] is a language model for the specification of UIMSs. Based on the interaction hierarchy paradigm, AITs generate a dialogue language for a UIMS, by specifying dialogues and subdialogues. Consequently, AITs define a grammar for this language. Input expressions are the notational tool for this grammar and are hierarchically ordered to form a system of grammatical productions controlled by expressions.

State Transition Diagrams

Unlike BNF descriptions, which are usually created textually, state transition diagrams [64] — or finite state machines — can graphically represent human-computer dialogue. The typical transition diagram is a set of states connected by arcs, each of which is labeled with an input token, output to display, or application procedure to call. Movement from one state to another involves traversal of the connecting arcs.

Capable of representing an interactive system, the RAPID/USE system [84,83] executes transition diagrams describing this system. A diagram's nodes represent messages to be displayed; its arcs, transitions caused by events or user input; and its boxes, executable application-actions.

The State-Diagram Interpreter System [36] represents the time sequence of dialogue, based on lexical, syntactic, and semantic levels [19]. These levels view a user interface from a linguistic viewpoint: the lexical level is concerned with the structure of tokens; the syntactic level, with sequences of tokens and output form and content; and the semantic level, with output presentation techniques and input operations. The interpreter system has one diagram per level, and arcs can have recursive calls to other diagrams. Either an input or output token is exclusively associated with a transition. The system produces device-independent representations that provide control of the cursor and screen.

With each state representing a mode, transition diagrams are useful for multi-moded interfaces. Interfaces requiring detailed syntactic parsing can benefit from transition diagrams. Jacob [37] demonstrates that transition diagrams more directly show the time

sequence of human-computer dialogue and are a better structured representation of a user interface than BNF.

However, transition diagrams exhibit some problems. For example, descriptions can get large and arcs are needed for all input and commands, leading to increased complexity. A partial solution is to use subdiagrams. Furthermore, mode-free interfaces presenting the user with multiple choices at any time or interfaces requiring concurrent operations on various objects, can result in an intricate web of arcs out of a state. The Interaction Objects system [35], a combination of state-transition diagrams and event languages, attempts to deal with some of these problems and can support some form of direct-manipulation interfaces.

Although both BNF-based techniques and transition diagrams show the grammatical relationships in, say command sequences, they alone cannot show the means by which a command is gathered (for example, menus, windows, and so on) and entered (for example, by typing or mouse selection); and the semantic feedbacks resulting from users' actions (for example, feedback during the movement of an icon).

Object-Oriented Languages

Object-oriented languages allow the interface developer to define interfaces in terms of objects. These are entities which are classified into classes having attributes and default behavior embodied by methods (procedures for performing activities). Communication between objects causes all activity and inheritance of classes is typical of object-oriented systems. One advantage of object-oriented systems is that they facilitate building complex objects by combining simpler ones. Another is that the process of creating a user interface is often simpler, because of a tendency to view a user interface in terms of the characteristics and behavior of objects such as windows, and menus.

The George Washington University UIMS (GWUIMS) [74] is based on an object-oriented design paradigm. It incorporates the lexical, syntactic, and semantic levels of an interaction language, by embodying the boundaries between levels within object classes. Object classes consequently represent different levels of abstraction. In addition, GWUIMS supports inheritance, attributes, and methods.

Declarative Languages

Declarative languages concentrate on what should happen, rather than how it should happen. UIMSs based on declarative languages do not concern themselves with event

sequences, but instead concentrate on the information passed, such as global variables linking interface to application. Such UIMSs usually support only form-based interfaces, in which the user fills fields with information. The types of supported interactions are usually limited to preprogrammed fixed ones, usually with no support for graphical manipulation of objects — except for graphical areas used for application output.

The COUSIN (COoperative User Interface) system [30,32,31] provides a form-based interface definition in an interpreted language. Each definition consists of a form declaration with attributed field definitions. User-application communication is accomplished with abstractions called *slots* — each slot represents a value of information. An example is a slot per parameter of an application: before executing a command, the user specifies its parameters by filling the appropriate fields in the form, each of which corresponds to a slot in the interface definition.

Event-Based Languages

Event-based systems contain event handlers, each defined by a procedure or module, that are triggered on the receipt of the event(s) to which they have been attached. Each input token is considered to be an event, and the event handler that traps it can call the appropriate application routine(s), perform some computations, call other event handlers, or cause other relevant changes. Thus, events can be generated either by input devices or other event handlers.

An advantage of UIMSs based on event languages is their ability to handle multiple processes, including *multi-thread dialogue* that presents multiple task paths available to a user at any point of a dialogue. Consequently, multiple interactions are easier to program. However, control flow in event-based languages is not localized (with changes easily propagated) — making it more difficult to create, understand, and debug code.

A Language for Generating Asynchronous Event Handlers (ALGAE) [16] is event based and supports message passing in a multiprocessing environment. Event specifications, written in a special-purpose Pascal-like language, form an interface specification. ALGAE generates event handlers from this specification.

The University of Alberta UIMS [25], based on the Seeheim model, also uses event specifications, written in a C-like event language, to generate event handlers. Instances of an event handler are created at run-time.

Other event-based systems include Squeak [6] — a textual language for mouse-based user interfaces, and the Sassafras UIMS [33], which incorporates the Event-Response Language — an event-based language that can support parallel dialogues.

Graphical Methods

Direct manipulation interfaces, which have a highly interactive nature and allow the use of a mouse to select and manipulate screen objects, are difficult to specify with language-oriented representations. In fact, such representations are not well suited for direct manipulation interfaces. Easier to use are graphical specification methods that use a pointing device, such as a mouse, to manipulate objects on the screen. This manipulation allows the definition of part or all of the interface. Some systems can even be used by a user, as opposed to a designer. But, ease of use complicates building the UIMS itself. Moreover, graphical techniques may not always support an extensive variety of interaction techniques.

MENULAY [3] is a preprocessor serving as the front end of a UIMS. MENULAY allows the designer to specify the graphical and functional relationships within and among the displays making up a menu-based system. It allows the placement and drawing of objects such as icons and other images on the screen. When the user selects that object, a semantic routine, written in a conventional programming language and linked to that object, is called. MENULAY code is compiled and linked to the run-time system that executes the user interface and handles user interactions.

The Dialog Editor [5] supports building user interfaces by direct manipulation. The designer can directly place interaction techniques, such as dialogue boxes and menus, on the screen; and designate places for input and output areas. It is up to the designer to specify, by typing, the names of action routines called when previously created interaction objects are user-executed.

Peridot [56,54] allows a designer to graphically create interaction techniques, such as scrollbars and menus, by manipulating lines, text, and other primitives. The paradigm used by Peridot is *programming by example*: by showing how a device or interaction object is manipulated, the user gives the system examples of how they should behave. By inference, Peridot can generate parameterized object-oriented code, from a designer's actions and sample parameter values.

The NeXT Interface Builder [1] uses a graphics editor to permit the graphical definition

of user interfaces. Consequently, a user-interface designer can construct a graphical user interface by the on-screen selection and manipulation of objects, such as menus and buttons, from an object library. The Interface Builder also allows the specification of actions for objects to perform, in response to user actions. For example, a designer can select a certain kind of button object from the Interface Builder's on-screen inventory of objects, move it to the desired screen location, label it, and attach an action to be performed when the user clicks on it. Moreover, a designer can create a custom object by first selecting a similar object and customizing its behavior and appearance. The interface specification developed using the Interface Builder is saved to an interface file. The compilation of this file places its interface data in an executable file. Such a binary description of an interface allows its integration into programs.

Other Methods

Having a knowledge-based representation of a user interface, the User Interface Development Environment (UIDE) [20,18] supports user-interface design and implementation. This representation consists of a class hierarchy of objects, object properties and actions, and preconditions (predicates that must be true for an action to occur) and postconditions (exist after an action has been executed). The knowledge base can generate a description of a user interface in Interface Definition Language [22]. Generation of different, but functionally equivalent interfaces, is possible by transforming [17,21] the interface represented in the knowledge base.

The Menu Interaction Kontrol Environment (MIKE) [62] permits a programmer to provide a list of application procedures and their parameter names and types. A menu is created from this list and then displayed to provide a simple interface. The user selects a procedure by typing a prefix unique to a procedure name. If the procedure has parameters the system prompts the user for each parameter. Once all parameters have been specified, the application's semantics are executed. MIKE allows icons to be used for some commands, and permits the designer to interactively change the interface with a graphical interface editor. The editor acts as a specification guide and obviates the need to learn new notational forms.

8.2 MUISL: The McGill User Interface Specification Language

Based on the definition and manipulation of user interface objects, MUISL is the author's experimental language for the specification of event-driven user interfaces. A component of MUISL, a *user interface object* is an entity that describes a certain user-interface interaction method. Examples of objects include a text window, a window containing other objects, a menu, a command button, and so on. *Classes* of objects describe a group of similar types of objects and contain default object-attributes and operations. Defining an object requires that it be assigned a class; it then has available for its use, the attributes and operations of that class. An object's *attributes* define characteristics such as its dimensions, contents, and so on. *Operations* are requests to carry out a command on an object, usually to retrieve information from the object, or change certain aspects of the object. Operations can embody the behavior of an object, allow inter-object communication, and specify relationships among objects. The set of an object's class-operations and attributes acts as the external interface of the object. Whereas a *class* identifies the type of the object, its *superclass* is the class from which it inherits attributes and operations.

An MUISL object-definition is based on the specification of the object's class, and optional: superclass, local variables, attributes, and actions. *Actions* contain a statement sequence consisting of MUISL statements, legal operations, any user-defined programming-language statement, and callback and event handler procedure-definitions. By using the proper operation, previously defined objects can be *instantiated*, or created, one or more times. Instantiation allows the specification of parent-child relationships because an object is instantiated as a child of another. Instantiation also makes *active* any defined event handlers or callbacks for that objects. Only then can these procedures receive events for that object.

8.2.1 Assumptions and Scope of Work

It is assumed that a MUISL specification is the input of a *MUISL tool*, currently *muigen*, which scans and parses the specification and generates user interface code if there are no errors. The generated code, in a *target programming language*, is to be compiled and run in a *target window system*. The MUISL tool is responsible for: the definition and implementation of classes, attributes, operations, and so on; and their mapping to an equivalent target

programming language and window system structures and statements.

MUISL assumes an underlying event-driven target window system and model of execution. Run-time aspects, such as low-level device interaction and management of events, are assumed to be handled by the target window system. It is assumed that the target window system supports the specification (to be called *registration*), management, and timely invocation of callbacks and event handlers. For example, MUISL allows the specifier to attach one or more procedures (event handlers) to a particular event, for a particular object. When the target window system detects the event on this object, it calls the attached event handler(s). For events not specified, it is assumed that the target window system has default event handlers.

MUISL also assumes that the target window system is responsible for flow of control and contains a main interaction loop that detects events, and calls appropriate routines. There is no need for the program represented by a MUISL specification to do any kind of polling. It is assumed that a MUISL tool generates one or more calls to target window system routines implementing this loop, after generating code specified by the MUISL specification. A window system such as the X Window System, satisfies the above conditions, and is well suited as a target window system; accordingly, C is appropriate as a target programming language because of its easy interface to the X Window System.

Although MUISL assumes an event-driven run-time model, it does not mandate a certain target programming language or window system. These are functions of the MUISL tool. At present, *muigen* uses the X Window System and the C programming language, respectively.

MUISL is intended for a programmer (to be called a *MUISL specifier*) and not a user. It does not assume that this programmer is knowledgeable in the target programming language and window system. However, if the generated code is to be modified and linked with other code, such knowledge is useful. Accordingly, the best user of MUISL is a programmer knowledgeable in both the target programming language and window system. Nevertheless a window and menu-based interface can be built, on top of MUISL, in order to guide a user in the MUISL-based specification of a user interface. Implementation of such an interface was not investigated, since the emphasis was to design MUISL and test its viability with *muigen*.

MUISL is an *experimental*, and not a production, language for the specification of user interfaces. Its current state is intended to show the basic characteristics of a specification language. Similarly, *muigen* is a sample tool to show the usefulness of MUISL. The work

done for this thesis represents initial steps towards future work; MUISL and *muigen* can easily be expanded to support more complex interactions.

Accordingly, MUISL is textual, not graphical. The intention was not to design a graphical language. As a result, MUISL does not support the specification of graphical objects such as lines and polygons.

Moreover, the classes, attributes, and operations (see Appendix D) that *muigen* currently supports are basic and limited. The objective for this thesis was to present a usable subset of interactions and not to provide for a multitude of fancy interactions. Given *muigen*'s architecture (see Appendix C), the addition of new classes, attributes, and operations is easily accomplished.

How does MUISL compare to current languages? This question is answered as part of an evaluation of MUISL, in Section 8.4.

8.2.2 The Language

The section is intended to discuss features of MUISL, whose lexical rules and grammar are provided in Appendix B.

A MUISL specification is created with a conventional text editor, and subsequently resides in an ASCII file. This specification has three main blocks: an optional global-variable declaration-block, *object definitions*,¹ and the *initialization block*. The following is an outline of a minimal MUISL specification, with ... indicating omitted specifications, and comments preceded with a #:

```
OBJECT
... #object definition contents here
END #of object definition
INIT #start of initialization block
... #may be empty
END #of initialization block
```

Lexical Restrictions

MUISL is a case sensitive language, and its reserved words (see Table 8.1) are in upper case so that they are prominent in a specification. The type identifiers mentioned in that

¹At least one object definition is required, because the purpose of a specification is to provide for the definition of objects

table are: INTEGER, REAL, CHAR, CARDINAL, OBJECT_ID, STRING, BUTTON_ID, BOOLEAN, KEYCODE, DIMENSION, and POSITION.

Word	Purpose
ACTIONS	Starts actions section
ATTRIBUTES	Starts attribute definition
BUTTON	Identifies a button statement
CALLBACK	Defines a callback procedure
CASE, OF	Start key/button statements
CLASS	Precedes object's class
ELSE	Starts else portion in conditional/key/button statements
END	Ends initialization block, object definition, and some statements
EVENT	Defines an event handler
IF, THEN	Start conditional statement
INIT	Starts initialization block
KEY	Identifies a key statement
NAME	Precedes object's name
OBJECT	Starts object definition
SUPERCLASS	Precedes object's superclass
Type identifiers	Define a type
VARIABLES	Starts declaration block

Table 8.1: Reserved Words in MUISL

Special MUISL symbols are shown in Table 8.2. The arithmetic symbols of that table are $<$, $>$, \leq , \geq , $==$ (equality), $<>$ (inequality), $-$, $+$, $||$ (or), $!$ (not), $*$, and $\&\&$ (and). Usage of punctuation symbols is kept to a minimum in order not to burden the MUISL speaker with too many syntactic details. Punctuation is used when necessary, such as in a list of variables, in which a comma separates identifiers of the same type. No semicolon separates different variable declarations or statements: a typical MUISL specification would have each on a different line. One use of the colon is after a reserved word, such as `VARIABLES` in order to indicate that a sequence of items is contained after this reserved word.

Special identifiers are those with a special prefix. Special prefixes, shown in Table 8.3 ensure consistency in naming. The special identifiers named `atident/Cc/callbacks?` and `atident/Ee/eventHandlers?` respectively denote callback and event handler attribute-names². Other identifiers preceded by special prefixes are not mandated by MUISL, but a property

²The notation used is explained in Appendix B

Symbols	Purpose
	Follows some reserved words, or precedes an operation's argument value
=	Assigns values to variables or attributes
,	Separates items in a list
()	Enclose items in some lists
[]	Enclose an operation
@	Indicates an external declaration or action
"	Encloses a string
Arithmetic symbols	Indicate arithmetic manipulation

Table 8.2: Special Symbols in MUISL

of the MUISL tool: it is this program which is responsible for defining such names and ensuring that a MUISL specification adheres to them.

Prefix	Usage	Example
AT	Certain attribute values	ATwhite
arg	Operation argument names	argObject
at	Attribute names	atButtonCursor
button	Button names	buttonLeft
cl	Class names	clTextWindow
event	Event names	eventKeyPress
key	Key names	keyA
obj	Operation names	objRootInstantiate

Table 8.3: Special Prefixes in MUISL

Variables and Scope

Global variables used within any object definition or the initialization block are declared in a global-variable declaration-block. However, each object definition and the initialization block forms its local scope. variables declared within each of these are visible only within that object definition or initialization block. In an object definition, a local variable of the same name and type as a global one, takes precedence over the global variable. Object names, which form part of an object definition, are considered as implicitly declared global variables.

A variable declaration block is preceded with the token VARIABLES. The block consists

of any combination of zero or more external declarations or types; each type is followed by a comma-separated list of identifiers. Preceded with a @, an *external declaration* is a declaration in the target programming language. This type of declaration acts as an escape mechanism, allowing the MUISL specifier to declare variables whose type may not be available in MUISL. Note that any characters after the @ are copied verbatim, without any checking. The following example contains two consecutive external declarations

```
VARIABLES :  
    @char *str;  
    @int a[10];
```

No semicolon separates (non-external) declarations of variables of different types. For example, the following is a legal declaration:

```
VARIABLES :  
    INTEGER a, b  
    OBJECT_ID SomeObject
```

MUISL types are limited to those considered to be most useful in defining a user interface. For example, OBJECT_ID declares a variable that identifies an object; BUTTON_ID a variable used as the identifier of a mouse button; KEYCODE, a variable used as a key identifier; DIMENSION, a variable used for the dimensions of an object; and POSITION a variable used to identify a mouse cursor's *x*- or *y*- coordinate.

Object Definitions

The reserved words OBJECT and END surround an object definition, which provides an object's class and optional: superclass, local variables, attributes, and actions

An object used by another, must have been *instantiated* either in the initialization block or in the definition of another object (which itself was previously instantiated). Instantiation creates *an* instance of the object, and an object defined once, can have many instances. This shortens the amount of specification needed: one definition can apply to different places. For example, a text window can be defined once, but instantiated multiple times if an interface requires text windows in more than one place. This obviates the need for defining different text window objects, each with a different name, but performing essentially the same function.

Instantiation allows the specification of parent-child relationships: an object is instantiated as a child of another. The object instantiated before any other must be instantiated as the child of a special top level object, whose name is a property of the MUISL tool and is assumed to have been created as the first object.³ In the above example, the text window is instantiated as a child of multiple objects.

An object is uniquely identified by the *object name* — the identifier following the NAME token. Identifying the object being acted upon, the object name is essential when manipulating objects. The following example outlines an object definition and its name:

```
OBJECT
NAME : SomeName
... #The rest of the object definition is here.
END #of object definition
#Other objects or the initialization block can use SomeName.
```

As previously mentioned, an object name defined in one object definition can be used in other object definitions or in the initialization block. In fact, the object name is an implicitly globally declared variable of type OBJECT_ID; explicitly declaring it as an object variable in the global-variable declaration block is allowed, but redundant.

Classes

An object's class, indicated by the identifier after the CLASS token, denotes an object's type. This class makes available to the object, default attributes and operations for that class. MUISL does not dictate class names, except for the requirement that they be preceded with the *cl* prefix. Class names, and their properties are defined by the MUISL tool. For example, an object defined as a text window must declare its class as that of a text window as follows:

```
OBJECT
NAME : TextObject
CLASS: clTextWindow
... #rest of the definition;
    #attributes and operations of clTextWindow can be used
END #object
```

³In *mugen*, this top level object is called *ATtopObject*.

Classes form the inheritance hierarchy: an object's *superclass*, the identifier after the SUPERCLASS token, is the class from which an object's class inherits attributes and operations. Except for the highest class, each class has a default superclass. This allows superclasses to contain attributes and operations that apply to the lower classes. Superclassing reduces the number of attributes and operations that a lower class needs to declare because its superclass has already declared them.

MUISL supports only single inheritance: a class can be a superclass to many classes, but it can only have one superclass. Although a class inherits attributes and operations from its superclass, the class's attributes and operations override those of the same name, but belonging to its superclass. A class can also inherit attributes and operations from classes that are its distant superclasses, but the attributes and operations of the closest superclass override similar ones of more distant superclasses. Thus, when an object uses an attribute or operation, its class is first searched; if neither attribute nor operation name is found, the object's superclass (either default superclass of the object's class, or the name after the SUPERCLASS token) is searched. This upward process continues until the attribute or operation is found, or the root of the class hierarchy is reached and the search fails.

For example, if the class *clWindow* is a superclass to the class *clTextWindow*, and *clRoot* is a superclass to *clWindow*, then *clRoot* is a distant superclass to *clTextWindow*. Objects of class *clTextWindow* inherit attributes and operations of *clWindow*, and then, those of *clRoot*. When the object of class *clTextWindow* uses an attribute named *x*, the attribute names of *clTextWindow* are searched for one called *x*. If it is not found, those of *clWindow* are searched for the attribute *x*. If this search fails, the attribute names of *clRoot* are then searched. Because of this method of inheritance, a class has the same number of or more attributes than its superclass, but never fewer. A superclass groups attributes and operations common to all of its subclasses — classes that it is a superclass of. Sibling classes are those with the same immediate superclass; attributes and operations of a class are not visible to its sibling class. An attribute or operation is *visible* to a class, if the attribute or operation can be used in that class. Visibility is possible for attributes and operations defined in current class of an object, or any of that class's superclasses.

If the superclass name is not included in an object definition, the search for attributes and operations not found under the object's class (defined after the CLASS token) commences in the object's default superclass. Default superclasses are derived from the class hierarchy shown in Figure 8.2. Occasionally, an object of class A would need to use an

attribute or operation of class B, where class B is not a superclass (immediate or distant) of class A. In this case, the specifier can override the default superclassing mechanism by specifying a class name after the SUPERCLASS token. For example, an interface is to consist of a form box window *F* with two viewport windows, *V1* and *V2* as children, where *V2* is placed to the right of *V1*. To indicate that *V2* is to be placed to the right of *V1*, an attribute of the class *clFormBoxWindow* must be used in the definition of *V2*. However, the class hierarchy shows that *clViewportWindow* and *clPanedBoxWindow* are sibling classes: the form box child placement attribute (called *atFormBoxWindowLeftNeighborObject*), is not normally visible to an object of class *clViewportWindow*. For example, the following object definition does not allow any attribute or operation of *clFormBoxWindow* to be used:

```
OBJECT
NAME : V2
CLASS: clViewportWindow
#Attributes and operations follow. None of clFormBoxWindow's can be
#used here because the immediate superclass is clCompoundWindow, and
#none of its subsequent superclasses are clFormBoxWindow.
...
END
```

The solution is to override the default superclassing as follows:

```
OBJECT
NAME : V2
CLASS: clViewportWindow
SUPERCLASS: clFormBoxWindow #overrides default
ATTRIBUTES. #Here, attributes of clFormBoxWindow can now be used.
    atFormBoxWindowLeftNeighborObject = V1
        #V1 is the left neighbor of V2.
END
```

The drawback of the above scheme is when an object *A* needs to use attributes or operations of *two* or more classes, both of which are not superclasses to *A*'s class. One solution to this problem would be to allow multiple inheritance [51]. Another would be to provide some mechanism to override the default superclassing, at the level of each attribute definition or statement. A third solution would be to make certain class-specific attributes

and operations, globally available in a common superclass. But, this dilutes the power of classing, in which attributes and operations are only associated with a certain class, and its subclasses.

Figure 8.2 shows the class hierarchy that *muigen* defines.⁴ Classes shown at the right of the figure are leaf classes that inherit from their superclasses, shown to the left of the figure. Class names used in a specification are translated, or *mapped*, to names understandable by the target window system. The class name following the CLASS token is required to be a leaf class, because leaf classes are the only ones that are presently mapped to target window system names. Most of the leaf classes were chosen because of their ease of mapping to X Toolkit *widgets* [49], which are objects providing user interface abstractions. This ease of mapping simplified the implementation because no new types of leaf objects were needed, the goal was to show a workable language that would simplify and speed up specification of user interfaces and not just to define classes. The current mapping of leaf classes does not restrict MUISL to objects in the X Toolkit: class names and their mappings can be altered to any ones that can be supported in an event-based window system.

First and Second Level Classes

The topmost class is *clRoot*, which has no superclass, but is the immediate superclass to *clWindow*, *clButton*, and *clMenu*. Being the top class, *clRoot* contains attributes and operations common to all classes and inherits no attributes and operations from other classes. Attributes such as the height and width of an object are common for all objects, operations such as object instantiation and destruction are also common for all objects.

The other second level classes — *clWindow*, *clButton*, and *clMenu* — divide interaction styles into windows, buttons, and menus, respectively. These were chosen because a window based user interface usually consists of windows, buttons, menus, or any combination of them.

Windows

Windows are either simple (class *clSimpleWindow*) — containing no other objects such as subwindows, or compound (class *clCompoundWindow*) — containing one or more objects. For compound windows, these objects can be other compound windows, or simple objects such as buttons and menus.

⁴Classes *clGraphicsWindow* and *clProgramWindow* are currently not supported

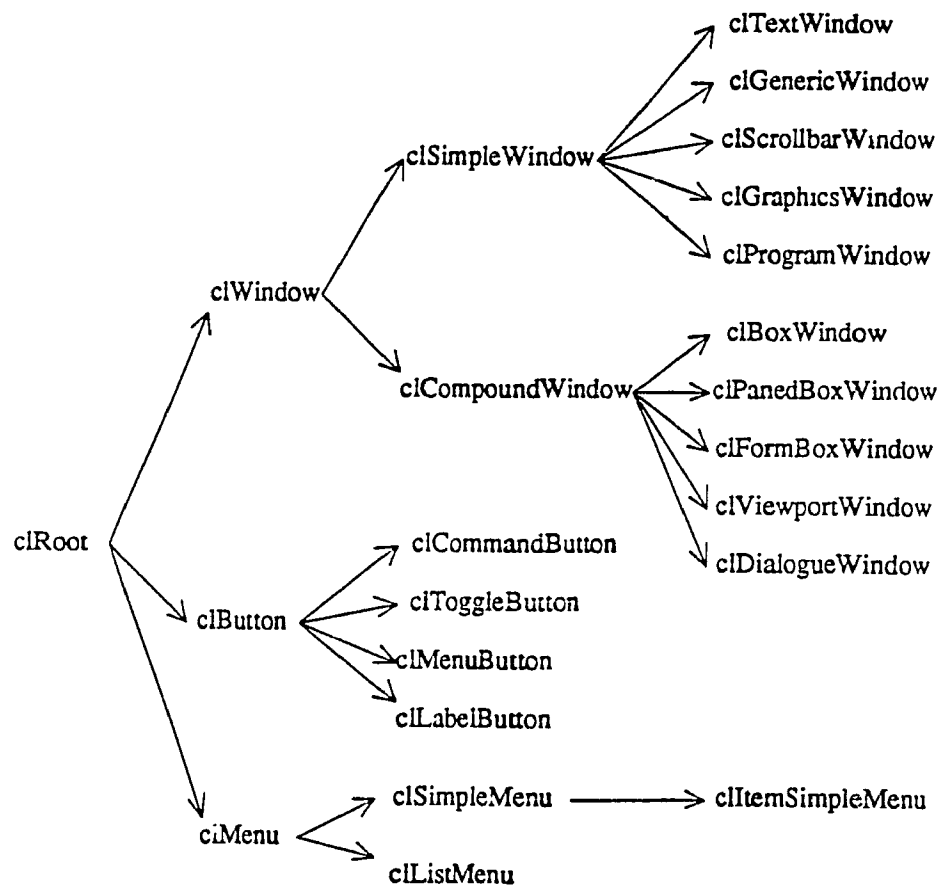


Figure 8.2: Class Hierarchy

Buttons

Class *clButton* is a superclass to classes such as command buttons (class *clCommandButton*), toggle buttons (class *clToggleButton*), menu buttons (class *clMenuButton*), and label buttons (class *clLabelButton*). All buttons are selectable rectangular screen regions that display a title. When the mouse cursor is over a button, the button's border is usually highlighted, indicating that it is the current focus of interest.

A *command button* is selected with the press of the left mouse button. The button's foreground and background colors are reversed, one or more programmer-specified actions are then executed, and the button reverts to its original state when the mouse button is released. Release of the mouse button, when the mouse cursor is outside of the command button, aborts the execution of actions associated with the latter button. Command buttons are useful for specifying a choice of different items to execute, as in a fixed menu.

A *toggle button* contains state information: the button is either set or not set by successive presses of the left mouse button. A *radio group* consists of a group of toggle buttons such that only one button can be set at any time. For example, a window containing printer options can have a toggle button for each type of printer, with only one printer being enabled at any instance. User interaction with toggle buttons is similar to that of command buttons.

A *label button* is a readonly button that displays a title or label. A *menu button* is a button that pops up a menu object of class *clSimpleMenu*, when the mouse cursor is inside the button, and any mouse button is pressed.

Menus

Third level classes whose superclass is *clMenu*, are simple-menu classes (class *clSimpleMenu*) and list-menu classes (class *clListMenu*). A list menu is a list of strings organized as a menu of columns or rows. Selecting any string, by pressing the left mouse button while the mouse cursor is over that string, calls one or more programmer-specified callbacks. These procedures are the same for each string.

A *simple menu* is a menu that contains one or more simple menu items, each of class *clItemSimpleMenu*. Simple menus are either pull-down menus (e.g. appearing when a mouse button is pressed) or pop-up menus (e.g. appearing for a certain combination of keys and/or mouse cursor clicks). Selecting a simple menu item executes one or more programmer-defined procedures.

Simple and Compound Windows

The fourth level of classes includes subclasses for *clSimpleWindow* and *clCompoundWindow*. Subclasses for the former include text windows (class *clTextWindow*), generic windows (class *clGenericWindow*), graphics windows (class *clGraphicsWindow*), program windows (class *clProgramWindow*), and scrollbar windows (class *clScrollbar*). Subclasses for the latter include box windows (class *clBoxWindow*), paned box windows (class *clPanedBoxWindow*), form box windows (class *clFormBoxWindow*), viewport windows (class *clViewportWindow*), and dialogue windows (class *clDialogueWindow*).

Text windows can be used to display text and allow a user to edit that text. These are text editors whose windows can contain optional scrollbars. *Generic windows* are the simplest types of windows and allow the MUISL specifier to manipulate them in any way. *Graphics windows*, currently not implemented, are to contain graphical objects, such as lines and polygons, which a specifier can create and manipulate. Also not implemented are *program windows* — windows executing a certain program, such as a UNIX shell. Program windows are often useful in interfaces such as window-based debuggers; for example, one subwindow can contain the text of the current source file, and another can be executing the debugger itself. *Scrollbar windows* can be used to provide scrollbars, at the programmer's control. For example, a scrollbar window can be used as a slider indicating the percentage done of a certain action.

A *box window* displays its children objects in an arbitrary fashion, left to the target window system. For example, a box window can have two immediate children: another box window containing command buttons, and a text window. For the display of vertically or horizontally tiled panes, a *paned box window* is useful. Each pane can be an object of any type, and is resizable, by default. In a *form box window*, the programmer can specify the location of one child with respect to another, unlike a box window. Containing a frame window with one inner window and one or two scrollbars, the *viewport window* acts as a viewport into the data of the inner window. The frame window clips non-visible data, and scrolling, which is managed by the viewport window, displays the appropriate part of the data. The MUISL specifier can manually create the equivalents of the graphics and text windows by proper management of a viewport window's contents. A *dialogue window* acts as a dialogue box, prompting the user for input. It usually consists of a text label, a text input area, and one or more buttons.

Attributes

Following the ATTRIBUTES token, attributes define the characteristics of the current object. The attributes whose values are set in an attribute definition override the default values set by the target window system and specific to the object's class.

Attributes are of three types: regular attributes, callback attributes, and event handler attributes. In an attribute definition, all kinds of attributes are set as follows:

$$< \textit{leftside} > = < \textit{rightside} >$$

The left side of an attribute definition contains the *attribute name*, which is a special identifier, prefixed with an *at*. This prefix is MUISL's only restriction: the rest of the attribute name is dependent on the names defined by the MUISL tool. For example, *muigen* uses the following convention for attribute names:

$$\textit{at} < \textit{class identifier} > < \textit{attribute name} >$$

For example, a typical attribute name is *atRootBackground*.

Regular Attributes

Regular attributes define object characteristics such as width and height. An *external attribute*, preceded by a @ and spanning a line, acts as an escape mechanism into the target window system: it allows any attribute definition allowed by the target window system. In an external attribute definition, it is assumed that the attribute name and value are valid in the target window system. It is the MUISL specifier's responsibility to ensure the correctness of attribute names and values in such an instance.

Non-external regular attribute definitions differ in the type of value specified on the right side: it can be a MUISL-defined *attribute value*, number, identifier, or string. MUISL-defined attribute values are special identifiers prefixed with an *AT*. These identifiers are defined by the MUISL tool. *Muigen*'s attribute names and values are tabulated in Appendix D.

The following object definition shows various types of regular attributes:

OBJECT

NAME: CommandButton1

CLASS: clCommandButton

ATTRIBUTES:

```

#external attribute --- valid in the target window system
@XtNinternalWidth = 6
#MUISL-defined attribute value is on the right side.
atButtonJustifyLabel = ATjustifyLeft
#Right side is a value (number or identifier).
atRootWidth = 20
atRootHeight = 10
#right side is a string
atRootLabel = "Quit"

```

END

Callback Attributes

Callback attributes, containing a list of one or more *callbacks*, are procedures called when certain actions (events) occur. As such, callbacks are a special case of event handlers. One use of callbacks is to contain calls to application procedures; for example, an application procedure can be called when the user presses a command button. The target window system is responsible for invoking callbacks, in the sequence of their specification, or registration, when the specific event happens. The MUISL specifier, however, only needs to specify them, without being concerned about how callbacks are invoked.

There are different kinds of callbacks associated with certain classes. A *destroy callback* is called when an object is destroyed, and is valid for all object classes. *Button* and *menu item callbacks* are called when the buttons or menu items are selected with a mouse click. *Menu callbacks* are called for the pop-up and pop-down of simple menus. A *text callback* is called for any change in a text window (class *clTextWindow*). *Scrollbar callbacks* are called for scrolling actions in a scrollbar object (class *clScrollbarWindow*).

Callback attributes are specified as a parenthesized list of one or more procedure names, each separated by a comma. The target window system calls them in the sequence of their declaration (from left to right). The current version of MUISL does not support the specification of parameters for procedure identifiers in a callback attribute. The following shows destroy and button callbacks for the command button object. MUISL allows a singular or plural version of the callback attribute name:

OBJECT

```

NAME : CommandButton1
CLASS: clCommandButton
ATTRIBUTES:
    #Proc1 and then proc2 are called when this object is destroyed.
    atRootDestroyCallbacks = ( proc1 , proc2 )
    #Proc3 will be called when the object is pressed with a mouse.
    atButtonCallback = ( proc3 )
END

```

Event-Handler Attributes

Whereas callback attributes are high-level methods of specifying procedures automatically called for standard user actions, *event-handler attributes* allow a lower level of specification: they consist of event names and associated procedures — event handlers — called when that event occurs. The MUISL specifier registers an event handler for a certain event, and the target window system is responsible for calling that procedure once the event occurs. In MUISL, registration of event handlers is accomplished with the event-handler attribute that consists of one or more parenthesized *event tuples*, each separated by commas. An event tuple⁵ is a pair of comma-separated event and procedure names: the procedure is automatically called whenever the target window system detects the event, for the current object. MUISL does not support specification of an event handler's parameters. Event names are special MUISL identifiers with the *event* prefix (see Appendix D). As with attribute names, event names are defined by the MUISL tool.

In the following example, procedure *proc1* is registered for the *KeyPress* event, and procedure *proc2*, for the *ButtonPress* event. This implies that the target window system calls *proc1* every time it detects the press of a key in the object *GenericWindow*, and *proc2* each instance it detects the press of a mouse button in that object. In this example, *proc1* and *proc2* are assumed to be procedures the MUISL specifier defines elsewhere.

```

OBJECT
NAME: GenericWindow
CLASS: clGenericWindow
ATTRIBUTES:

```

⁵ *Tuple* is used to mean a 2-tuple.

```

atRootEventHandlers = ( (eventKeyPress, proc1),
                        (eventButtonPress, proc2) )

END

```

If no callback or event handler is specified, the target window system calls its default procedures. If the contents of the specified callback or event handler are not respectively defined with the *callback* or *event-handler* statement, no procedure template is generated. It is the responsibility of the MUISL specifier to create such procedures in the generated file. Callback and event-handler attributes are well suited to MUISL's assumption of an underlying event-based target window system that detects events, manages them, and subsequently calls the appropriate callbacks or event handlers. Both types of procedures are powerful methods of controlling the behavior of objects in response to events.

Actions

Preceded by the ACTIONS token, the actions section of an object definition consists of zero or more statements. *Simple statements* can be *external statements*, *operations*, *assignments*, or *conditional statements*. *Procedure statements*, which can either be *callback statements* or *event-handler statements*, are usually containers of one or more simple statements. A callback statement generates a (callback) procedure template filled with simple statements specified in the body of this statement. An event-handler statement generates an event-handler template with the statements specified in the body of this statement. Simple statements not enclosed by a procedure statement, are generated in the procedure that defines the object in which they are specified. Simple statements in the initialization block are generated in a separate procedure.

External Statements

Prepended with a @, external statements act as an escape mechanism into the target programming language: they allow the specification of any statement in that language. Usage of external statements not only makes MUISL an escapable language, but also adds flexibility to it. The following partial object definition shows an external statement:

```

OBJECT
... #object name, class, etc.
ACTIONS :

```

```

#MUISL does not have print statements.
@printf("Creating object");
... #rest of actions
END

```

Operations

An operation is a request to execute a command on an object. Surrounded by left and right square brackets, an operation is denoted by the *operation name* (indicating what operation to execute), and a sequence of one or more *arguments* (indicating what values an operation should use).

MUISL requires that operation names contain a leading *obj*, but it does not dictate the specific names of operations. Operation names are dependent on the names defined by the tool implementing MUISL. *Muigen* uses the following convention for operation names

obj<*class identifier*><*operation name*>

For example, a typical operation name is *objRootInstantiate*. Operations are defined for certain classes, and are valid only for the class for which they are defined, or any subclasses of that class. The *objRootInstantiate* operation, for example, is valid for any object of class *clRoot* or any subclass of *clRoot*. In this case, the operation is valid for all classes of objects since *clRoot* is the top class. Further details of operations are in Appendix D

Each argument of an operation consists of an argument name and value, separated by a colon. As with operation names, MUISL does not place restrictions on argument names except that they start with a *arg* prefix. Semantically, the receiver of an MUISL operation (that is, the object to which the operation applies) must appear as an argument. In the current implementation of *muigen*, the receiver of an operation is the argument value whose argument name is *argObject*. The receiver was made as an argument, instead of a separate value after the operation name, in order to keep the syntax of operations uniform

Corresponding to an argument name is an *argument value*, which is the value used by the operation, for that argument name. Argument names are needed to identify argument values, and to allow arguments to be specified in any order. Argument values can be identifiers (such as a pre-declared variable identifier), numbers, strings (such as a file name), attribute names, or attribute values.

The following example shows a partial specification using operations. The object is defined, then one instance of it is created in the initialization block. The resulting user interface will display a string in a text window.

```

OBJECT
NAME: TextObject
CLASS: clTextWindow
ATTRIBUTES:
    atTextWindowUpdate = ATeditable #default is ATreadOnly
ACTIONS:
    #This will be generated in the same procedure that defines the object.
    [ objTextWindowMessage argObject: TextObject argText: "Hello world" ]

END

INIT #initialization block
    #Create an instance of the text object; the parent object of the top
    #MUISL object is given by the attribute value, ATtopObject. The
    #object is created as a fixed window always visible on the screen
    #(ie. ATnoPopup).

    [ objRootInstantiate argObject: TextObject argParent: ATtopObject
      argPopupType: ATnoPopup]

END

```

Assignment Statements

Assignment statements allow variables to be set either to values of expressions or results of operations ⁶ For example, because MUISL does not currently support operations as parts of expressions, an operation returning a value can be assigned to a variable which can later be used.

Expressions on the right side of an assignment statement are a subset of typical expressions found in a programming language. The MUISL grammar in Appendix B shows a precedence among operators in an expression: this precedence is only to simplify parsing

⁶ *Mugen* guarantees that an operation returning a value and used in an assignment generates a single statement returning a value

by the MUISL tool. The precedence rules of the target programming language apply to the generated code.

Conditional Statements

A conditional statement allows the selective execution of two sets of statements, depending on the value of a condition. Note that conditionals can be nested.

The following example shows a conditional statement and an assignment, both in the context of a callback for a list menu. The index of the selected menu item is retrieved, and an action is taken, based on a condition:

```
OBJECT
NAME: ListMenuObject
CLASS: clListMenu
ATTRIBUTES:
    #Set the appropriate attributes.
    ..
VARIABLES:
    INTEGER index
ACTIONS:
    CALLBACK ( atListMenuCallback, #attribute name for callback
               proc1 )
        #Defines a callback called when a list menu item is pressed
        #The CALLBACK statement will be explained later.

        #Get the index of the menu item pressed.
        index = [objListMenuGetCurrentItemIndex argObject:ListMenuObject]
        IF index == 1 THEN
            #Do something with the index value.
            ...
        ELSE
            #Do something else.
            ...
        END # of if statement
    END #of callback
```

```
END #of object definition
#rest of the specification
...
```

Procedure Statements

The MUISL specifier can provide the contents of a callback or event handler in one of two methods: either as a MUISL procedure statement; or as a target programming language procedure added to the generated file, or linked with it. The procedure statement allows the specifier to concentrate on the contents of the callback or event handler, instead of having to worry about their syntax in the target programming language. Because MUISL assumes an underlying event-based target window system, callbacks and event handlers are usually heavily used in interactions with an object. Callbacks and event handlers are associated only with the object in which they are defined.

Callback Statements

Identified by the leading CALLBACK token, the callback statement is used to specify the contents of a callback. Recall that a callback is a procedure automatically called by the target window system, under certain conditions. A callback statement is not allowed for classes of objects (or their superclasses) which do not have corresponding callback attributes. For an object, a callback statement can be uniquely identified by the tuple following the CALLBACK token. The first element of the tuple is the name of the callback attribute for which the procedure is a callback. This name is used to check if a callback is allowed for the class of the object being defined. The second tuple-element contains the name of the callback.

Legal callback statements without corresponding callback attributes set in the attributes section, do not only generate a callback, but also a callback attribute for the object in which the statements are defined. Consequently, there is no need to define callback attributes if the corresponding callback statements are specified.⁷

One or more *different* callback statements are allowed for the same callback attribute (as is the case for callback attributes). The order in which callbacks are called, in the case of no defined callback attributes, corresponds to the lexical order of the callback statements.

⁷It is redundant, but not illegal, to do so.

An example of a callback statement was previously shown when the conditional statement was discussed. In that example, the callback statement for the list menu callback named *proc1*, was defined. There is no need for the following statement in the attributes section:

```
atListMenuCallback = ( proc1 )
```

Generated is a callback, named *proc1*, and containing the simple statements within the MUISL callback statement. These statements are mapped to the corresponding statements of the target programming language. The callback's title, default parameters, and other syntactic details, are automatically generated.

Event-Handler Statements

The EVENT token identifies the start of an event-handler statement. A tuple of event and procedure names uniquely identifies an event handler statement within an object definition. This statement is used to specify the contents of an event-handler which the target window system automatically calls, once it detects the event for the current object.

There is no need to specify a corresponding event handler attribute for an event-handler statement. Such an attribute is automatically generated, if it does not exist, but there is an event-handler statement. Unlike callback statements, only one event-handler statement per event, per object, is allowed.

An event-handler statement not only registers a procedure as an event handler for an event (i.e. it sets an event-handler attribute if it that has not already been set), but also generates the event handler itself. Other than the standard template code generated for an event handler, statements within the MUISL event-handler statement are mapped to statements in the target programming language, and are included as part of the event handler. When an event occurs, the target event-based window system automatically calls the event handler specified for the event. If an event handler has been specified either in an event-handler attribute or statement, it is the procedure invoked, otherwise, the invoked procedure is a default target window-system procedure.

Statements within an event handler are a sequence of zero or more simple statements *key statements*, or *button statements*. Identified with starting CASE and KEY tokens, a key statement is a case statement whose case labels are key names. Semantically, such a statement can be included in an event-handler statement for a keyboard event, such as a

key press or release. Each element of this case statement contains the name of a key, and zero or more simple statements that are executed if the keyboard event involved that key. A simple statement sequence labeled by two or more key names is executed if a keyboard event is detected for any of those keys. If no key matches the key activated, the statement sequence in the optional ELSE-part of the key statement, is executed.

Identified with starting CASE and BUTTON tokens, button statements are syntactically and semantically similar to key statements, except that the case labels are names of mouse buttons. A case element is selected if the button activated matches that of a case label.

MUISL does not dictate the names of keys or buttons, except that they be respectively preceded with a *key* or *button* token. It is the responsibility of the MUISL tool to map key or button names to those supported by the target window system. *Muigen's* names are tabulated in Appendix D.

Key and button statements were included as part of MUISL event statements because they allow an easy-to-understand and natural method of performing actions based the type of key or button involved. Each key or button statement generates a corresponding case statement in the target programming language.

The following example combines various types of event statements. The first event-handler statement will cause an event handler, named *handle_button_press*, to be generated. An event-handler attribute for the mouse button press event will also be added as an attribute of *GenericObject*. *Handle_button_press* will be called by the target window system whenever a mouse button is pressed while the mouse cursor is inside *GenericObject*. The other two event statements will respectively generate event handlers for a key press and the entrance of the mouse cursor into *GenericObject*.

```

OBJECT
NAME : GenericObject
CLASS: clGenericWindow
ACTIONS:
    EVENT ( eventButtonPress, handle_button_press)
    CASE BUTTON OF
        buttonLeft : #left mouse button
        buttonMiddle: #middle mouse button
        @printf("Left or middle button");
        ... #other actions here

```

```

        buttonRight: #right mouse button
            @printf("Right button");
            ... #other actions here
        ELSE @printf("Button not recognized");
    END #case button
END #event handler

EVENT ( eventKeyPress, handle_key_press )
    CASE KEY OF
        keyA:
            ... #actions based on key pressed
        ELSE @printf("Button not recognized");
    END #case key
END #event handler

EVENT ( eventEnter, handle_enter_window )
    @printf("Entered window");
    ... #other actions here
END #event handler
END #object

```

Initialization Block

The final portion of a MUISL specification consists of the *initialization block* which is identified by the leading INIT token. The initialization block is intended to be used as the container of statements that create instances of previously defined objects. The block is not limited to these statements, though. An initialization block generates a procedure with its contents mapped from the enclosed MUISL statements. This procedure is called within the generated code. Variables used in the initialization block can be declared after the VARIABLES token and colon symbol. An example of the initialization block was previously given.

It is assumed that the program generated from a MUISL specification passes control to a main interaction loop implemented by the target window system, and called after the initialization block.

Examples of complete MUISL specifications and their resultant user interfaces are shown in Appendix E.

8.3 Muigen: The MUISL-Based User-Interface Generator

This section briefly describes *muigen*, a tool that scans, and parses a MUISL specification file. If there are no errors, it generates user interface code, using C as the target programming language, and the X Window System (Version 11, Release 4, with the X Athena Widget Set, and X Toolkit Intrinsics) as the target window system. *Muigen* itself consists of about 8,000 lines of C code, using *lex* and *yacc*.

Muigen was designed and implemented with the purpose of providing a sample MUISL-based user interface generator which would demonstrate the viability of MUISL as a user interface specification language. *Muigen*'s architecture and method of generating files are described in Appendix C.

8.3.1 Definitions

Some definitions used include:

- Mapping: the process of translating from a MUISL name to one or more target programming language and window system names.
- Specification File (SF): a file containing a MUISL specification.
- Generated File (GF): the output user interface file generated by *muigen*; this file contains source code in the target programming language and is to be compiled and run under the target window system.
- MUISL tool: a program that accepts an SF as input and outputs a GF.
- MUISL specifier: the programmer writing an SF.
- *Muigen* developer: the programmer who codes, modifies, or modifies *muigen*.
- Initialization Files (IFs): files containing MUISL or internal *muigen* names and their mappings to target programming language and window system code, and other internal *muigen* names.

Steps in creating a user interface using MUISL and *muigen* are to: create an SF with any text editor; create the GF by invoking *muigen* on the SF; using a compiler and linker for the target programming language, compile and link the GF with any other source code files (for example, application source code files in the target programming language), and run the resultant user interface in the target window system.

8.3.2 Initialization Files

An important aspect of *muigen* is its use of IFs: it reads these files at the start of execution and dynamically binds their contents to tables. A MUISL name read by the parser is mapped to target programming language code by finding its entry in an appropriate table. IFs make *muigen* a table-driven program, thus minimizing the use of hard-wired information and facilitating changes of the target programming language and window systems. IFs also allow the *muigen* developer⁸ the ability to modify mapping information without the need for recompilation of *muigen*.

IFs are used to: initialize class names; define table names; map class names, attribute names, attribute values, methods, variable types, and event/key/button names to names in the target programming language; construct the class hierarchy; and provide target programming language code templates to be used in the generated-code file. Contents of IFs are further explained in Appendix D.

Based on grammar, the two types of IFs are: Operation IFs (OIFs), and Non-Operation IFs (NOIFs). The former map MUISL operation names into target programming language code and include information on the position and number of parameters of operation-names. The latter files map a name (such as a MUISL class name) to another entity, such as a target programming language code-template; or serve to dynamically initialize a *muigen* data structure.

Non-Operation Initialization-File Format

The grammar of NOIFs is shown in Figure 8.3. Notation and some tokens used are similar to those of Appendix B. Nonterminals are shown in *italics*; terminals are in **bold**, and *integer* denotes an integer value.

A typical NOIF consists of one or more definitions; each definition contains a logically related set of mappings, with each mapping consisting of a record. The unique identifier

⁸Initialization files are not intended for use by the MUISL specifier


```

definition_list ::= { definition }+
definition      ::= ~ ident integer record_list
record_list     ::= { record }+
record          ::= { ident , string }
string          ::= ' { character \ ' } '

```

Figure 8.3: Non-Operation Initialization-File Grammar

in a record, identifies the name to be mapped; and the string provides the mapping of this identifier. A string is used because some mappings consist of more than one name.

Each definition is identified by a *unique* name, followed by an integer indicating the number of records in this definition. *Muigen* allocates a table for each definition, and the size of this table is given by the integer for that definition. If the integer is smaller than the actual number of records, *muigen* prints an error message, and exits. The last record in each definition must be a null record; “empty” definitions must have this record. Reserved words in NOIFs include NULL (for a null record element), and UNSUPPORTED (for an element not yet supported by *muigen*).

The following example shows part of the IF that maps MUISL’s attribute names to corresponding X Window System names. Comments, preceded with a # character, cause the remainder of the line to be ignored:

```

~ clMenuAttributeTable 3 #number of records for this example

{ atMenuCursor, 'XtNcursor' }
  #Description: Menu's default cursor
  #Values: AT*Cursor

{ atMenuForeground, 'XtNforeground' }
  #Description: Menu's foreground color
  #Values: ATwhite, ATblack; default: ATblack

{ NULL, 'NULL' } #last record must be null

~ clCompoundWindowAttributeTable 1

```

```
{ NULL, 'NULL' } #last record must be null
```

Operation-Initialization-File Format

The grammar of OIFs is shown in Figure 8.4. The notation used is similar to that of Section 8.3.2. The main difference between NOIFs and OIFs is the format of each record; otherwise, these types of files are similar. A NOIF record is structured as such, in order to allow *muigen* to check operation arguments. The record elements from the second onwards are strings since they may contain more than one identifier. In Figure 8.4, the first record element identifies the operation name; the second element, the mapping of this name (%s's indicate places for argument values); the third, a list of MUISL argument names whose values will respectively replace the %s's; the fourth, a list of MUISL argument names present in the MUISL operation; the fifth, the return type (NULL if none). Lists in record-element strings consist of identifiers separated by one or more spaces.

```
definition_list ::= { definition }+
definition      ::= ~ ident integer record_list
record_list     ::= { record }+
record          ::= { ident , string , string ,
                    string , string }
string          ::= ' { character \ ' } '
```

Figure 8.4: Operation Initialization-File Grammar

The following is an example of a small part of an OIF:

```
~ clRootMethodTable 3 #for demonstration purposes

{ objRootInstantiate, ' %s = create%s(%s,%s);\n',
  'argObject argObject argParent argPopupType',
  'argObject argParent argPopupType', 'NULL' }
# [ objRootInstantiate argObject:<ident> argParent:<ident>
#           argPopupType: <attribute_value>]
# Valid attribute values: ATnoPopup, ATmenuPopup, ATobjectPopup,
#           ATdialoguePopup
```

```

{ objRootDestroy, 'XtDestroyWidget(%s);\n', 'argObject',
  'argObject', 'NULL' }
# [ objRootDestroy argObject:<ident>]

{ NULL, 'NULL', 'NULL' , 'NULL', 'NULL'} #last record must be null

~ clButtonMethodTable 1 #last record must be null
{ NULL, 'NULL', 'NULL' , 'NULL', 'NULL' }

```

8.4 Evaluation of MUISL and Muigen

This section evaluates both MUISL and *muigen* by examining the following issues: ease of use, ease of understanding, support for prototyping, separation of the interface from the application, underlying concepts and syntax, completeness and correctness, extensibility and escapability, locality of definition, functionality, portability and availability, programmer control, ease of design and implementation, and the relation to *xmupe2*. When appropriate, comparisons are made to other systems.

Ease of Use

The first issue to be examined is usability, or ease of use. Being textual, a MUISL specification is created as an ASCII file, using any text editor. The specification is easily modifiable by editing the SF. *Muigen* accepts this SF and generates the resultant user interface code. This code is then compiled and linked with other application code, and executed by the target window system.

However, MUISL is less easy to use than graphical systems such as Peridot [56,54], or the Dialog Editor [5]. These systems can permit even users to specify user interfaces. A disadvantage of using MUISL is that there is no interface editor — such as that in the Dialog Editor, MIKE [62], or MENULAY [3] — to simplify the process of building an interface specification. However, the author's intention was not to build one, but instead concentrate on the design of MUISL and the design and implementation of *muigen*.

Except for the lack of an interface editor, usage of MUISL and *muigen* is similar to that of other systems. For example, in MIKE, the generated user interface is compiled and linked with application and library code. MENULAY is also similar in that the result of a

session with the interface editor is a specification stored in a file, which is generated into C code, compiled, and run.

Ease of Understanding

MUISL is a textual language and the MUISL specifier must invest time to learn aspects of the language. However, MUISL is easy to understand and learn. For example, there are few parts to an object definition, standard names follow a certain convention, and notation of operations is consistent. MUISL clearly is less difficult to understand than its target window system.

Although the language is not complex, it places certain limitations that the MUISL specifier must be aware of. For example, MUISL has some lexical limitations on some types of identifiers: key names must be preceded with *key*; attribute names, with *at*; standard attribute value identifiers, with *AT*; and so on. But, these prefixes add uniformity and consistency to the usage of certain names. An advantage of such a naming scheme is its flexibility: the same names can be mapped to names in different target window systems and programming languages. MUISL is also case sensitive and requires reserved words such as **OBJECT**, **CLASS**, and so on, be in upper case.

The event- and object-based model, in which objects can be defined with optional attributes and operations, enhance MUISL's understandability. Inheritance of class attributes and operations reduces the need for repetition of common attributes and operations. Creation of objects is accomplished with one or more instantiations. Finally, by assuming an event-based model, whose run-time aspects are handled by the target window system, the MUISL specifier need not understand how events within objects are handled. They are the responsibility of the target window system. By incorporating event handlers and callbacks, MUISL assumes an external (user interface) instead of an internal (application) model of control: the user interface is responsible for calling application routines. One of the previously mentioned advantages of external control is its support for prototyping.

A possible disadvantage of MUISL is that it is intended only for a programmer. This person must also understand the underlying object and event bases of MUISL. Graphical systems such as Peridot or the Dialog Editor obviate the need for learning any language, are easier to understand than MUISL, and can be used by users.

Support for Prototyping

Using MUISL and *muigen* allows the rapid prototyping of user interfaces, and facilitates the specification, design, and implementation of user interfaces. The MUISL specifier can use this language to completely specify a user interface, without necessarily having to write supporting code in the target programming language. The user interface is then compiled and run: if changes are required, the MUISL specifier can easily change the MUISL specification, generate new user-interface code, compile it, and re-run it. The MUISL specifier saves time and effort and achieves a quick turnaround time, because there is usually no need to repeatedly write or modify lengthy or intricate user interface code in a conventional programming language. Modification of the smaller SF is typically easier and quicker. The result is a concentration on the functionality of a user interface, rather than on implementation details specific to a window system. As with other systems, the emphasis is on providing tools to assist the user interface specifier.

For example, each of the examples in Appendix E was initially constructed in fewer than twenty minutes. The equivalent time to construct them from scratch, using a conventional programming language, would have taken over an hour each.⁹ The reduction of effort and time also extended to the testing and modification of the MUISL specification files, and proved to be easier and more convenient than an equivalent modification of conventional programming language code. The author was capable of easily experimenting with different aspects of the user interfaces in each example, and rapidly customizing the final result with no difficulties. MUISL specifications in this appendix were six to eight times smaller than the generated code.

In supporting rapid-prototyping, MUISL and *muigen* are similar to other UIMSs. For example, the Sassafras UIMS [33] also supports the rapid development of user interfaces by using the iterative development approach — testing of the interface is possible independently of the application. Another example is the Dialog Editor, which also permits the quick building, and modification of a user interface, without affecting the application.

Separation of the Interface from the Application

MUISL allows the separation of the user interface from the application for which the interface is intended. By stressing the form and interactions within a user interface, MUISL abstracts the design of a user interface from application code. But, MUISL provides hooks

⁹Both times assume a specifier knowledgeable in MUISL or the target window system, respectively. The latter, of course, is *much* harder.

into the application. For example, callback procedure statements can contain calls to application code. The result is a user interface that is a separate-modular entity

UIMSs such as Sassafras also support user interface and application routines to be separated without limiting their ability to exchange data. Other systems also encourage dialogue independence. For example, COUSIN's slots [30.32.31] encourage thinking in terms of data exchanged by the application and user, instead of how they are displayed or modified

Underlying Concepts and Syntax

MUISL includes concepts and resulting syntax similar to those of other systems. The object-oriented paradigm is similar to GWUIMS's [74]. MUISL has classes, attributes, operations (corresponding to methods), and inheritance — concepts embodied in GWUIMS. GWUIMS identifies an object with tokens such as *Class*, *Attributes*, *Methods*. MUISL also has similar tokens to identify respective parts of an object definition. Like GWUIMS, MUISL allows communication with an object by specifying the object's name or id (as an argument), the name of the operation, and a list of other arguments. MUISL's use of a class hierarchy, objects, their attributes, and actions is also similar to their use in UIDE [20.18]. Some syntax used in both systems, such as the use of parenthesized lists of comma-separated items, is identical.

MUISL's use of an event-based paradigm is similar to that of event-based systems such as the University of Alberta UIMS [25]. The model in both assumes that when an event occurs, it is sent to the proper event handler(s). In the above UIMS, only active event handlers can receive events; an event handler is made active by using an explicit statement to create it. In MUISL, simply defining an event handler in an object definition is insufficient to make it active. An event handler is made active by the instantiation of the object for which it is a handler. MUISL similarly assumes a conceptual model of active event handlers executing concurrently and processing events as they come in. There is nothing in MUISL which prevents an event handler from invoking another event handler, or deactivate an active event handler: this ability is a function of the available operations defined by the MUISL tool. Note that MUISL allows the definition of the same event handler for different events, or different objects.

The usage of tokens such as *EVENT*, *VAR*, *IF*, and *INIT*, and assignment statements in MUISL is similar to their usage in the University of Alberta UIMS. In both, event handlers are defined for particular events. In the latter system, a file named as an event handler

file, contains sections defining: parameters to the event handler, its local variables, events it can process, and bodies of procedures, each of which responds to a particular event. In MUISL, event handlers are also attached to a particular object and make use of local and global variables and supported statements.

Some of the syntax of MUISL's operations was inspired by Smalltalk's messages, but there are some differences: in an operation, the operation name is listed first and as previously mentioned, the receiver of an MUISL operation must appear as an argument. Moreover, the syntax of Smalltalk-like messages is unconventional, and operation syntax seems more natural to those used to procedure calls. However, an MUISL operation is not necessarily a procedure call, although it may generate such a call. An MUISL operation can generate one or more statements in the target programming language. Like a Smalltalk message, an operation serves as a modularity mechanism: it specifies *what* command should be carried out, but not *how* it is accomplished. The latter is achieved by mapping the operation to the generated target programming-language code.

Completeness and Correctness

Can *mugen* guarantee that a MUISL specification is correct or completely specifies a user interface? Completeness is a difficult requirement that remains an open question with many user interface tools. *Mugen* attempts to assess correctness as much as possible: lexical and syntactic errors are easily detected. Semantic errors, such as type clashes in assignment of operations' results to variables, and bad arguments to a operation, are flagged. However, some errors, such as those in external statements, are beyond the scope of *mugen* and are left to the compiler of the target programming language. *Mugen* can only guarantee correctness to a certain degree. The onus is on the MUISL specifier to guarantee the rest.

Extensibility and Escapability

By including external statements and declarations, MUISL is escapable. Both allow the inclusion of arbitrary target programming language statements and declarations that MUISL does not support. The generated file resulting from a specification file also gives the MUISL specifier both the ability to modify user interface code, and the flexibility to extend or tailor this code according to the particular application. This is because the generated file is essentially a program in the target programming language. The only proviso is that the MUISL specifier be familiar with the target programming language and window system.

A program driven by tables dynamically initialized from initialization files (IFs), *muigen* is easily extensible. IFs also enhance the power and flexibility of *muigen*, and permit the modification of features such as operations and attributes. For example, the class hierarchy shown in Figure 8.2 is stored in one IF. To change this hierarchy, the MUISL *developer* must only update this IF. No *muigen* code has to be altered, and recompilation of *muigen* is not necessary. However, adding a new class requires modification of several IFs. This is presently a manual job, but is an excellent candidate for automation. IFs also facilitate the alteration of the target window system or programming language, because only IF manipulation is required. The time to read IFs and initialize the corresponding tables is negligible.

MUISL's classing mechanism and inheritance of attributes and operations support locality of definition. A change in the attribute or operation of a particular class applies to all inheriting classes. Recall that objects can use the attributes section to change the default attribute values of their classes. A system such as the Dialog Editor is similar in that it shares resources, such as a default background color, that apply to multiple user-interface interaction-objects. Changing a shared resource applies to all objects sharing that resource.

Functionality

Given its limitations, MUISL can still produce useful, viable and functional user interfaces. The MUISL specifier can use the generated file, as is, or modify it, to produce the user interface. MUISL is powerful enough to specify both the attributes and operations of an interface, in addition to the hierarchy of objects. Usage of objects as building blocks of a user interface adds to MUISL's usable power (set of user interfaces that can be built), by allowing the construction of complex objects from simple ones (see the examples in Appendix E). GWUIMS also has this feature. A drawback of this is the increased amount of specification entailed; for example, to create a window of command buttons, a container window and each of the command buttons must be defined separately. Then, this container must first be instantiated, and each command button has to be instantiated as a child of this container.

The current version of MUISL does support a number of interactions, such as windows, menus, buttons, scrollbars, dialogue boxes, and so on — many of which are similar to those of the Dialog Editor. Numerous usable user interfaces can be built solely from these types of interactions. MUISL also allows the creation of user interfaces that handle input from, and

output to, objects. This is possible because the run-time aspects of input and output are assumed to be handled by an event-based target window system. For example, operations to write to a window generate equivalent window-system code which handles output into the proper screen area.

All UIMSs are restricted in the forms of user interfaces that they can generate [76]. MUISL, as a specification language is no exception. Other systems, such as Peridot, are easier to use, provide more functionality than MUISL, and are aimed at producing graphical, direct-manipulation interfaces. But, Peridot cannot help with the textual command interfaces, or with the coding of the semantics of an application [56].

Portability and Availability

Not constrained to a certain target programming language or window system, MUISL is portable¹⁰ The current target programming language, C, is widely available; the current target window system, the X Window System, has been ported to a wide variety of architectures, and is a popular system. There are a number of similarities in MUISL to window systems, such as the X Window System, but at a higher level of abstraction. This is readily apparent in the choice of some attributes and operations: MUISL attempts to shift as much responsibility as possible to the target window system, and to concentrate on the form of, and interaction within, a user interface.

Some UIMSs are based on possibly non-portable run-time or window systems. Systems such as MENULAY, or SYNGRAPH [63], generate code in conventional programming languages such as C and Pascal, respectively. *Muigen*, however, can easily change the target window system or programming language.

Programmer Control

MUISL does enforce a particular style of interface: that based on objects, their attributes, operations, and events. By mapping MUISL specifications to the target window system with its standard object attributes and operations, *muigen* provides the programmer with uniformity across all interfaces. But, the programmer is free to change the generated code, which usually has to be linked with the programmer's application code. The ability to modify the generated code allows the programmer considerable control.

¹⁰However, the target window system must have an event-based run-time model.

Ease of Design and Implementation

The most difficult part of the work was the design of MUISL: the current language is the last of numerous earlier versions. Usage of objects, classing, attributes, operations, and an event-based model introduced consistency and simplified the work. Once MUISL was relatively stable, the design and implementation of *muigen* were mechanical exercises and proceeded rapidly. Implementation was speeded and simplified by the use of tools such as *lex* and *yacc*, and the presence of an event-based target window system that reduced the amount of programming. The IFs that fill *muigen*'s tables, also simplified coding.

Relation to Xmupe2

A natural question to ask is whether MUISL can specify *xmupe2* and *muigen*, be used to generate it. The answer is a qualified *yes*. Using MUISL and *muigen* would have reduced the amount and effort of coding parts of *xmupe2*. The windows, menus, event handlers, and callbacks can be specified and generated. For example, event handlers for keyboard or mouse events, can be written in MUISL. The complex window structures of fragments and other windows is not difficult to specify with MUISL, nor is the display of text and highlighting of certain portions. Examples in Appendix E attest to the ability to define and interact with complex window structures. However, other portions of *xmupe2* which directly interact with the computational component, draw graphics, or are written in Modula-2, cannot be currently generated. The capability to draw graphics can be added as an extension to MUISL. The Window List of *xmupe2* cannot and could not be written in MUISL. Calling Modula-2 code is possible in MUISL, by using the external statement, but interacting with Modula-2 code and its data structures presents a problem, and cannot be supported by MUISL. Issues such as traversal of the unparsed buffers, and computational component menu structures are best written in a conventional programming language since they interface directly with MUPE-2 internals. As previously mentioned, about half of *xmupe2*'s approximately 14,000 lines of code, interface with the computational component: this half could not have been generated.

Chapter 9

Conclusions

This thesis has dealt with two key issues: (a) a user interface for the MUPE-2 programming environment, and (b) generating similarly styled-user interfaces, not necessarily limited to programming environments, from a new specification language.

The first issue deals with *xmupe2*, a user interface for MUPE-2 in its current state. This environment introduces a number of novel features which have been reflected in its user interface. *Xmupe2* successfully reflects the internal state of MUPE-2 at all times and supports: both programming-in-the-small and large; the creation, location, display, and manipulation of multiple fragments; the unparsing of both text and graphics; the display and updating of structured cursors; structured and semi-structured cursor movements; the management and firing of computational component editing commands; interaction with user events and communicating the required ones to the computational component; and the update context-sensitive mouse-based menus.

A user-friendly program, *xmupe2*'s implementation on a bit-mapped screen and its use of menus and the mouse both allow the user to easily and quickly use MUPE-2's commands. Interaction methods such as windows, buttons, and menus offer an intuitive, yet powerful, method of communication with the user. Help is offered at all junctures, and some operations can be aborted — when possible. All of *xmupe2*'s user-directed messages are displayed in one window, and an old message is erased before showing the new one. Windows and menus have clear titles and the latter have understandable item names. Mouse-based pop-up and pull-down menus save screen space and permit a quick traversal and selection of an item. User feedback is accomplished by changing the mouse cursor's shape: the shape changes for different locations of the mouse cursor and for certain internal system-activities.

Finally, windows can easily be manipulated on the screen and be iconized or resized to save space.

Using the widely-available X Window System enhances the portability and flexibility of *xmupe2*. The modular architecture of *xmupe2* isolates its window-system dependent code from other code and facilitates modification of the program. Much care was taken to keep a clear interface between the C and Modula-2 portions of the user interface. Modula-2 was chosen for part of *xmupe2* in order to interact with MUPE-2's computational component. C was chosen for interaction with the window system because it simplified implementation and obviated the need for an intricate window-system - Modula-2 interface.

Xmupe2 is extensible: it is not difficult to add support for new commands implemented by the computational component. For example, *xmupe2*'s menus are created by general purpose routines that simply create a menu based on the parameters passed to them. Another example is the method *xmupe2* interacts with MUPE-2 commands: it first calls the appropriate computational component routine and then uses the same steps to display the results of a command. These steps are to: update the editing menu, map the unparsed buffer, and display the structured cursor.

Requiring considerable effort to design and code, the handcrafted *xmupe2* took about seven months to design and implement and resulted in approximately 14,000 lines of code. The continual problem that faced *xmupe2* was ensuring its correct interaction with the computational component — itself a large and complex program.

Xmupe2 has clearly achieved the goals set forth in this thesis. It is a workable, user friendly user interface for a programming environment. *Xmupe2* also satisfies the requirements enumerated in Section 3.1. Especially important is that *xmupe2* simplifies the manipulations of fragments.

As seen with *xmupe2*, a user interface is often a significant part of an application's code. Moreover, handcrafting a user interface, such as *xmupe2*, is often a complex, tedious, and time consuming process. Given that user interfaces are often developed and tested by prototyping, many changes usually have to be made to the first implementation of a user interface. As a result, the second issue with which the thesis deals is the generation of user interfaces from a specification language. This thesis shows that it is possible to: design MUISL, a simple experimental language for the specification of user interfaces; build a tool, *muigen*, to generate code from this specification; and use the language and generator to facilitate the development of sample user interfaces.

MUISL is an event-based language that views a user interface as a set of objects. The language uses an object class hierarchy, with each class containing attributes and operations which subclasses can inherit. Each object is uniquely identified by its name and class. The MUISL specifier can override default attributes for that class in an attributes section, and also include variable definitions and a statement sequence applicable to each object definition. One type of statement allows the specification of the contents of callbacks or event handlers applicable to each object. An object used in a statement sequence must be instantiated as a child of another previously instantiated object. Instantiation allows the creation of object hierarchies and the single definition of a multiply instantiated object.

To test the viability of MUISL as a specification language, the author designed and implemented *mugen*, a table-driven MUISL-based generator of user-interface code. Consisting of about 8000 lines of C code implemented in 2.5 months, *mugen* uses C as the target programming language, and the X Window System as the target window system. A flexible feature of *mugen* is the dynamic initialization of its tables from initialization files. These files contain information on class names and their hierarchy, attribute names and values, operations, and code templates, among others. Using initialization files allows the alteration of target programming language, target window system, class names, and so on, without having to modify *mugen*.

MUISL and *mugen* have allowed the author to quickly define, execute, and modify sample user interfaces, such as those of Appendix E. The equivalent process without specification language and generator would have taken much longer and required significantly more coding. It is clear that generation, when possible, is much easier than handcrafting. For example, sample interfaces that the author specified with MUISL and generated with *mugen*, took fewer than twenty minutes each. Generated files were also typically four to six times the size of the specification files. The MUISL specifications were easier to design, test, and modify than equivalent handcrafted ones which would have taken a programmer more than an hour each to code. Moreover, handcrafting the sample user interfaces would have required proficiency in the X Window System, a task considerably more difficult than mastering the relatively simple MUISL.

The work carried out in this thesis can be expanded in a number of ways. The MUPE-2 computational component has yet to be finished and *xmupe2* should be upgraded to support future features of MUPE-2. For example, support for an incremental compiler can be easily added to *xmupe2*. MUISL is currently a textual language that is suitable for a programmer,

not a user. A window-based interface to MUISL, such as an interface editor, could be built in order to allow a more user-friendly method of writing a MUISL specification. The language does not support the specification of graphics in user interfaces, among other interaction methods, and should be altered to do so. Finally, MUISL and *mugen* could be used as the nuclei of a graphical UIMS providing a yet more powerful and intuitive method of user-interface specification.

Appendix A

Xmupe2 Architecture

The relation of the user interface to the rest of MUPE-2 is shown in Figure A.1.

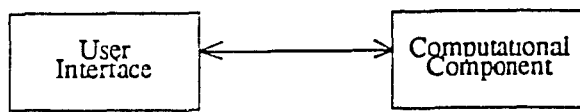
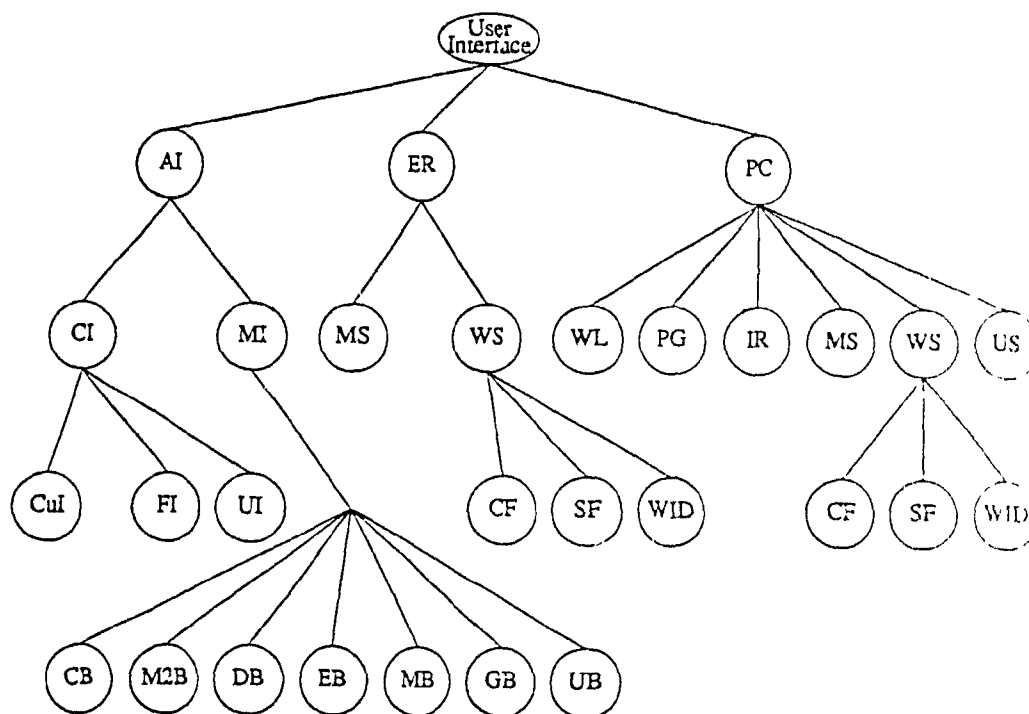


Figure A.1: The User Interface and Computational Component

Xmupe2 has the modular decomposition shown in Figure A.2. The rest of MUPE 2's architecture is not shown because it is not relevant to this thesis. The communication between *xmupe2* and the computational component (shown in Figure A.1) occurs only through one layer of *xmupe2*'s modules.

Except for the Modula-2 Interface, all code is written in C. The three main layers of *xmupe2* are:

- The *Application Interface* (AI) acts as the intermediary between *xmupe2* and the computational component which is written in Modula-2.
 - The *C Interface* (CI) contains the C language interface to: manage the display of cursors (CuI), drive the creation of fragments (FI), and interact with the unparsed buffer (UI).
 - The *Modula-2 Interface* (MI) is the only layer that directly interacts with the computational component. This layer contains code that: retrieves the structured cursor's coordinates and interacts with the movements of the structured



Node Abbreviations and Meanings			
AI	Application Interface	ER	Event Response
PC	Presentation Component	CI	C Interface
MI	Modula-2 Interface	MS	Menu System
WS	Window System	WL	Window List
PG	PIL Graphics	IR	Initializer
US	Utilities System	CuI	Cursor Interface
FI	Fragments Interface	UI	Unparser Interface
CB	Cursor Buffer Layer	M2B	Modula-2 Buffer Layer
DB	Driver Buffer Layer	EB	EditOps Buffer Layer
MB	Menu Buffer Layer	GB	General Manager Buffer Layer
UB	Unparser Buffer Layer	CF	Complex Fragments
SF	Simple Fragments	WID	Widgets

Figure A.2: Xmupe2 Modular Decomposition

cursor (CB), defines the C-Modula-2 interface (M2B), contains the main Modula-2 program module (DB), interacts with editing commands (EB), interacts with internal editing menus (MB), communicates with the General Manager (GB) — a manager of internal fragment structures, and retrieves information from the unparser (UB).

- The *Event Response* (ER) system acts as the dialogue control component, responding to user events and calling other *xmupe2* code. Code in this layer directly interfaces with the X Window System.
 - The *Menu System* (MS) contains event handlers and callbacks for menus.
 - The *Window System* (WS) contains event handlers and callbacks for windows representing PIL fragments (CF), PIS fragments (SF), and other non-fragment windows called widgets (WID).
- The *Presentation Component* (PC) system performs the role of initializing and displaying windows and menus. Some code in this layer directly interfaces with the X Window System.
 - The *Window List* (WL) system manages the Window List, which contains information on windows representing fragments and PIL nodes.
 - The *PIL Graphics* (PG) system creates and manages the graphics data structures displaying the PIL node hierarchy in a PIL Graphics Window.
 - The *Initializer* (IR) initializes *xmupe2*'s variables and data structures and calls the X Toolkit's main interaction loop. This loop is responsible for the management and dispatching of events to *xmupe2*'s event handlers and callbacks.
 - The *Menu System* (MS) contains code to create menus.
 - The *Window System* (WS) contains code to create windows.
 - The *Utilities System* (US) contains various utilities used by *xmupe2*.

Appendix B

MUISL Lexical Rules and Grammar

Notation: (...) indicates contents)

<code>::=</code>	=	is composed of
<code>x</code>	=	<code>x</code> is a terminal
<code>x</code>	=	<code>x</code> is a nonterminal
<code>[x]</code>	=	<code>x</code> is optional
<code>{x}</code>	=	0 or more occurrences of <code>x</code>
<code>{x}+</code>	=	1 or more occurrences of <code>x</code>
<code>(...)</code>	=	group the contents
<code>/.../</code>	=	match any one character in list
<code>*</code>	=	match 0 or more occurrences of the preceding
<code>+</code>	=	match 1 or more occurrences of the preceding
<code>.</code>	=	match any character
<code>\</code>	=	do not match the character(s) following
<code>\n</code>	=	newline character
<code>-</code>	=	specifies a range

B.1 MUISL Lexical Rules

comment ::= `#.*\n`

digit ::= `0-9`

letter ::= `(a-z | A-Z)`

character ::= .

B.2 MUISL Grammar

```

interface_definition ::= [ variables ] { object_definition }+ initialization_block
initialization_block ::= INIT [ variables ] [ ACTIONS : { simple_statement } ] END
object_definition ::=
    OBJECT object_name class [ superclass ] [ attributes ] [ variables ] [ actions ] END
object_name ::= NAME : ident
ident ::= letter { letter | digit | - }
class ::= CLASS : class_name
class_name ::= clident
superclass ::= SUPERCLASS : class_name
attributes ::= ATTRIBUTES : { attribute_definition }
attribute_definition ::=
    regular_attribute_definition
    | callback_attribute_definition
    | event_handler_attribute_definition
regular_attribute_definition ::=
    external
    | attribute_name = ( attribute_value | value | string )
callback_attribute_definition ::= attribute_callback_name = ( ident_list )
event_handler_attribute_definition ::= attribute_event_name = ( event_tuple_list )
attribute_name ::= atident
attribute_value ::= ATident
value ::= ident | number
number ::= { digit }+ [ . { digit }+ ] [ /Ee/ [ + | - ] { digit }+ ]
string ::= " { character \ " } "
attribute_callback_name ::= atident/Cc/allback[s]
ident_list ::= ident { , ident }
attribute_event_name ::= atident/Ee/vent/Hh/andler[s]
event_tuple_list ::= event_tuple { , event_tuple }

```

```

event_tuple ::= ( event_name , ident )
event_name ::= eventident
variables ::= VARIABLES : decl_block
decl_block ::= { type ident_list | external }
type ::=
    INTEGER | REAL | CHAR | CARDINAL | BOOLEAN | STRING
    | OBJECT_ID | BUTTON_ID | KEYCODE | DIMENSION | POSITION
external ::= @.*
actions ::= ACTIONS : { statement }
statement ::= simple_statement | procedure_statement
simple_statement ::= external | operation | assignment | conditional
operation ::= [ operation_name argument_list ]
operation_name ::= objident
argument_list ::= { argument_name : argument_value } +
argument_name ::= argident
argument_value ::= value | string | attribute_name | attribute_value
assignment ::= ident = ( operation | expression )
expression ::= simple_expression [ relation simple_expression ]
relation ::= < | > | ≤ | ≥ | == | <>
simple_expression ::= [-] term { add_operator term }
add_operator ::= + | - | ||
term ::= factor { mul_operator factor }
mul_operator ::= * | / | &&
factor ::= value | ! factor | ( expression )
conditional ::=
    IF expression THEN { simple_statement } ELSE { simple_statement } END
procedure_statement ::= callback | event_handler
callback ::=
    CALLBACK ( attribute_callback_name , ident ) { simple_statement } END
event_handler ::= EVENT ( event_name , ident ) { event_statement } END
event_statement ::= key_statement | button_statement | simple_statement
key_statement ::= CASE KEY OF { key_case_element } + [ else_part ] END
key_case_element ::= keycode : { simple_statement }

```

keycode ::= **key***ident*

else_part ::= **ELSE** { *simple_statement* }

button_statement ::=

CASE BUTTON OF { *button_case_element* }+ [*else_part*] **END**

button_case_element ::= *button_name* : { *simple_statement* }

button_name ::= **button***ident*

Appendix C

Muigen Architecture and File Generation

This appendix first discusses *muigen*'s architecture, then gives a brief overview of the steps *muigen* uses to produce a GF.

C.1 Muigen Architecture

Muigen has a modular architecture which greatly simplified and speeded implementation, testing, and maintenance of the program. For example, the modular architecture eases *muigen*'s expansion and enhancements. Definitions used later include:

- The *Mapping Table* (MT) is a table of pointers to other tables, each of which either contains code templates (the Template Table), or maps most MUISL names to target programming language and window system names, the MT and tables to which it points are initialized from the contents of the IFs.
- The *Class Table* (CT) is a table that stores information about each class — its MUISL name, mapping, attributes, operations, and superclass.
- An *Object-Definition Structure* (ODS) is a data structure storing information specific to an object definition; information includes the object name, object class, superclass, local variables, attributes, and actions (including the code of local callbacks and event handlers).

- The *Object-Definition Structure List* (ODSL) is the list of ODSs from which code for each defined object is generated.

Muigen's modular decomposition is graphically shown in Figure C.1. The modules perform the following functions:

- The *Driver* is the main module that calls other modules.
- The *Initializer* reads and parses IFs, initializes the MT and CT from the IFs, initializes other *muigen* data structures, and performs other initializations.
- The *Checker* performs lexical analysis, parsing, and semantic checking of the SF. This module uses *lex* and *yacc*.
- The *Builder* accepts tokens from the Checker and builds the ODSL.
- The *Generator* creates the GF from the ODSL and MT.

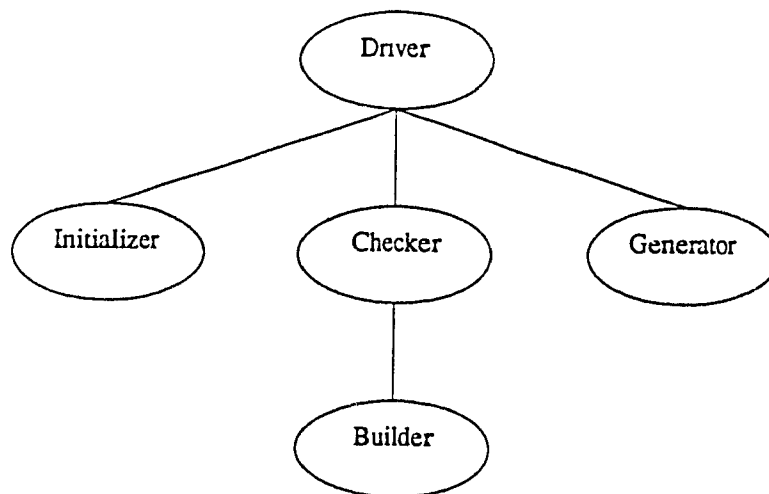


Figure C.1: *Muigen* Modular Decomposition

The *Driver* first calls the *Initializer* which aborts *muigen* if there is any error in the IFs. Information passed from the *Initializer* to the *Driver* includes the CT and MT. These tables are used by the *Checker* as it reads and parses the SF. While parsing, the *Checker* calls the *Builder* at the appropriate tokens. The *Builder* creates an ODS for each object definition and fills it with information gleaned from an object definition, with the assistance of the data in the MT and CT. If there are no errors detected by the *Checker*, the *Driver* then

calls the Generator, and passes to it the ODSL created by the Builder. The Generator uses the ODSL and code templates in the Template Table to create the GF.

C.2 File Generation

When the Generator creates the GF, its general algorithm is:

```
Generate header (include-statements, macros, non-MUISL globals, etc.)
Generate global procedures
Generate global variables
For each ODS in the ODSL
    Generate event handlers
    Generate callbacks
    Generate object definition procedure
Generate instantiation procedure
Generate main()
```

Generation of the object definition procedure, which will contain the attributes, actions, and non-procedure statements of an MUISL object definition, uses the information from the current ODS to do the following:

```
Generate procedure header
Generate local variables
Generate regular attributes
Generate object creation procedure
Generate callback attributes
Generate event handler attributes
Generate actions
Generate procedure trailer
```

Generation of the event handlers, callbacks, and the instantiation procedure is essentially similar (except for the different procedure headers), and follows these steps:

```
Generate procedure header
Generate local variables
Generate actions
Generate procedure trailer
```


Appendix D

Initialization Files

This appendix describes the contents of Initialization Files (IFs) read by *muigen* to initialize its tables. The IFs contain mappings to the current target programming language and window system, C and the X Window System, respectively. The purpose of this appendix is not to show the mappings, but to explain the names that a MUISL specifier can use. For identification purposes, IFs contain an *.ini* suffix. Prefixes have been chosen to indicate a non-code mapping (*mapping*), or non-mapping initialization of a table (*init*), or code template mapping (*template*). Of the IFs shown below, all but *mappingOperation.ini* are NOIFs. The contents of the IFs whose names are followed by an asterisk (*) are explained later in this appendix:

initClassTable.ini Contains class names used in initializing the CT data structure. The class names in this file are those shown in Figure 8.2.

initMappingTable.ini Initializes the MT data structure, which points to tables to be filled by the rest of the IFs. Entries in this file include names to point to tables of variable types, attribute names, attribute values, key mappings, mouse button mappings, event names, code templates, and so on.

mappingAttributeName.ini * Maps MUISL attribute names to X Window System attribute names.

mappingAttributeValue.ini * Maps MUISL attribute values to X Window System attribute values.

mappingButtonNames.ini * Maps MUISL button names to X Window System mouse button names.

mappingKeyNames.ini * Maps MUISL key names to X Window System key names.

mappingClass.ini Maps MUISL class names, already initialized in the CT from *init-ClassTable.ini*, to X Window System names.

mappingEventMasks.ini Maps internal *muigen* event-handler masks to X Window System event mask names.

mappingEventNames.ini * Maps MUISL event names to X Window System event names

mappingOperation.ini * Contains MUISL operation mappings to C/X Window System routines.

mappingSuperclass.ini Contains class hierarchy of MUISL object-types.

mappingVariable.ini Maps MUISL variable names to C/X Window System names

templateCode.ini * Maps internal *muigen* names to C/X Window System code-templates.

The rest of this appendix first explains the contents of the IFs previously marked with an asterisk (*). Table D.1 gives the prefixes to be appended to the attribute names of the corresponding attribute-name tables. For example, the prefix *atRoot* is to be appended to the names of class *clRoot*'s attributes, listed in Table D.2. The resulting attribute names would be *atRootBackground*, *atRootBorderWidth*, and so on. Tables D.2, D.3, D.4, D.5, D.6, D.7, D.8, D.9, D.10, D.11, D.12, D.13, D.14, D.15, D.16, and D.17 describe the attribute names that can be used on the left side of an attribute definition. Table D.18 describes the attribute values that can be used on the right side of an attribute definition. Table D.19 lists legal mouse names, and Table D.20 gives a partial list of legal key names (the full list is too long to enumerate here). In the latter table, an ellipsis indicates further elements. Event names are shown in Table D.21.

Notation used includes: capitalized type names that indicate a value or variable of that type, non-italicized terminals, and italicized nonterminals. The initials TWS denote *target window system*; an asterisk (*) matches any character. Other notation is similar to that of Appendix B.

Class	Attribute-Name Prefix
clRoot	atRoot
clButton	atButton
clMenu	atMenu
clSimpleWindow	atSimpleWindow
clToggleButton	atToggleButton
clMenuButton	atMenuButton
clSimpleMenu	atSimpleMenu
clListMenu	atListMenu
clTextWindow	atTextWindow
clScrollbarWindow	atScrollbarWindow
clBoxWindow	atBoxWindow
clPanedBoxWindow	atPanedBoxWindow
clFormBoxWindow	atFormBoxWindow
clViewportWindow	atViewportWindow
clDialogueWindow	atDialogueWindow
clItemSimpleMenu	atItemSimpleMenu

Table D.1: Attribute-Name Prefixes

Name	Description	Values	Default
Background	Background color	ATblack, ATwhite	ATwhite
BorderWidth	Border width	DIMENSION	1
Callback	Callback(s)	(<i>ident_list</i>)	NULL
DestroyCallback	Destruction callback(s)	(<i>ident_list</i>)	NULL
EventHandlers	Event handler(s)	(<i>Event_tuple_list</i>)	NULL
Height	Height	DIMENSION	0
Width	Width	DIMENSION	0
MapWhenInstantiated	Display when instantiated?	ATyes, ATno	ATyes
IsSensitive	Receive events?	ATyes, ATno	ATyes
PositionX	Parent-relative x coordinate	POSITION	0
PositionY	Parent-relative y coordinate	POSITION	0
Label	Label to display	STRING	NULL

Table D.2: clRoot Attributes

Name	Description	Values	Default
Callback	Button-press callback	(<i>ident_list</i>)	NULL
Cursor	Mouse-cursor shape	AT*Cursor	TWS
Font	Text font	AT*Font	TWS
Foreground	Foreground color	ATwhite, ATblack	ATblack
JustifyLabel	Label's alignment	ATjustifyLeft, ATjustifyCenter, ATjustifyRight	ATjustifyCenter

Table D.3: clButton Attributes

Name	Description	Values	Default
Cursor	Mouse-cursor shape	AT*Cursor	TWS
Foreground	Foreground color	ATwhite, ATblack	ATblack

Table D.4: clMenu Attributes

Name	Description	Values	Default
Cursor	Mouse-cursor shape	AT*Cursor	TWS
Foreground	Foreground color	ATwhite, ATblack	ATblack

Table D.5: clSimpleWindow Attributes

Name	Description	Values	Default
RadioGroup	Toggle button in radio group	<i>ident</i>	NULL
ButtonState	Set button?	ATyes, ATno	ATyes

Table D.6: clToggleButton Attributes

Name	Description	Values	Default
SimpleMenuName	clSimpleMenu to pop up	<i>string</i>	TWS

Table D.7: clMenuButton Attributes

Name	Description	Values	Default
PopdownCallback	Popdown-callback(s)	(<i>ident_lst</i>)	NULL
PopupCallback	Popup-callback(s)	(<i>ident_lst</i>)	NULL
PopUpItemOnEntry	Default menu item	(<i>ident</i>)	TWS

Table D.8: clSimpleMenu Attributes

Name	Description	Values	Default
Callback	clListMenu item-callback(s)	(<i>ident_lst</i>)	NULL
Font	Text font	AT*Font*	TWS
DefaultColumns	Menu column number	CARDINAL	2
ForceColumns	Force columns?	ATyes, ATno	ATno
ItemStrings	Menu items	STRING	NULL
NumberItems	Menu item number	CARDINAL	0

Table D.9: clListMenu Attributes

Name	Description	Values	Default
BreakLine	Break line?	ATyes, ATno	ATno
UpperDisplayPosition	Character position at top-left corner	POSITION	0
DisplayNonPrintables	Display nonprintables?	ATyes, ATno	ATyes
EchoChars	Echo characters?	ATyes, ATno	ATyes
Update	Text update-type	ATreadOnly, ATappendOnly, ATeditable	ATreadOnly
Font	Text font	AT*Font*	TWS
ScrollHoriz	Horizontal scrollbar?	ATscrollAlways, ATscrollNever, ATscrollWhenNeeded	ATscrollNever
ScrollVert	Vertical scrollbar?	ATscrollAlways, ATscrollNever, ATscrollWhenNeeded	ATscrollNever
FromString	String to display	<i>string</i>	NULL
FromFile	File to display	<i>string</i>	NULL
Type	Text from string/file?	ATstringText, ATfileText	ATstringText

Table D.10: clTextWindow Attributes

Name	Description	Values	Default
JumpCallback	Scroll jump-callback(s)	(ident_list)	NULL
WindowLength	Vertical scrollbar height/ Horizontal scrollbar length	DIMENSION	1
MinThumbSize	Minimum thumb pixel-size	DIMENSION	7
Orientation	Scrollbar orientation	ATvertical, AThorizontal	ATvertical
DownCursor	Vertical backward-scrolling cursor	AT*Cursor	ATdownArrowCursor
UpCursor	Vertical forward-scrolling/ horizontal thumbing cursor	AT*Cursor	ATupArrowCursor
RightCursor	Horizontal backward-scrolling/ vertical thumbing cursor	AT*Cursor	ATrightArrowCursor
LeftCursor	Horizontal forward-scrolling	AT*Cursor	ATleftArrowCursor
VertCursor	Vertical inactive cursor	AT*Cursor	ATvertDoubleArrow Cursor
HorizCursor	Horizontal inactive cursor	AT*Cursor	AThorizDoubleArrow Cursor
ScrollCallback	Scrolling callback(s)	(ident_list)	NULL
ShownThumbSize	Percentage thumb-size	REAL	0 0
Thickness	Vertical width/ Horizontal height	DIMENSION	14
ThumbTop	Percentage thumb-top location	REAL	0 0

Table D.11: clScrollbarWindow Attributes

Name	Description	Values	Default
HorizSpaceBetweenChildren	Horizontal pixel-space between children	DIMENSION	4
VertSpaceBetweenChildren	Vertical pixel-space between children	DIMENSION	4
Shape	Box shape	ATtallAndNarrow, ATshortAndWide	ATtallAndNarrow

Table D.12: clBoxWindow Attributes

Name	Description	Values	Default
Cursor	Mouse-cursor shape	AT*Cursor	TWS
Orientation	Pane-stacking orientation	ATvertical, AThorizontal	ATvertical
MaxSize	Maximum child size	DIMENSION	Infinity
MinSize	Minimum child size	DIMENSION	Grip size
AllowResize	Allow child resizing?	ATyes, ATno	ATyes
ShowGripBetweenPanels	Show grip between panes?	ATyes, ATno	ATno

Table D.13: clPannedBoxWindow Attributes

Name	Description	Values	Default
DefaultDistance	Default inter-children spacing	CARDINAL	4
LeftNeighborObject	Left-neighbor	OBJECT.ID	NULL
TopNeighborObject	Top-neighbor	OBJECT.ID	NULL
HorizDistance	Horizontal inter-children spacing	CARDINAL	TWS
VertDistance	Vertical inter-children spacing	CARDINAL	TWS

Table D.14: clFormBoxWindow Attributes

Name	Description	Values	Default
AllowHorizScrollbar	Allow horizontal scrollbar when needed?	ATyes, ATno	ATno
AllowVertScrollbar	Allow vertical scrollbar when needed?	ATyes, ATno	ATno
ForceScrollbars	Force allowed scrollbars?	ATyes, ATno	ATno
UseBottomEdge	Place horizontal scrollbar on bottom edge?	ATyes, ATno	ATno
UseRightEdge	Place vertical scrollbar on top edge?	ATyes, ATno	ATno

Table D.15: clViewportWindow Attributes

Name	Description	Values	Default
Text	Input text	STRING	NULL
DefaultDistance	Default inter-children spacing	CARDINAL	1
LeftNeighborObject	Left-neighbor	OBJECT_ID	NULL
TopNeighborObject	Top-neighbor	OBJECT_ID	NULL
HorizDistance	Horizontal inter-children spacing	CARDINAL	TWS
VertDistance	Vertical inter-children spacing	CARDINAL	TWS

Table D.16: clDialogueWindow Attributes

Name	Description	Values	Default
Callback	Menu item callback(s)	(int_list)	NULL

Table D.17: clItemSimpleMenu Attributes

When necessary, target programming language and window system code templates in *templateCode.ini* contain %s characters that are replaced with the appropriate names, such as object names, mapped class names, and so on. This file contains code templates for information such as include files, global definitions, procedure templates, and so on. Some examples of entries in this file are.

```
{ templateEventHandlerHeader,
  #%s is filled with specifier's event handler name
  'void %s(widget, client_data, event)
    caddr_t client_data;\n XEvent *event;\n { ' }

{ templateAssignment, ' %s = %s ;\n'}

{ templateExternal, '%s'}

{ templateInstantiateProcedureHeader, '\n void Instantiate()\n{\n'}
```

Operation names and arguments specified in *mappingOperation.ini* are summarized below. The notation used for operation names is: *objClassOperationName*. The notation *→ type* indicates a return type associated with an operation. The argument *argObject* is the receiver of an operation — the object to which the operation applies; except for the

Name	Description
ATwhite	White color
ATblack	Black color
ATyes	True
ATno	False
ATreadOnly	Readonly text
ATappendOnly	Appendonly text
ATeditable	Editable text
ATscrollNever	No scrollbar
ATscrollAlways	Always a scrollbar
ATscrollWhenNeeded	Scrollbar, if needed
ATstringText	Displayed string
ATfileText	Displayed file
ATvertical	Vertical scrollbar
AThorizontal	Horizontal scrollbar
ATsmallFont	6x13 font
ATsmallFontBold	6x13 bold font
ATmediumFont	8x13 font
ATmediumFontBold	8x13 bold font
ATlargeFont	9x15 font
ATlargeFontBold	9x15 bold font
ATupArrowCursor	↑ cursor
ATdownArrowCursor	↓ cursor
ATrightArrowCursor	⇒ cursor
ATleftArrowCursor	⇐ cursor
AThorizDoubleArrowCursor	⇔ cursor
ATvertDoubleArrowCursor	↕ cursor
ATdotCursor	• cursor
ATcircleCursor	○ cursor
ATcrosshairCursor	+ cursor
ATupDownArrowCursor	↑↓ cursor
ATneArrowCursor	↗ cursor
ATnwArrowCursor	↖ cursor
ATwatchCursor	Watch-shaped cursor
ATquestionCursor	? cursor
ATjustifyLeft	Left-justified label
ATjustifyCenter	Center-justified label
ATjustifyRight	Right-justified label
ATtallAndNarrow	Tall & narrow clBoxWindow
ATshortAndWide	Short & wide clBoxWindow
ATnoPopup	One-time popup
ATmenuPopup	Menu popup/popdown
ATdialoguePopup	Dialogue-box popup/popdown
ATobjectPopup	Non-menu/dialogue popup/popdown
ATtopObject	Top parent object

Table D.18: Attribute Values

Name	Description
buttonLeft	Left mouse button
buttonMiddle	Middle mouse button
buttonRight	Right mouse button

Table D.19: Mouse Buttons

Name	Description
keya	a key
keyA	A key
.	
keyz	z key
keyZ	Z key
keyl	l key
.	

Table D.20: Keys

Name	Event
eventKeyPress	Key pressed
eventKeyRelease	Key released
eventButtonPress	Button pressed
eventMotion	Mouse moved
eventEnter	Object entered
eventLeave	Object exited
eventExpose	Object exposed
eventVisible	Object visible
eventCreate	Object created
eventDestroy	Object destroyed
eventUnmap	Object undisplayed
eventMap	Object displayed
eventConfigure	Object manipulated
eventResize	Object resized
eventCirculate	Object hidden/unhidden

Table D.21: Event Names

operation *objRootInstantiate*, objects are assumed to have been instantiated. Recall that an operation's arguments can be specified in any order.

Operations associated with class *clRoot* are:

- [*objRootInstantiate* *argObject:ident* *argParent:ident*
argPopupType: (*ATnoPopup* | *ATmenuPopup* | *ATobjectPopup* | *ATdialoguePopup*)]
Purpose. Instantiate a previously defined object.
- [*objRootDestroy* *argObject:ident*]
Purpose: Destroy an *ATnoPopup* object. Will exit the program if destroyed object is the parent of all others.
- [*objRootDestroyPopup* *argObject:ident*]
Purpose: Destroy an *ATmenuPopup*, *ATobjectPopup*, or *ATdialoguePopup* object.
- [*objRootDestroyExit* *argObject:ident*]
Purpose: Destroy an object and exit the program.
- [*objRootSetAttribute* *argObject:ident* *argAttributeName:attribute.name*
argAttributeValue: (*ident* | *string* | *attribute_value*)]
Purpose. Set an object's attribute to a value.
- [*objRootGetAttribute* *argObject:ident* *argAttributeName:attribute.name*
argAttributeValue:ident]
Purpose. Retrieve an object's attribute-value to a variable.
- [*objRootMove* *argObject:ident* *argX:value* *argY:value*]
Purpose: Move an object to an (*x,y*) location.
- [*objRootMap* *argObject:ident*]
Purpose: Display an undisplayed object.
- [*objRootUnMap* *argObject:ident*]
Purpose: Undisplay a displayed object.

Operations associated with class *clToggleButton* are:

- [objToggleButtonAddToGroup argObject:*ident* argToggleButtonInGroup:*ident*]
Purpose: Add a toggle button to the group of toggle buttons identified by the value of *argToggleButtonInGroup*.
- [objToggleButtonRemoveFromGroup argObject:*ident*]
Purpose: Remove the toggle button from its group of toggle buttons

Operations associated with class *clListMenu* are:

- [objListMenuPopUp argObject:*ident*]
Purpose: Pop-up a list menu.
- [objListMenuPopDown argObject:*ident*]
Purpose: Pop-down a list menu.
- [objListMenuHighlightItem argObject:*ident* argItemIndex:CARDINAL]
Purpose: Highlight a list-menu item at a certain index.
- [objListMenuUnhighlightItem argObject:*ident*]
Purpose: Unhighlight the currently highlighted list-menu item
- [objListMenuGetCurrentItemString argObject:*ident*] → STRING
Purpose: Get the string of the highlighted list-menu item.
- [objListMenuGetCurrentItemIndex argObject:*ident*] → CARDINAL
Purpose: Get the index of the list menu item currently highlighted. Indices start at zero.

Operations associated with class *clTextWindow* object are:

- [objTextWindowLoadFromFile argObject:*ident* argFilename:string]
Purpose: Load a file into a text window.
- [objTextWindowSaveToFile argObject:*ident*]
Purpose: Save a text window's displayed text to the file from which it was read
- [objTextWindowSaveToNamedFile argObject:*ident* argFileName string]
Purpose: Save a text window's displayed text to the named file

- [objTextWindowHasTextChanged argObject:*ident*] → BOOLEAN

Purpose: Return TRUE if a text window's contents have changed.

- [objTextWindowReplaceText argObject:*ident* argText:*string*
argStartPosition:CARDINAL argEndPosition:CARDINAL]

Purpose Delete text from the absolute character positions starting from the value for *argStartPosition* to the value for *argEndPosition*. Replace deleted text with new text, given by the value of *argText*.

- [objTextWindowGetText argObject:*ident*] → STRING

Purpose: Get text displayed in a text window.

- [objTextWindowHighlightText argObject:*ident* argStartCharIndex:CARDINAL
argEndCharIndex:CARDINAL]

Purpose: Highlight text between starting and ending absolute character positions, inclusive.

- [objTextWindowUnhighlightText argObject:*ident*]

Purpose: Unhighlight highlighted text.

- [objTextWindowMessage argObject:*ident* argText:*string*]

Purpose: Display text in a text window, erasing any previous messages.

The operation associated with class *clDialogueWindow* is:

- [objDialogueWindowGetText argObject:*ident*] → STRING

Purpose: Get a dialogue window's text.

The number of the above operations, attribute names, and values, may be currently limited, but is well suited to simple experimentation with a specification language. New operations and attributes can be added to the IFs, without having to recompile *muigen*.

Appendix E

Sample MUISL Specifications

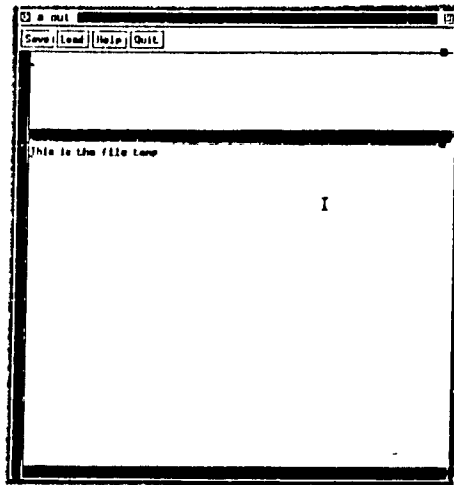
This appendix describes sample MUISL specifications, and shows the user interfaces that *muigen* generated from these specifications. The generated files are not shown because of space limitations.

E.1 Example 1

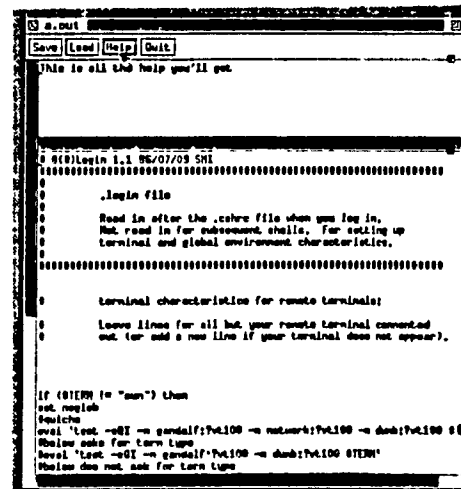
The first example specifies a single window having multiple children. The resultant interface is shown in Figure E.1. It consists of a window with buttons and two text windows. The upper text window is for messages and the user's communication with the program. The lower text window is for text editing. Frame (a) of the figure shows the initial window that appears when the program is invoked. The lower text window contains the contents of a file called *temp*. In Frame (b), the user has typed the name of a file in the upper text window and has clicked the *Load* button to load this file. The lower text window contains the contents of the loaded file. Frame (c) shows the results of clicking the *Help* button: a message appears in the upper text window.

The complex hierarchy of windows includes: an outer paned box surrounding all windows (*OuterPanedBox*) and three box windows (*InnerBox*, *InnerUpperText*, and *InnerLowerText*) as children of the outer paned box. *InnerBox* acts as the container of the command buttons *SaveButton* (for saving the current file), *LoadButton* (for loading a file whose name is typed in *InnerUpperText*), *HelpButton* (for help messages in *InnerUpperText*), and *QuitButton* (for quitting).

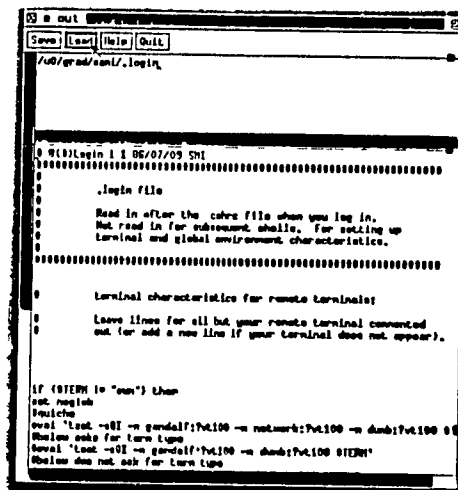
The specification of Example 1 first defines each object with its attributes and actions



(a)



(c)



(b)

Figure E.1: Example 1 Interface

Objects that need to override the default classing include the SUPERCLASS token in their definition. The specification instantiates the objects in the correct order to pop up the window. This example also shows the usage of callbacks and operations:

```
# Description: Shows clPanedBoxWindow, clBoxWindow, clTextWindow,
#               clCommandButton
```

```
#Define the outermost paned box
```

```
OBJECT
```

```
NAME: OuterPanedBox
```

```
CLASS: clPanedBoxWindow
```

```
ATTRIBUTES:
```

```
    atRootHeight = 500
```

```
    atRootWidth = 500
```

```
END #OuterPanedBox
```

```
#Define the children of OuterPanedBoxWindow
```

```
#Define the box containing buttons
```

```
OBJECT
```

```
NAME: InnerBox
```

```
CLASS: clBoxWindow
```

```
ATTRIBUTES:
```

```
    atRootHeight = 25
```

```
END #InnerBox
```

```
#Define the text windows
```

```
#Upper Text window for messages
```

```
OBJECT
```

```
NAME: InnerUpperText
```

```
CLASS: clTextWindow
```

```
ATTRIBUTES:
```

```
    atRootHeight = 100
```

```
    atTextWindowUpdate = ATeditable
```

```
    atTextWindowScrollHoriz = ATscrollAlways
```

```
    atTextWindowScrollVert = ATscrollAlways
```

```
END #InnerUpperText
```

```
OBJECT
```

```
NAME: InnerLowerText
```

```
CLASS: clTextWindow
```

```
ATTRIBUTES:
```

```
    atRootHeight = 375
```

```
    atTextWindowUpdate = ATeditable
```

```
    atTextWindowScrollHoriz = ATscrollAlways
```

```
    atTextWindowScrollVert = ATscrollAlways
```

```
    atTextWindowType = ATfileText
```

```
    atTextWindowFromFile = "temp"
```


END #InnerLowerText

#Define children of InnerBox

OBJECT

NAME. SaveButton

CLASS clCommandButton

SUPERCLASS. clTextWindow

ATTRIBUTES

atRootCallback = (callback_save) #not needed if CALLBACK statement
#is below

atRootLabel = "Save"

VARIABLES

STRING str

ACTIONS

CALLBACK (atRootCallback, callback_save)

#atRootCallback is used because default superclass is overridden

[objTextWindowSaveToCurrentFile argObject: InnerLowerText]

END #callback

END #SaveButton

OBJECT

NAME. LoadButton

CLASS: clCommandButton

SUPERCLASS. clTextWindow

ATTRIBUTES.

atRootLabel = "Load"

#no need for callback attribute if CALLBACK statement is below

VARIABLES

STRING str

ACTIONS

CALLBACK (atRootCallback, callback_load)

str = [objTextWindowGetText argObject: InnerUpperText]

[objTextWindowLoadFromFile argObject: InnerLowerText
argFileName: str]

END #callback

END #LoadButton

OBJECT

NAME HelpButton

CLASS. clCommandButton

SUPERCLASS clTextWindow #to allow usage of objTextWindow messages

ATTRIBUTES.

atRootLabel = "Help"

#no need for callback attribute if CALLBACK statement is below

ACTIONS.

CALLBACK (atRootCallback, callback_help)

#atRootCallback is used because default superclass is overridden

```

        [ objTextWindowMessage argObject: InnerUpperText
          argText: "This is all the help you'll get\n" ]
      END #callback
    END #HelpButton

OBJECT
NAME: QuitButton
CLASS: clCommandButton
ATTRIBUTES:
  atRootLabel = "Quit"
  #no need for callback attribute if CALLBACK statement is below
ACTIONS:
  CALLBACK ( atButtonCallback, callback_quit )
    [ objRootDestroyExit argObject: ATtopObject ]
  END #callback
END #QuitButton

INIT
ACTIONS:
  [ objRootInstantiate argObject: OuterPanedBox argParent: ATtopObject
    argPopupType: ATnoPopup ]

  [ objRootInstantiate argObject: InnerBox argParent: OuterPanedBox
    argPopupType: ATnoPopup ]

  [ objRootInstantiate argObject: InnerUpperText argParent: OuterPanedBox
    argPopupType: ATnoPopup ]

  [ objRootInstantiate argObject: InnerLowerText argParent: OuterPanedBox
    argPopupType: ATnoPopup ]

  #children of InnerBox
  [ objRootInstantiate argObject: SaveButton argParent: InnerBox
    argPopupType: ATnoPopup ]

  [ objRootInstantiate argObject: LoadButton argParent: InnerBox
    argPopupType: ATnoPopup ]

  [ objRootInstantiate argObject: HelpButton argParent: InnerBox
    argPopupType: ATnoPopup ]

  [ objRootInstantiate argObject: QuitButton argParent: InnerBox
    argPopupType: ATnoPopup ]
END #INIT

```

E.2 Example 2

The second example shows the ability to instantiate multiple independent windows, at the user's request. Figure E 2 displays the results of the user's creation of multiple windows. It consists partly of a window (labeled with *a.out*) containing command and menu buttons, that is the only one that initially appears. The other windows — text and generic windows, and a dialogue box — were created by clicking the appropriate item from the menu attached to one of the buttons. Also shown is a pull-down menu titled *WindowMenu*.

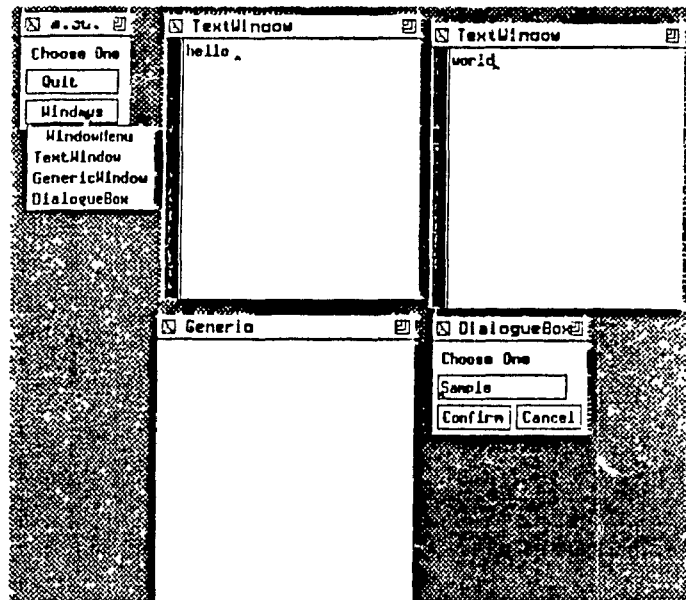


Figure E.2: Example 2 Interface

The window of buttons (*OuterBox*) includes *LabelButton* (for displaying a title), *QuitButton* (for quitting), and *MenuButton* (for pulling down the menu *WindowMenu*). This pull-down menu consists of items to create different windows: *MenuItem1* to create an instance of the text window, *TextWindow*; *MenuItem2* to create an instance of the generic window, *GenericWindow*; and *MenuItem3* to create an instance of the dialogue box, *DialogBox*. The dialogue box contains a text area and two command buttons (*CancelButton* and *ConfirmButton*).

All objects are defined with their attributes and actions, but not all are immediately instantiated. *OuterBox*, its children buttons, and *WindowMenu* are first instantiated. When

a menu item is clicked, the appropriate window (*GenericWindow*, *TextWindow*, or *DialogBox* with its buttons) is instantiated. Note that this example shows the usage of event handlers (see the definition of *GenericWindow*):

```
# Description : Shows clCommandButton, clDialogueWindow,
#               clMenuButton, clLabelButton, clSimpleMenu,
#               clItemSimpleMenu, clGenericWindow, clTextWindow,
#               clBoxWindow
```

```
# Define the outermost box
```

```
OBJECT
NAME: OuterBox
CLASS: clBoxWindow
END #OuterBox
```

```
#Define children of OuterBox
```

```
OBJECT
NAME : LabelButton
CLASS: clLabelButton
ATTRIBUTES:
    atRootLabel = "Choose One"
    atRootBorderWidth = 0
END #LabelButton
```

```
OBJECT
NAME : QuitButton
CLASS: clCommandButton
ATTRIBUTES:
    atRootLabel = " Quit  "
ACTIONS:
    CALLBACK ( atButtonCallback, callback_quit )
              [ objRootDestroyExit argObject: ATtopObject]
    END #callback
END #QuitButton
```

```
OBJECT
NAME : MenuButton
CLASS: clMenuButton
ATTRIBUTES:
    atRootLabel = " Windows "
    atMenuButtonSimpleMenuName = "WindowMenu"
END #MenuButton
```

```
# Define the simple menu
```

```
OBJECT
```

```

NAME: WindowMenu
CLASS: clSimpleMenu
ATTRIBUTES
    atRootLabel = "WindowMenu"
END #WindowMenu

# Objects created from the simple menu

# The text window

OBJECT
NAME TextWindow
CLASS clTextWindow
ATTRIBUTES
    atRootWidth = 200
    atRootHeight = 200
    atTextWindowUpdate = ATeditable
    atTextWindowScrollVert = ATscrollAlways
END #TextWindow

#Generic window
OBJECT
NAME GenericWindow
CLASS clGenericWindow
ATTRIBUTES
    atRootWidth = 200
    atRootHeight = 200
ACTIONS
    EVENT ( eventButtonPress, handle_key_press )
        CASE BUTTON OF
            buttonLeft :
                [ objRootDestroyPopup argObject: GenericWindow ]
        ELSE
            @fprintf(stderr,"Button pressed\n");
        END # case
    END #event

    EVENT ( eventEnter, handle_enter )
        @fprintf(stderr,"Entered window\n");
    END #event
END #GenericWindow

# Define the children of the dialogue box
OBJECT
NAME ConfirmButton
CLASS clCommandButton
SUPERCLASS clDialogueWindow
ATTRIBUTES

```

```

    atRootLabel = "Confirm"
VARIABLES:
    STRING str
ACTIONS:
    CALLBACK ( atRootCallback, callback_confirm )
        str = [ objDialogueWindowGetText argObject: DialogueBox ]
        @fprintf(stderr, "Retrieved: %s\n",str);
        [ objRootDestroyPopup argObject: DialogueBox ]
    END
END #ConfirmButton

OBJECT
NAME: CancelButton
CLASS: clCommandButton
ATTRIBUTES:
    atRootLabel = "Cancel"
ACTIONS:
    CALLBACK ( atRootCallback, callback_cancel )
        [ objRootDestroyPopup argObject: DialogueBox ]
    END
END #CancelButton

# Define the dialogue box
OBJECT
NAME: DialogueBox
CLASS: clDialogueWindow
ATTRIBUTES:
    atRootLabel = "Choose One"
    atDialogueWindowText = "Sample"
ACTIONS:
    [ objRootInstantiate argObject: ConfirmButton argParent DialogueBox
      argPopupType: ATnoPopup ]

    [ objRootInstantiate argObject: CancelButton argParent DialogueBox
      argPopupType: ATnoPopup ]
END #DialogueBox

# Define simple menu items

OBJECT
NAME: MenuItem
CLASS: clItemSimpleMenu
ATTRIBUTES:
    atRootLabel = "TextWindow"
    atItemSimpleMenuCallback = ( callback_item1 )
ACTIONS:
    CALLBACK ( atItemSimpleMenuCallback, callback_item1 )
        [ objRootInstantiate argObject: TextWindow argParent ATtopObject

```

```

                                argPopupType:ATObjectPopup ]

    END #callback
END #MenuItem1

OBJECT
NAME MenuItem2
CLASS clItemSimpleMenu
ATTRIBUTES
    atRootLabel = "GenericWindow"
    atItemSimpleMenuCallback = ( callback_item2 )
ACTIONS.
    CALLBACK ( atItemSimpleMenuCallback, callback_item2 )
        [ objRootInstantiate argObject: GenericWindow argParent: ATtopObject
          argPopupType:ATObjectPopup ]

    END #callback
END #MenuItem2

OBJECT
NAME MenuItem3
CLASS clItemSimpleMenu
ATTRIBUTES
    atRootLabel = "DialogBox"
    atItemSimpleMenuCallback = ( callback_item3 )
ACTIONS
    CALLBACK ( atItemSimpleMenuCallback, callback_item3 )
        [ objRootInstantiate argObject: DialogBox argParent: ATtopObject
          argPopupType:ATObjectPopup ]

    END #callback
END #MenuItem3

INIT
ACTIONS:
    [ objRootInstantiate argObject: OuterBox argParent: ATtopObject
      argPopupType:ATnoPopup ]

    [ objRootInstantiate argObject: LabelButton argParent: OuterBox
      argPopupType:ATnoPopup ]

    [ objRootInstantiate argObject: QuitButton argParent: OuterBox
      argPopupType:ATnoPopup ]

    [ objRootInstantiate argObject: MenuButton argParent: OuterBox
      argPopupType:ATnoPopup ]

    [ objRootInstantiate argObject: WindowMenu argParent: MenuButton

```

```
        argPopupType:ATmenuPopup ]  
[ objRootInstantiate argObject:MenuItem1 argParent:WindowMenu  
  argPopupType:ATnoPopup ]  
[ objRootInstantiate argObject:MenuItem2 argParent:WindowMenu  
  argPopupType:ATnoPopup ]  
[ objRootInstantiate argObject:MenuItem3 argParent WindowMenu  
  argPopupType:ATnoPopup ]  
END #INIT
```


Bibliography

- [1] *The NeXT System Reference Manual (Release 1.0 Preliminary Edition)*. NeXT Inc., 1989.
- [2] H. B. Beretta and et al. XS-1: An Integrated Interactive System and Its Kernel. In *Proceedings 6th International Conference On Software Engineering*, pages 340-349, 1982.
- [3] W. Buxton and et al. Towards a Comprehensive User Interface Management System. *Computer Graphics*, 17(9):35-42, July 1983.
- [4] M. Caplinger and R. Hood. An Incremental Unparser for Structured Editors. In *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, pages 65-74, 1986.
- [5] L. Cardelli. *Building User Interfaces By Direct Manipulation*. Research Report 22, Digital Equipment Corp. Systems Research Center, Palo Alto, Calif., 1987.
- [6] L. Cardelli and R. Pike. Squeak: A Language for Communicating with Mice. In *SIGGRAPH '85 Conference Proceedings*, pages 199-204, July 1985.
- [7] Peterson, C.D. *Athena Widget Set - C Language Interface*. MIT X Consortium, 1989.
- [8] S. Choudhury. *A Fragment Based Program Editor*. Master's thesis, School of Computer Science, McGill University, Montreal, August 1986.
- [9] J. Coutaz. Abstractions for User Interface Design. *IEEE Computer*, 18(9):21-34, September 1985.
- [10] E.G. Davis and R.W. Swezey. Human Factors Guidelines in Computer Graphics: A Case Study. *Int. J. Man-Machine Studies*, 18:113-133, 1983.

- [11] M. Delisle, D.E. Menicosy, and M.D. Schwartz. *Magpie — An Interactive Programming Environment for Pascal*. Technical Report CR-83-4, Computer Research Laboratory, Applied Research Laboratories, Tektronix Inc., 1983.
- [12] L.P. Deutsch and E.A. Taft. *Requirements for an Experimental Programming Environment*. Technical Report CSL-80-10, Xerox PARC, 1980.
- [13] V. Donzeau-Gouge and et al. Programming Environments Based on Structured Editors: The MENTOR Experience. In H. Schrobe D. Barstow and E. Sandwell, editors, *Interactive Programming Environments*, McGraw-Hill Book Company, 1984.
- [14] G. Engels, T. Janning, and W. Schafer. A Highly Integrated Tool Set For Program Development Support. In *Proc. ACM SIGSMALL Conference*, pages 1–10, May 1988.
- [15] G. Engels, M. Nagl, and W. Schafer. On the Structure of Structure-Oriented Editors for Different Applications. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ACM SIGPLAN Notices, 22(1):190–198, January 1987.
- [16] M.A. Flecchia and R.D. Bergeron. Specifying Complex Dialogs in Algol. In *Proc SIGCHI+GI87*, pages 229–234, 1987.
- [17] J. Foley. Transformations on a Formal Specification of User-Computer Interfaces. *Computer Graphics*, 109–112, April 1987.
- [18] J. Foley and et al. Defining Interfaces at a High Level of Abstraction. *IEEE Software*, 6(1):25–32, January 1989.
- [19] J.D. Foley. The Structure of Interactive Command Languages. In *Proceedings of the IFIP Workshop on the Methodology of Interaction*, pages 227–234, North Holland Publ., Amsterdam, 1980.
- [20] J.D. Foley and et al. A Knowledge-Based User Interface Management System. In *Proceedings of the ACM CHI'88 Conference on Human Factors in Computing Systems*, pages 67–72, May 1988.
- [21] J.D. Foley, W.C. Kim, and C.A. Gibbs. Algorithms to Transform the Formal Specification of a User-Computer Interface. In *Human-Computer Interaction - INTERACT '87*, pages 1001–1006, 1987.

- [22] C. Gibbs, W.C. Kim, and J. Foley. *Case Studies in the Use of IDL: Interface Definition Language*. Technical Report GWU-IIST-86-30, Dept. of EE & CS, George Washington University, Washington, D.C., 1986.
- [23] A. Goldberg and D. Robson. *Smalltalk 80: The Language and Its Implementation*. Addison-Wesley, 1983.
- [24] M. Green. Report on Dialogue Specification Tools. In G.E. Pfaff, editor, *User-Interface Management Systems*, pages 9-20, Springer-Verlag, 1985.
- [25] M. Green. The University of Alberta User Interface Management System. In *SIGGRAPH '85 Conference Proceedings*, pages 205-213, July 1985.
- [26] W.J. Hansen. User Engineering Principles for Interactive Systems. In *AFIPS Fall Joint Computer Conference Proceedings*, pages 523-532, 1971.
- [27] H.R. Hartson and D. Hix. Human-Computer Interface Development: Concepts and Systems for its Management. *ACM Computing Surveys*, 21(1):5-92, March 1989.
- [28] H.R. Hartson, D. Hix, and R.W. Ehrich. A Human-Computer Dialogue Management System. In *Proceedings of INTERACT '84, First IFIP Conference on Human-Computer Interaction*, pages 57-61, International Federation for Information Processing, 1984.
- [29] P. Hayes, E. Ball, and R. Reddy. Breaking the Man-Machine Communication Barrier. *IEEE Computer*, 14(3):19-30, March 1981.
- [30] P. Hayes and P. Szekely. Graceful Interaction Through the COUSIN Command Interface. *International Journal of Man-Machine Studies*, 19(3):285-305, September 1983.
- [31] P.J. Hayes. Executable Interface Definitions Using Form-Based Abstractions. In R.H. Hartson, editor, *Advances in Human Computer Interaction, Volume 1*, pages 161-190, Ablex Publishing Corporation, Norwood, New Jersey, 1985.
- [32] P.J. Hayes, P.A. Szekely, and R.A. Lerner. Design Alternatives for User-Interface Management Systems Based on Experience with COUSIN. In *Proc. SIGCHI 85*, pages 169-175, 1985.
- [33] R.D. Hill. Supporting Concurrency, Communication, and Synchronization in Human-Computer Interaction — The Sassafras UIMS. *ACM Transactions on Graphics*, 5(3):179-210, July 1986.

- [34] Apple Computer Inc. *Inside Macintosh, vol. I*. Addison-Wesley, Reading, Mass., 1985.
- [35] R.J.K. Jacob. A Specification Language for Direct-Manipulation Interfaces. *ACM Transactions on Graphics*, 5(4):283-317, October 1986.
- [36] R.J.K. Jacob. A State-Transition Diagram Language for Visual Programming. *IEEE Computer*, 18(8):51-59, August 1985.
- [37] R.J.K. Jacob. Using Formal Specifications in the Design of a Human-Computer Interface. *Communications of the ACM*, 26(4):259-264, April 1983.
- [38] Richards, J.N.J and et al. On Methods for Interface Specification and Design. *International Journal of Man-Machine Studies*, 24:545-568, 1986.
- [39] B.W. Kernighan and J.R. Mashey. The UNIX Programming Environment. *IEEE Computer*, 14(4):12-24, April 1981.
- [40] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.
- [41] C. Lewerentz and M. Nagl. A Formal Specification Language for Software Systems defined by Graph Grammars. In *Proceedings of WG'84 Workshop on Graphtheoretic Concepts in Computer Science*, pages 224-241, Linz, 1984.
- [42] N.H. Madhavji. Fragtypes: A Basis for Programming Environments. *IEEE Transactions on Software Engineering*, 14(1):85-97, January 1988.
- [43] N.H. Madhavji. Operations for Programming in the small. In *IEEE 8th International Conference on Software Engineering*, pages 15-25, August 1985.
- [44] N.H. Madhavji, S. Choudhury, R. Robson, and N. Friedman. On Commands for an Integrated Programming Environment. In K. Hopper and I.A. Newman, editors, *Foundations for Human-Computer Communication*, pages 407-423, North-Holland Publishing Co., 1986.
- [45] N.H. Madhavji, N. Leoutsarakos, and D. Vouliouris. Software Construction Using Typed Fragments. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, pages 163-178, Springer Verlag, 1985.
- [46] N.H. Madhavji, L. Pinsonneault, and K. Toubache. Modula-2/MUPE-2 Language and Environment Interactions. *IEEE Software*, 3(6):7-17, November 1986.

- [47] N.H. Madhavji, L. Pinsonneault, K. Toubache, and J. Desharnais. A New Approach to Cursor Movements in User Interfaces of Integrated Programming Environments. *Information and Software Technology*, 30(9):535-546, November 1988.
- [48] N.H. Madhavji, M. Zhang, S. Boulos, and G. Yuan Xiang. Semi-structured Cursor Movements in MUPE-2. *Software Engineering Journal*, 4(6):309-317, November 1989.
- [49] J. McCormack and P. Asente. X.11 Toolkit for the X Window Manager. In *Proc. ACM SIGGraph Symp. User-Interface Software*, pages 46-55, 1989.
- [50] J. McCormack, P. Asente, and R. Swick. *X Toolkit Intrinsics - C Language Interface*. MIT X Consortium, 1989.
- [51] B. Meyer. Reusability: The Case for Object-oriented Design. *IEEE Software*, 4(2):50-64, March 1987.
- [52] B. Meyer, J. Nerson, and S.H. Ko. Showing Programs on a Screen. *Science of Computer Programming*, 5:111-142, 1985.
- [53] M. Mikelsons. Prettyprinting in an Interactive Programming Environment. In *Proc. SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 108-116, June 1981.
- [54] B.A. Myers. Creating Interaction Techniques by Demonstration. *IEEE Computer Graphics and Applications*, 51-60, September 1987.
- [55] B.A. Myers. User-Interface Tools: Introduction and Survey. *IEEE Software*, 6(1):15-23, January 1989.
- [56] B.A. Myers and W. Buxton. Creating Highly-Interactive and Graphical User Interfaces by Demonstration. In *SIGGRAPH '86 Conference Proceedings*, pages 249-258, August 1986.
- [57] I. Nassi and B. Schneiderman. Flowchart Techniques for Structured Programming. *ACM SIGPLAN Notices*, 8(8), August 1973.
- [58] P. Naur. Revised Report on the Algorithmic Language ALGOL 60. *Communications of the ACM*, January 1963.
- [59] J. Nievergelt. Errors in Dialogue Design and How to Avoid Them. In *Document Preparation Systems*, pages 1-10, North Holland, 1983.

- [60] K. Normark. *Programming Environments — Concepts, Architectures, and Tools*. Technical Report R 89-5, Institute of Electronic Systems, Aalborg University, Denmark, 1989.
- [61] D. Notkin. The Gandalf Project. *The Journal of Systems and Software*, 5(2) 91–105, May 1985.
- [62] D.R. Olsen. MIKE: The Menu Interaction Kontrol Environment. *ACM Transactions on Graphics*, 5(4):318–344, October 1986.
- [63] D.R. Olsen and E.P. Dempsey. SYNGRAPH: A Graphical User-Interface Generator. In *SIGGRAPH '83 Conference Proceedings*, pages 43–50, July 1983.
- [64] D.L. Parnas. On the Use of Transition Diagrams in the Design of a User Interface for an Interactive Computer System. In *Proc. 24th National ACM Conference*, pages 379–385, 1969.
- [65] G.E. Pfaff(ed.). *User-Interface Management Systems*. Springer-Verlag, 1985.
- [66] L. Pinsonneault. *Data Structures for a Programming Environment*. Master's thesis, School of Computer Science, McGill University, Montreal, July 1987.
- [67] P. Reisner. Formal Grammar and Human Factors Design of an Interactive Graphics System. *IEEE Trans. Software Eng.*, SE-7(2):229–240, March 1981.
- [68] S.P. Reiss. Graphical Program Development with PECAN Program Development Systems. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 30–41, April 1984.
- [69] D. Rosenthal and A. Yen. User Interface Models Summary. *Computer Graphics*, 17(3):16–20, January 1983.
- [70] R.W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, 5(3):79–109, April 1986.
- [71] B. Schneiderman. Designing Menu Selection Systems. *Journal of the American Society for Information Sciences*, 37(2):57–70, March 1986.
- [72] B. Schneiderman. *Designing the User Interface*. Addison-Wesley, Reading, Mass., 1987.

- [73] B. Schneiderman. Direct Manipulation: A Step Beyond Programming Languages. *IEEE Computer*, 57-69, August 1983.
- [74] J.L. Sibert, W.D. Hurley, and T.W. Bleser. An Object-Oriented User Interface Management System. In *SIGGRAPH '86 Conference Proceedings*, pages 259-267, August 1986.
- [75] D.C. Smith and et al. Designing the Star User Interface. *BYTE*, 7(4):242-282, April 1982.
- [76] P.P. Tanner and W.A.S. Buxton. Some Issues in Future User Interface Management System (UIMS) Development. In G.R. Pfaff, editor, *User Interface Management Systems*, pages 67-79. Springer-Verlag, 1985.
- [77] T. Teitelbaum and T. Reps. The Cornell Program Synthesizer: A Syntax Directed Programming Environment. *Communications of the ACM*, 24(9):563-573, September 1981.
- [78] W. Teitelman. A Display-Oriented Programmer's Assistant. *International Journal of Man Machine Studies*, 11(2):157-187, March 1979.
- [79] W. Teitelman. A Tour Through Cedar. In *7th International Conference on Software Engineering*, pages 181-195, March 1984.
- [80] W. Teitelman and L. Masinter. The Interlisp Programming Environment. *IEEE Computer*, 14(4):25-33, April 1981.
- [81] A. Tesler. The Smalltalk Environment. *BYTE*, 7(4), 1981.
- [82] J. Van Den Bos. Abstract Interaction Tools: A Language for User Interface Management Systems. *ACM Transactions on Programming Languages and Systems*, 10(2):215-247, April 1988.
- [83] A.I. Wasserman. Extending Transition Diagrams for the Specification of Human-Computer Interaction. *IEEE Trans. Softw. Eng.*, SE-11(8), August 1985.
- [84] A.I. Wasserman and D.T. Shewmake. Rapid Prototyping of Interactive Information Systems. *SIGSOFT Software Engineering Notes*, 171-180, December 1982.
- [85] N. Wirth. *Programming in Modula-2 (Third Corrected Edition)*. Springer-Verlag, 1985.